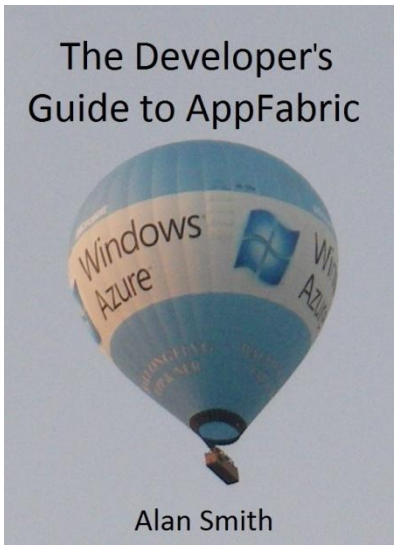
# The Developers Guide to AppFabric

*October 2011*

## Introduction




*Windows Azure hot air balloon over Stockholm City, August 2nd 2011.*

"The Developer's Guide to AppFabric" is a free e-book for developers who are exploring and leveraging the capabilities of the Azure AppFabric platform.

The goal is to create a resource that will evolve and mature in parallel with the Azure AppFabric technologies. The use of an electronic format will allow sections to be added as new technologies are released and improved as the knowledge and experience of using the technologies grows within the developer community.

The first version, published on the 3th October 2011, marks seven years to the day since the first version of "The Blogger's Guide to BizTalk" was published.

The first section will provide an introduction to the brokered messaging capabilities available in the Azure AppFabric September 2011 release. The next section will go into a lot more depth and explore the brokered messaging feature set. Future sections will cover the Access Control Service, relayed messaging, and cache. Features like the application model and integration services will be covered as they emerge and are released.

The book will always be free to download and available in CHM and PDF format, as well as a web based browsable version. The planned release schedule for the remainder of 2011 is to update the guide with new content monthly, around the start of each month. Updates will be made available on the website and announced through my blog and twitter.

Developer's Guide to AppFabric: http://www.cloudcasts.net/devguide/

Twitter: @alansmith

I will publish content from future releases of the guide as articles on my blog.

Blog: http://geekswithblogs.net/asmith

I have recorded some of the code examples in the guide and published them as webcasts on the CloudCasts website.

CloudCasts: cloudcasts.net

**About the Author**

Alan Smith is from the north of England, and living in Stockholm Sweden. He has been working as a developer with Microsoft technologies since 1995 and specializes in "Connected Systems" technologies such as BizTalk Server, WCF, WF, Windows Azure and AppFabric. He is a certified trainer, and authors and delivers courses in BizTalk Server, AppFabric, Windows Azure, WCF and WF.

He has been a Microsoft MVP for seven years, the first four a BizTalk Server and the last three as a "Connected Systems Developer" MVP. He is an active member of the Microsoft Application Server Group advisor team and provides feedback to the development teams on emerging AppFabric technologies.

Alan hosts the website CloudCasts.net, a community webcast site covering Microsoft "Connected Systems" technologies. He is a regular speaker at conferences and local user group events and co-manages the Sweden Windows Azure Group (SWAG).

**Providing Feedback**

As "The Developer's Guide to AppFabric" is a community resource published in electronic format any feedback provided by readers can be used to improve the quality of the next release. Whilst I have tried to make an effort to ensure that all the content is correct I have had to make a number of assumptions relating to features that are currently not fully documented. If you have any comments or suggestions that could improve the content of the current or future versions of the guide please let me know and I will try to ensure that content is updated for the next release.

Any questions, comments or questions can be sent using the feedback form on my blog.

**Contact the Author**

I like to play an active role in the development community and respond to many questions from developers around the world who are working with Microsoft technologies.

Please feel free to contact me with any questions relating to AppFabric development or the training I deliver.

Email: cloudcasts.net@gmail.com

## Review Team

As this guide is a community project, all feedback and comments are welcome. The following people have provided valuable feedback on the guide and helped to improve the quality of the content and code samples.

**Michael  Stephenson**

Michael is an independent consultant based in the UK.  He is primarily focused around the SOA/BPM technologies in the Microsoft space including BizTalk, WCF, WF, and has a strong interested in upcoming technologies.

http://geekswithblogs.net/michaelstephenson

**Roman Kiss**

Roman is an independent consultant in South California. He focuses on loosely coupled event driven connected architectures using the edge of the MS Technologies.

**Neil Mackenzie**

http://convective.wordpress.com/

## Azure AppFabric Services

Windows Azure AppFabric provides a number of middle services that can be leveraged by cloud-based and on-premise applications. The services that are currently available are:

- Relayed messaging
- Brokered messaging
- Access control service
- Cache

It is expected that the AppFabric platform will evolve rapidly with new services becoming available in community technology preview (CTP) and then moving to production. The current services are available in CTP.

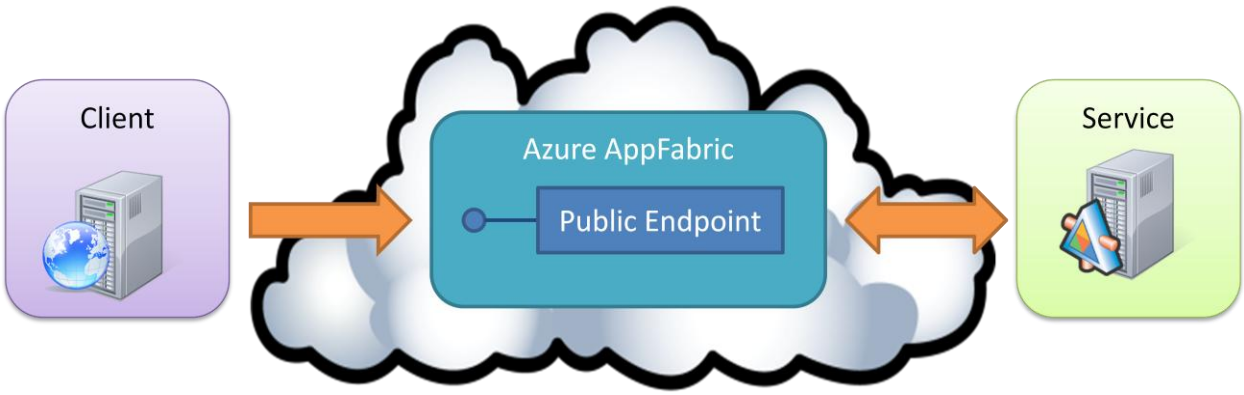
- AppFabric applications

By the end of 2011 we are likely to see a second CTP of AppFabric applications, and a first CTP of AppFabric integration services.

**Relayed Messaging**

The AppFabric relayed capabilities were released as a CTP under the name of "BizTalk Services" back in 2007. In October 2008 they were re-branded as ".NET Services", then rebranded again as "Windows Azure AppFabric Service Bus Relayed Messaging" before the production release in January 2010.

The implementation of relayed messaging has evolved from the original" BizTalk Services" CTP, but the principle remains the same. A service hosted behind a firewall or NAT can expose a public endpoint "in the cloud" that can be consumed by any authenticated client that has access to the internet.




This provides some great capabilities that applications can leverage to enable connectivity in situations where it would typically be challenging or impossible to achieve.

Although relayed messaging provides a lot of advantages it is not without its drawbacks.

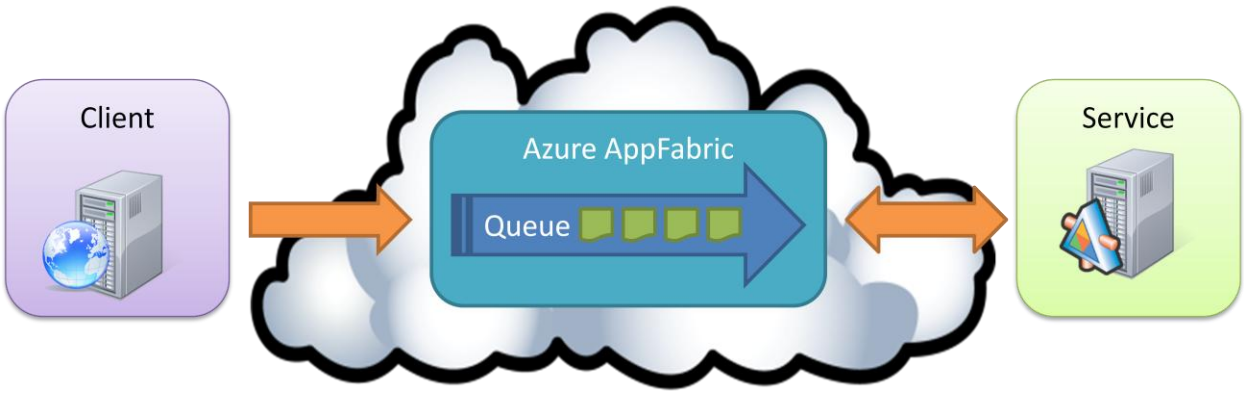
If the service is not running or not connected to the AppFabric service bus the public endpoint is not available.

High bursts of traffic from clients can result in reduced performance and timeouts.

It is challenging to implement load-balancing on the service, limiting scalability and availability.

**Brokered Messaging**

Brokered messaging derives its name from the fact that there is a broker, or intermediary in the message channel that provides durable storage of in transit messages.



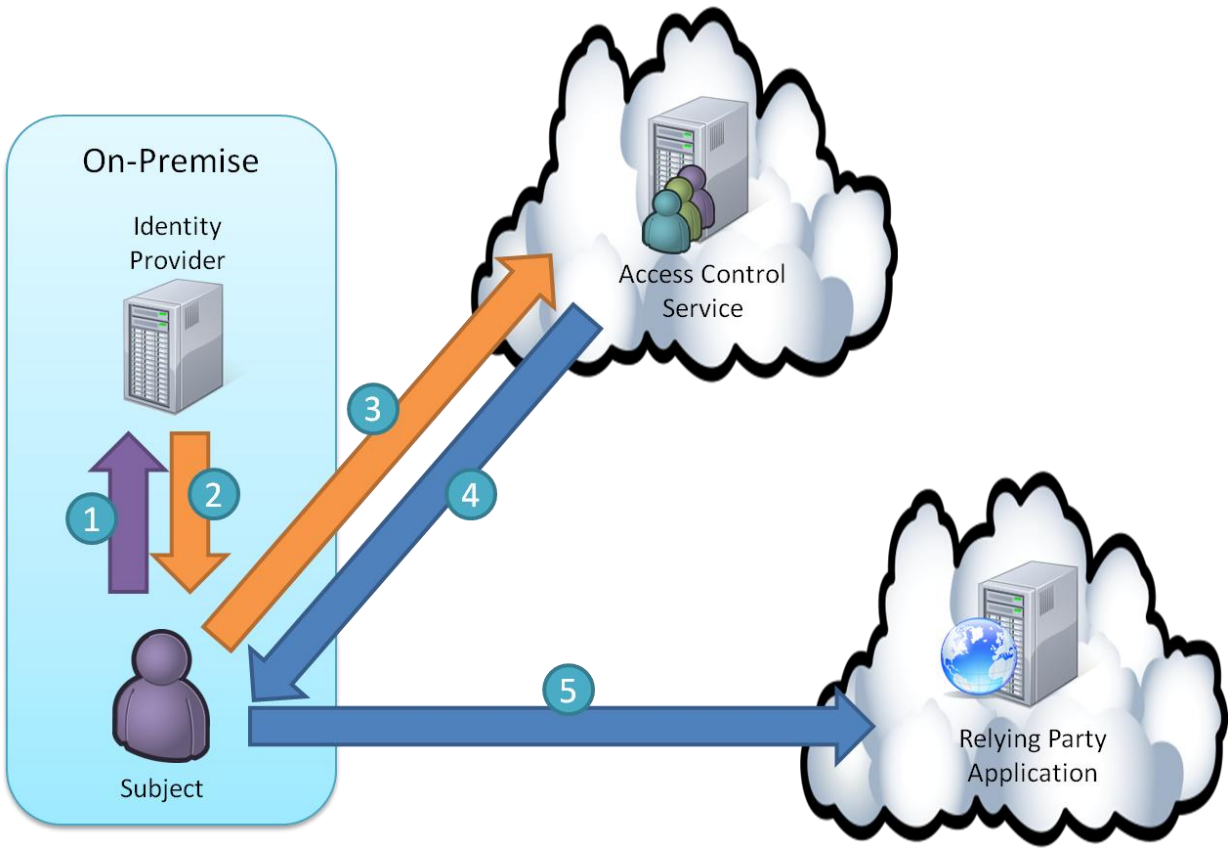

In AppFabric brokered messaging queues or topics and subscriptions can be used to broker messages between a client and a service. This provides a number of advantages.

If the service goes offline the client can still send messages to the queue, and is not aware of any service disruption. This allows the clients perception of service availability to be dependent on the availability of the brokered messaging service, and not the service processing the messages.

If there is a large burst in activity the queue can keep messages in a durable store until they are processed.

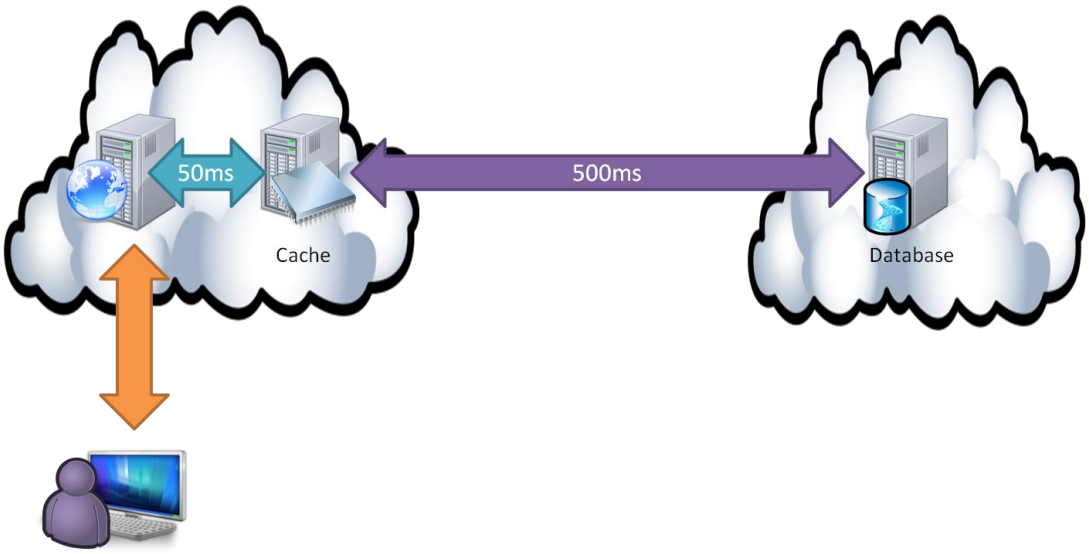
It is easy add additional service instances to receive and process messaging, providing scalability and availability.

**Access Control Service**




**Cache Service**

The AppFabric cache service has been developed form the codename “Velocity” caching technology. The cache is available in both Windows Server AppFabric and Windows Azure AppFabric.

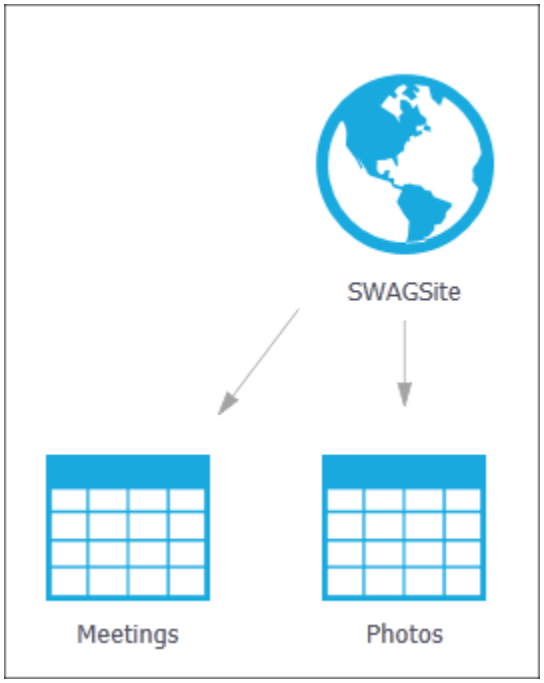


The cache service can be used as a fast access store for data that would otherwise take time to retrieve. Using a cache will improve performance and provide a more responsive experience for the users. As could based services can incur charges for transactions and bandwidth, using a cache in the correct scenario can also reduce costs.

The AppFabric cache can also be used as a session store when session state needs to be maintained between multiple web role instances in an Azure application.

**AppFabric Applications**

AppFabric applications were introduced June CTP of Azure AppFabric. The CTP featured Visual Studio tools to allow the modeling and local testing of AppFabric applications.




The SDK and developer tools were freely available but the capabilities to deploy and test applications in the Azure hosting environment was limited to 500 users. It is hoped that a second CTP of AppFabric applications will be released before the end of 2011 and that hosting capabilities will be available to more users. It is expected that significant changes will have been made to the application model in this release.

**Integration Services**

AppFabric integration services will deliver cloud-based capabilities similar to those available in BizTalk Server. Messaging capabilities such as message translation and transformation will be available as well as business aligned capabilities such as business activity monitoring and trading partner management.

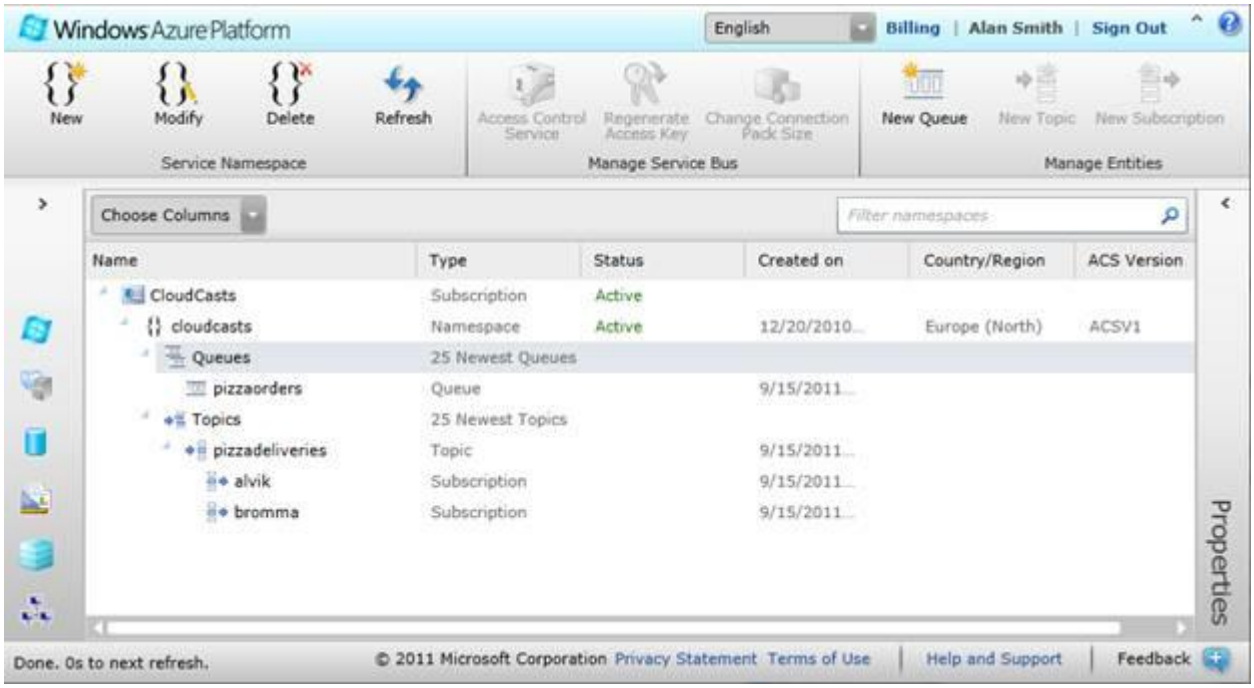
It is expected that the first CTP of AppFabric integration services will be released before the end of 2011.

## Brokered Messaging

The Azure AppFabric brokered messaging service adds durable queues and publish-subscribe messaging to the AppFabric service bus.

Queuing technologies have traditionally been used to communicate messages between applications asynchronously. On the Microsoft platform MSMQ is a common choice as it is part of the Windows operating system. In cross platform scenarios IBM MQ Series is often chosen, as a queuing technology that can be used in a number of operating systems. For integration scenarios BizTalk Server offers an enterprise class publish-subscribe messaging engine and integration with many line-of-business applications. The Windows Azure storage services also provide message queuing functionality that can be leveraged by cloud and on-premise applications.


Windows Azure AppFabric service bus brokered messaging provides enterprise class queuing functionality that can be used in cloud-based applications and also leveraged by on-premise applications. The messaging capabilities provide traditional queues based on the first in first out (FIFO) model based on queues, as well as a publish-subscribe model based on topics and subscriptions. The message service endpoints are exposed as REST endpoints, and .NET programming APIs are available for interacting with the queues directly as well as using WCF bindings to implement queued services.



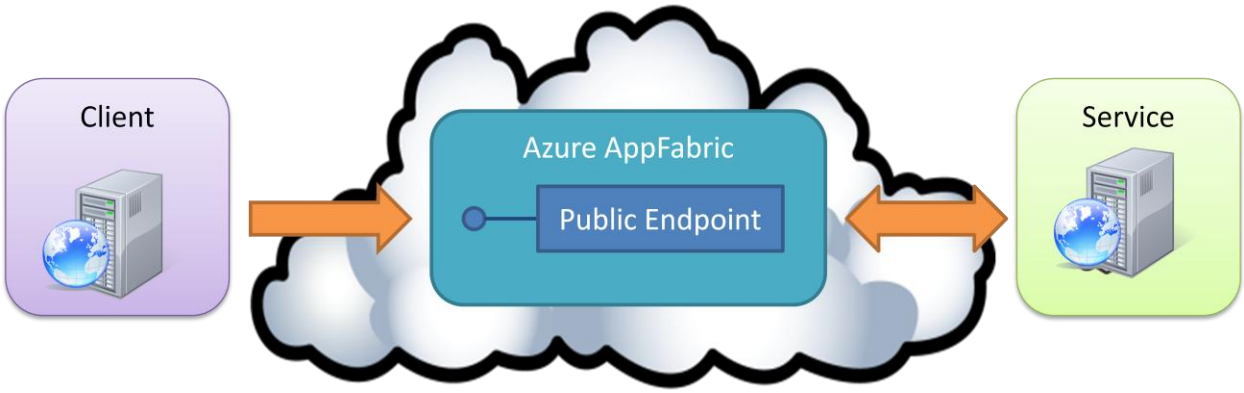

## Relayed Messaging vs. Brokered Messaging

The Azure AppFabric service bus now contains two distinct messaging models, relayed messaging and brokered messaging.

**Relayed Messaging**

The AppFabric relayed capabilities were released as a CTP under the name of “BizTalk Services” back in 2007. In October 2008 they were re-branded as “.NET Services”, then rebranded again as “Windows Azure AppFabric Service Bus Relayed Messaging” before the production release in January 2010.

The implementation of relayed messaging has evolved from the original” BizTalk Services” CTP, but the principle remains the same. A service hosted behind a firewall or NAT can expose a public endpoint “in the cloud” that can be consumed by any authenticated client that has access to the internet.



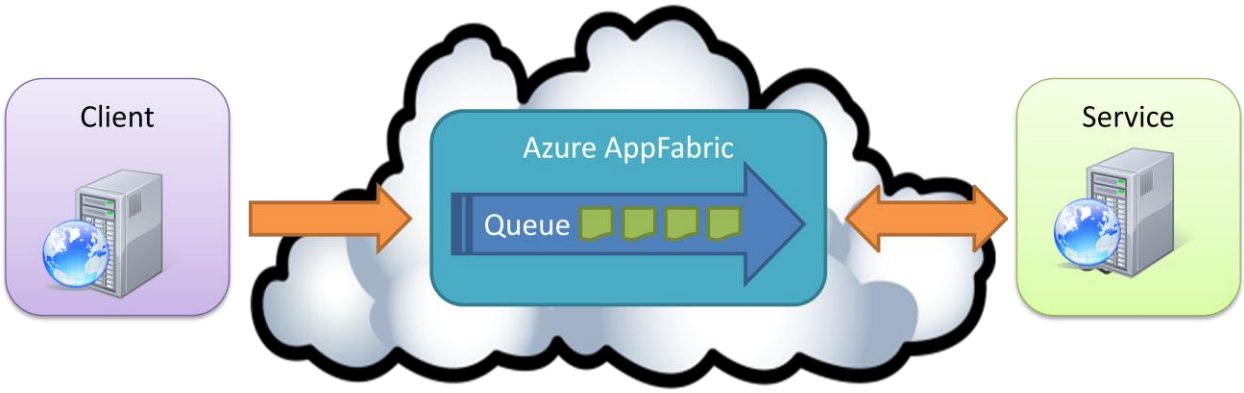
This provides some great capabilities that applications can leverage to enable connectivity in situations where it would typically be challenging or impossible to achieve.

Although relayed messaging provides a lot of advantages it is not without its drawbacks.

- If the service is not running or not connected to the AppFabric service bus the public endpoint is not available.
- High bursts of traffic from clients can result in reduced performance and timeouts.
- It is challenging to implement load-balancing on the service, limiting scalability and availability.

**Brokered Messaging**

Brokered messaging derives its name from the fact that there is a broker, or intermediary in the message channel that provides durable storage of in transit messages.




In AppFabric brokered messaging queues or topics and subscriptions can be used to broker messages between a client and a service. This provides a number of advantages.

- If the service goes offline the client can still send messages to the queue, and is not aware of any service disruption. This allows the clients perception of service availability to be dependent on the availability of the brokered messaging service, and not the service processing the messages.
- If there is a large burst in activity the queue can keep messages in a durable store until they are processed.
- It is easy add additional service instances to receive and process messaging, providing scalability and availability.

**When to Use What**

For services using a request-response messaging pattern relayed messaging is usually the best option. The advantages of storing messages in an intermediary are usually mitigated by the expected response time of the service. It is possible to implement a request-response pattern using brokered messaging

and implementing message correlation, but this adds complexity and additional latency to the implementation.

For asynchronous one-way messaging brokered messaging is typically the best option. The advantages of durable storage, handing bursts of messages and high availability of the brokered messaging service are especially suitable for these scenarios.

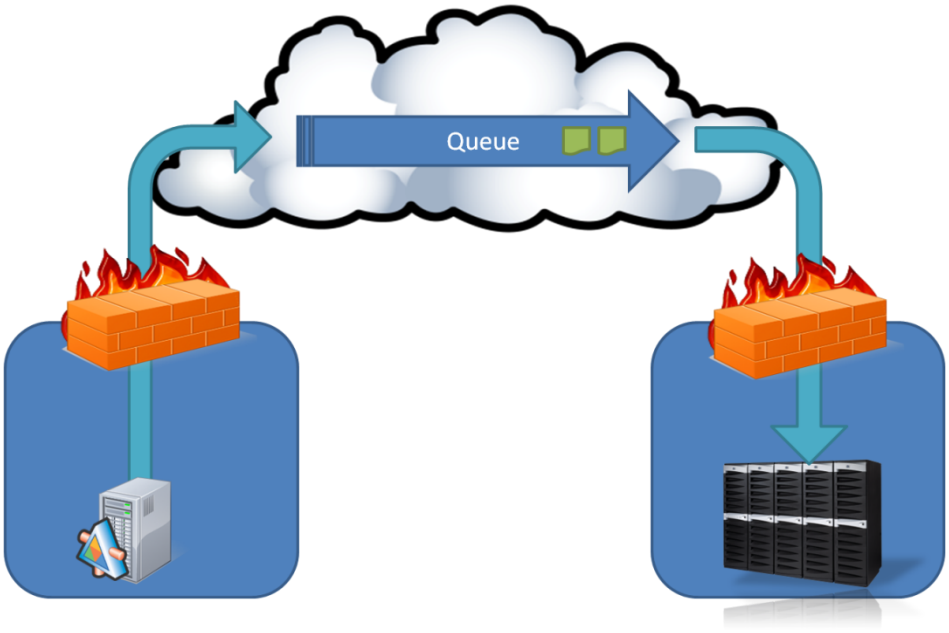
Relayed messaging does include the option to use message buffers to even out brief bursts in load, but as these capabilities are now superseded by brokered messaging and the recommendation is to consider brokered messaging for these types of implementations.

## Messaging Scenarios

Message based systems have many uses in the development of enterprise applications. Reliable and secure communication between applications is often critical, along with the need to handle dynamically varying load levels. There is also a strong demand for integration with web services and legacy line of business applications.

**Secure Network Traversal**

As the AppFabric messaging services are cloud-based, hosted at Microsoft data centers, it makes them ideal for scenarios where messages must be exchanged between applications hosted in different server rooms or organizations. This could include communication between cloud based applications, between cloud-based and on-premise applications, and also between applications hosted on-premise.



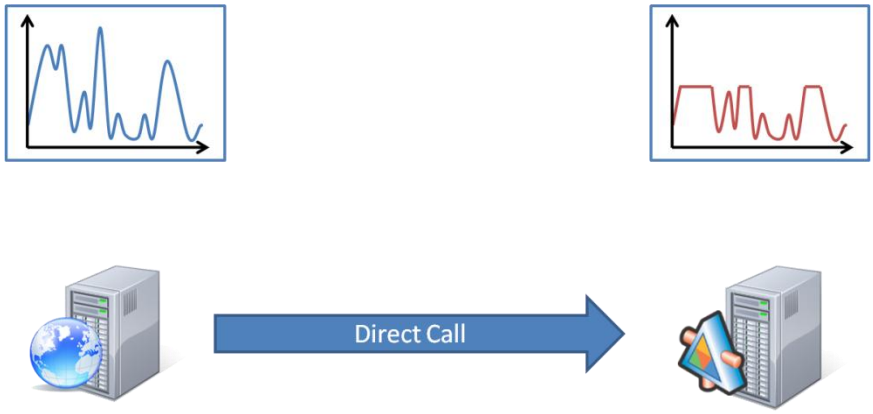

Often on-premise applications are located behind firewalls and proxy servers making direct communication challenging to implement. Outbound communication between these on-premise

applications and the AppFabric messaging entities allows for secure network traversal when enqueueing and dequeuing messages.

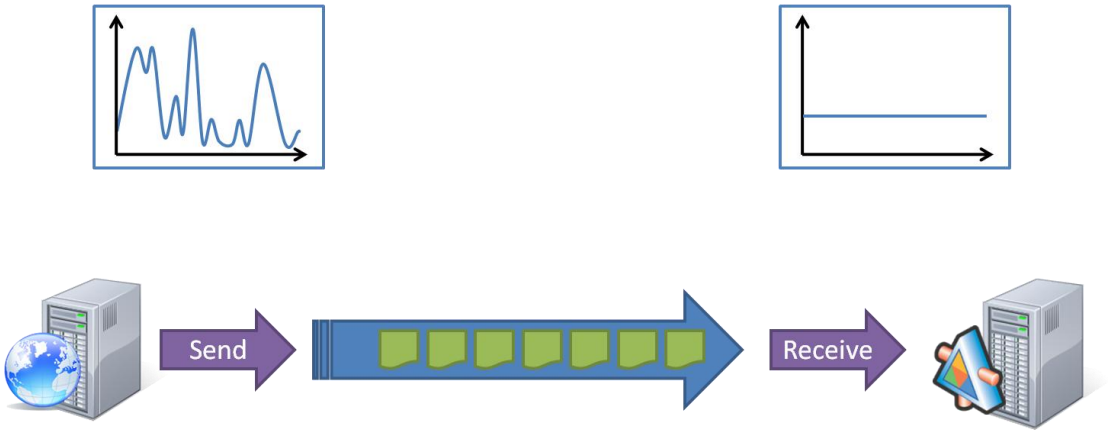
**Load Leveling**

Message queuing can be used to handle scenarios when a large burst of messages is created in a short time period and the messages can be processed in an asynchronous fashion.

Consider a scenario where a large retailer with many branches needs to process end of day transaction messages from cache tills. At closing time all the cache tills in all the branches will send a transaction report, all within a relatively short time period. If a direct service call model was used to process these messages there is a chance that the load would exceed the capacity of the service, resulting in service timeouts and possible loss of data. Providing an environment that has sufficient capacity to handle these large message loads may not be cost efficient and any such environment would be ticking over on idle for the majority of its lifetime.



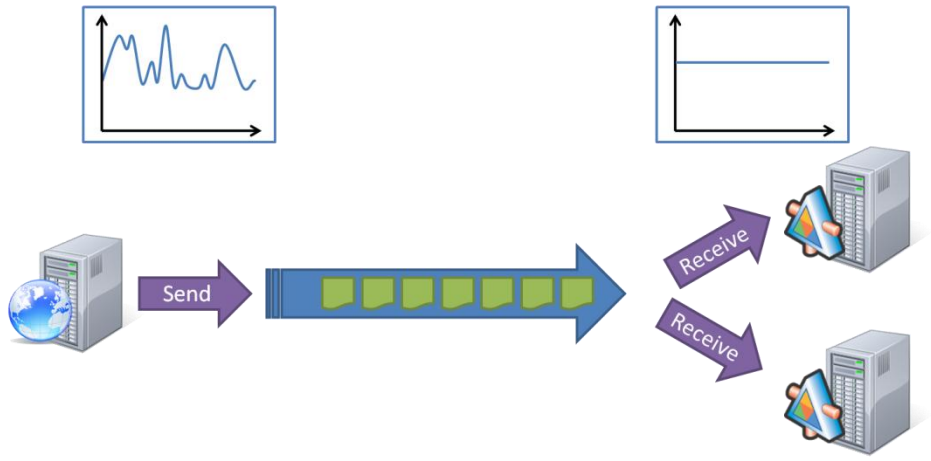

As sales information sometimes does not need to be updated in the line of business systems before start of traiding the next day it may be optimal to process these messages asynchronously.



Creating a queue for the end of day transaction messages and an application to receive and process those messages would allow the messages to be processed during the hours after close of business.

**Load Balancing**

In the above scenario there will be times when the message load could be exceptionally high. Holiday shopping periods and in store promotions are good examples of this. Using a message queuing system makes it easy to balance the message load by adding additional instances of the receiving application.




Using a cloud-based technology such as Azure AppFabric allows instances of the receiving application to easily be provisioned and then removed when required. This provisioning can be automated if required.

**Resilience against Service Failure**

When a queue is used as an intermediary between two applications the system becomes somewhat resilient against the failure of the service that receives and processes the messages. Brief service outages and network interruptions are sometimes to be expected in real world scenarios where parts of the infrastructure are outside the control of the application. Queued messaging can handle these scenarios very well.

There may also be scenarios where there is a severe failure of the receiving application. This could range from a few minutes to several hours. Provided the queue has sufficient capacity to store the in-transit messages whilst the receiving application is restored to an operational state it is possible to recover from these scenarios with no loss of data.

**End of Day Processing**

In other scenarios there may be a trickle of messages during the course of the business day. Instead of providing a service that is available 24/7 to handle this load a queue could be used to accumulate the messages. At the end of the business day a service could be automatically provisioned that would process the messages accumulated on the queue, and then unprovisioned once the workload is complete. Cloud-based technologies are ideal for this scenario, as service provision is quick and can be

automated. If the message processing service is only provisioned for one hour a day there could be a 96% saving in compute resources.

**Integration**

Anyone who has worked with BizTalk Server will understand the power of using asynchronous publish-subscribe messaging when developing integration solutions. In future releases Azure AppFabric will include integration services, which will provide similar pipeline and transform capabilities similar to those found in BizTalk Server. In the current version the AppFabric messaging capabilities can be used in integration scenarios using .NET, WCF and REST based applications.

## Windows Server AppFabric Service Bus Brokered Messaging

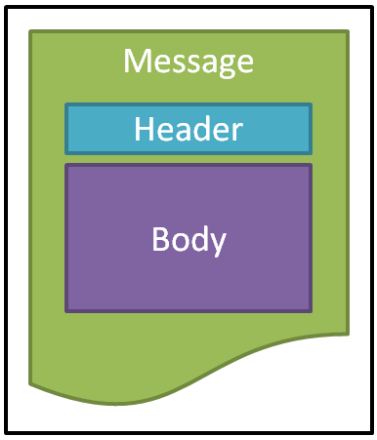
Windows Azure AppFabric service bus messaging provides enterprise class asynchronous messaging capabilities hosted in Windows Azure datacenters. AppFabric service bus relay service is designed to enable synchronous connectivity between web services located in different environments; AppFabric messaging is designed to enable asynchronous communication.

Queue-based communication is not new to Windows Azure, the Windows Azure storage services have provided queues alongside table and blob storage since the Azure CTP was released towards the end of 2008. What is new is the broad range of capacities that AppFabric messaging provides.

The following concepts are used in AppFabric brokered messaging.

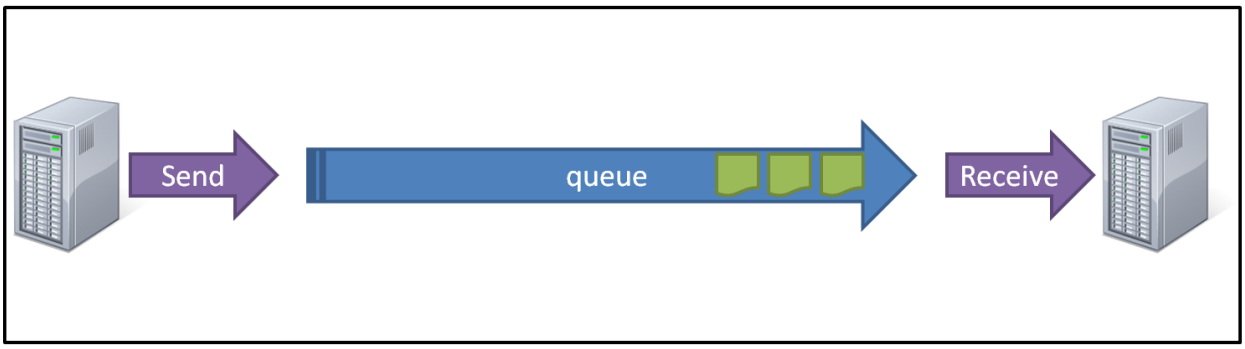
**Messages**

All data communicated through the AppFabric messaging services is encapsulated in messages. Messages contain the data that is being transmitted in the message body, and contextual information in a message header.




**Queues**

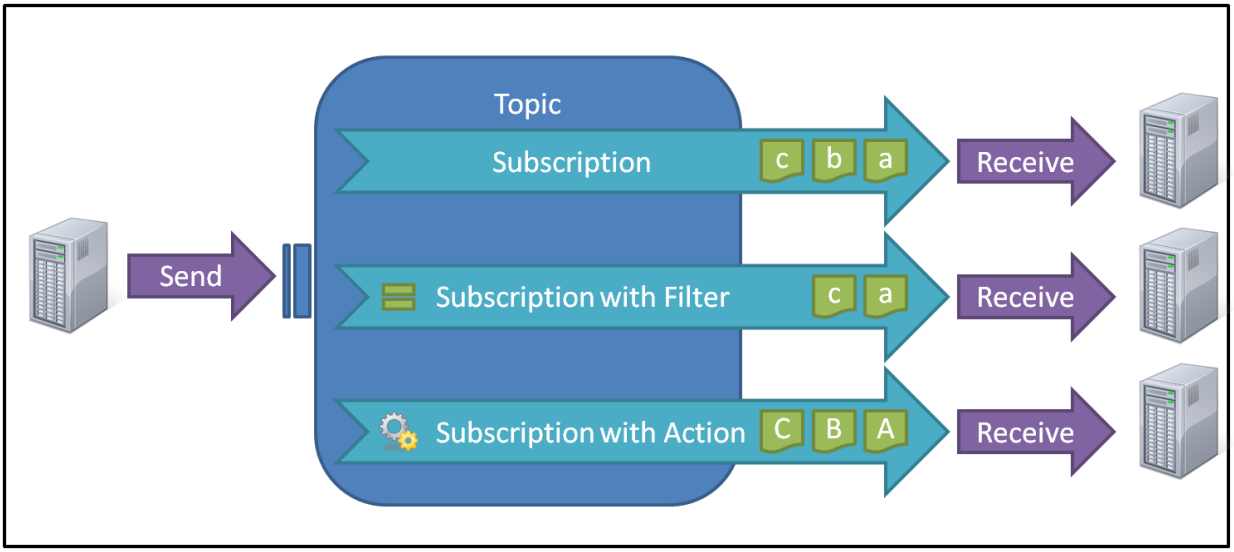
Service bus queues provide a FIFO queuing model similar to MSMQ and the queues in Windows Azure storage services. Messages are placed on a queue by a sending application and received by a receiving application. AppFabric queues are provide durability for messages, meaning that the messages are stored in a durable storage system and are resilient to system failures. AppFabric queue provide options for deadlettering messages and message duplicate detection as well as sessions and correlation.




**Topics and Subscriptions**

Topics and subscriptions provide a publish/subscribe model for exchanging messages between applications. A topic acts as the enqueueing end of a queue, whilst subscriptions act as the dequeueing end of a queue. Subscriptions are created within a topic, and a topic can contain zero or more subscriptions.

A sending application sends messages to a topic, these messages are then routed to zero, one or more subscriptions based on a set of rules. The receiving application can then receive messages from the subscriptions.




Subscriptions can subscribe to all messages in a topic, or use a filter to subscribe messages based on values in the message properties. Subscriptions can also use actions to modify the message property collection.

### *When to Use What*

With two sets queuing capabilities now present in the Azure platform with different APIs, feature sets, and no doubt pricing models one of the challenges for developers is understanding when to use queues in Azure storage services and when to use AppFabric messaging services.

The following table shows a feature comparison of the two services.

| Feature | Azure Storage Queues | AppFabric Messaging |
|---|---|---|
| .NET Programming Model | Azure Storage Client | Direct programming model. |
| REST Interface | Supported | Supported |
| Development Simulator | Supported | |
| WCF Bindings | | Supported |
| Message Serialization | Strings<br>Byte arrays | Serializable objects<br>Streams. |
| Maximum Message Size | 64 KB | 256 KB |
| Durable Messaging | Supported | Supported |
| Maximum Message Lifetime | 1 week | Unlimited |
| Message Expiration | Supported | Supported |
| Publish-Subscribe | | Supported |
| Duplicate Detection | | Supported |
| Message sessions | | Supported |
| Schduled Enqueue Time | Supported | Supported |
| Message deferral | | Supported |
| Dead-Lettering | | Supported. |
| Dequeuing Modes | Get-delete mode. | Peak-Lock<br>Receive and Delete |
| Receive Latency | Receiver acts as a polling consumer. Receive latency is proportional to polling interval. | Receiver acts as a “sticky polling” consumer, receive latency is very low and not related to polling interval. |
| Pricing Model | Based on data center boundary bandwidth, storage used and transactions. | Based in entity hours and message throughput. |

**Azure Storage Service Queues**

Azure storage queues provide a basic asynchronous queue based service that is included in an Azure storage service account. The programming model is simple and similar to the Azure storage table and blob programming model. The billing model for storage service queues is consistent with the other services in Azure storage. Azure storage service queues have an 64 KB limit in message size and 7 days maximum message lifetime, and do not provide the features required for enterprise class messaging systems.

Consider using Azure storage service queues when you require basic queuing functionality with smaller message sizes and lifetimes. As the pricing and programming model is combined with that of other

Azure storage services, they should also be considered using them when other Azure storage services are used.

As the pricing for AppFabric brokered messaging has not been announced it is not possible to compare the costs of using the different queuing systems on this at this point in time.

**Azure AppFabric Service Bus Brokered Messaging**

Azure AppFabric messaging services provide an enterprise class messaging technology that supports more advanced features, such as publish-subscribe, deadlettering, message sessions and duplicate message detection. Although the billing model has not been announced it is likely to be more complex than that of Azure storage services and there is no way yet of understanding the pricing differences. Maximum message size is limited to 256 KB and message lifetime is practically unlimited.

Consider using Azure AppFabric messaging services when developing systems that require the more advanced enterprise messaging features or would be limited by the maximum message size and lifetime imposed by Azure storage queues.

**AppFabric Relay Service Message Buffers**

The AppFabric relay service provides the option to use message buffers to provide temporal decoupling between clients and services using a one-way messaging protocol. Message buffers are stored in memory and therefore are limited in teams of resilience from failures and message expiration.

The messaging functionality provided in the AppFabric messaging services greatly supersedes that of the relay service message buffers and it is therefore recommended to use AppFabric queues, topics and subscriptions instead of relay service message buffers.

It is expected that message buffering will remain a feature in the relay service for the considerable future for compatibility reasons, and there may be scenarios where it is chosen instead of AppFabric messaging services.

### *Comparisons with Other Messaging Technologies*

As well as the queuing service available in Azure storage services there are other technologies available on the Microsoft platform that can be used for asynchronous messaging.

**MSMQ**

Microsoft message queuing (MSMQ) was introduced into the Windows platform in 1997. It provides reliable, durable asynchronous queuing between servers and processes running on the Windows platform. BizTalk Server and WCF provide integration features for MSMQ. MSMQ is included in the Windows license and communication is limited to Microsoft operating systems.

**BizTalk Server**

BizTalk Server is an integration platform that provides asynchronous publish-subscribe messaging, orchestration and business intelligence capabilities. BizTalk Server has been available since 2000, but the current version derives from the architecture of the BizTalk Server 2004 release. Since 2004 BizTalk has been improved substantially and has matured into an enterprise class integration platform used by over 10,000 organizations.

As BizTalk Server is an integration platform by design it has the capability to integrate with a wide range of line-of-business systems using standard and propriety protocols and message formats. Adopting BizTalk Server in an enterprise requires considerable expense for licenses, training and system monitoring.

Microsoft will continue to release future versions of BizTalk server, but the main focus for new developments is to add messaging and integration capabilities to the AppFabric platform.

## *AppFabric Messaging Service Capabilities*

The AppFabric messaging service provides a number of capabilities that make it an attractive choice for implementing enterprise class messaging capabilities.

**Reliable Delivery**

In messaging scenarios the reliability of message delivery is often an important factor. If a messaging system is to be used in business or mission critical applications it is vital that messages sent from one application will be successfully delivered to the destination application.

Different industries and organization will have different definitions of what is reliable. In messaging systems we tend to think that reliable messaging ensures that messages are stored in a durable store and are therefore resilient against system component failures. BizTalk developers are very familiar with the concepts of durable messaging as all messages passing through a BizTalk message channel will be persisted in a SQL database.

Web service developers can work with WS-* standards such as WS-ReliableMessaging and WS-Reliability to improve the reliability of web service operations. BizTalk Server does not support the WS-ReliableMessaging standards as it does not ensure that in-transit messages are persisted in a durable store, and is therefore not considered reliable.

Reliable messaging systems will typically use durable storage to store messages that are in transit. Azure AppFabric queues, topics and subscriptions leverage the high availability data storage services provided by the Windows Azure platform to ensure the messages are persisted reliably.

When considering the reliability of messaging systems it is important to also consider the reliability of the mechanisms that are used to enqueue and dequeue messages. When receiving messages an application can receive the message from a queue, process it, and then mark the message as complete. If the receiving application fails to process the message, it is not marked as complete, and will become visible on the queue again after a specified timeout. This helps to ensure that all messages are received and processed, but can have the side effect that if the receiving application takes too long to process the message it may become visible again for another receiver to process it.

The enqueueing and dequeuing of messages using transactions can help to ensure reliability in messaging applications. When using the direct programming model the sending of a message and the completing of a received message are transactional operations. This allows developers to create applications that leverage distributed transactions to preserve data integrity. An example of this could be to ensure that data is written into a database in the same transaction that a message is marked as complete.

When implementing messaging systems the following semantics can be used to describe message reliability.

*At-Most-Once Delivery*

The postal service is a good example of at-most-once delivery. If I post a letter to a friend it will be delivered to its destination at most once, with a very small chance that it will get lost. There is no way that the receiver will receive two copies of the letter.

In messaging systems at-least-once delivery is the most supported semantic. In enterprise applications it is often not considered sufficiently reliable.

*At Least Once Delivery*

Suppose I photocopy my letter, then post the copy to a friend and ask him to call me once he has received it. If I don't get a call within one week I can make another copy of the letter and resend it and wait another week for a call. This ensures that my friend will get the letter, assuming the address is correct, but there is also a chance that he will be on vacation, or forget to call, or call me when I am out and not leave a message. In this scenario my friend will receive the letter, but may end up getting two or more copies of the same letter.

When using a messaging system that supports at-most-once delivery developers can implement retry logic to ensure that the message reaches the destination. One of the side-effects of implementing this is that there is a chance that more than one copy of the message will be delivered to the destination. In some scenarios this is acceptable, as the receiving of duplicate messages will not affect business processes, in other scenarios it is not.

*Exactly Once Delivery*

Now suppose that I post my friend a copy of the letter, and then call him a week later to check if the letter has arrived. I make sure that he confirms that it has not been delivered before sending another copy of the letter. I also place a codeword on the back of the letter and tell him that if he receives another letter with the same codeword that he should burn it instead of opening it. I now have a way to ensure exactly-once-delivery.

In messaging systems duplicate detection and a two-phase commit protocol can be used to ensure that exactly-once-delivery can be achieved, even when the underlying transport mechanisms only support at-most-once delivery. AppFabric queues, topics and subscriptions provide support for duplicate detection and transactional operations.

*Ordered Delivery*

There are many scenarios where the order in which messages are processed in is critical. Consider a system where order and order update messages for a line-of-business application are received and processed. If the system attempts to process an order update message before an order message is processed the update operation will fail. If two update messages are sent for the same order the updates must be made to the line-of-business system in the order in which they are sent otherwise the older update may overwrite the newer one.

AppFabric queues and topics guarantee ordered delivery of messages, meaning that the message placed on a queue or topic will be received in the same order. There are, however, some exceptions to this. If a receiving application chooses to defer a message then other messages can be received before the deferred message, and deferred messages can be received in any order using message receipts. It is the responsibility of the receiving application to ensure that any required ordering of deferred messages is performed.

Ordered delivery does not necessarily mean ordered processing. In load-balanced scenarios with more than one receiver, the messages may be delivered to the receivers in a sequential order, but there is no guarantee that the receivers will complete the processing of the messages in that order. If this is the case a solution would be to ensure that there is only one active receiver using active-passive clustering, or implement logic to ensure that messages are processed in order.

**Low-Latency**

When developing messaging and integration systems there can be some scenarios where sending, receiving, and processing a message can take a few seconds. In some cases this will have no impact on business operations as the reliability of the message delivery will be the main concern. In other scenarios a messaging system may be used to process request and response messages for customers looking up product information on a website. If the user had to wait more than a second for every page to load they would find it a frustrating experience.

As soon as we add durability into a message channel we also add latency. BizTalk developers are very familiar with this, and the current version of BizTalk Server allows administrators to tune BizTalk to provide lower latency in scenarios where it is required.

AppFabric messaging provides durable messaging with a low latency, however as the messaging infrastructure is hosted in Azure datacenters you will probably find that the latency you experience from AppFabric messaging is proportional to the physical distance between the applications and the datacenter you are using for hosting.

**Availability**

The Azure data centers used to host the AppFabric messaging services are designed to provide high availability. The asynchronous and durable nature of the messaging services allows developers to build applications that provide high availability for their users. Even if the application receiving and processing message from a queue suffers an outage the sending application can still place messages on the queues and topics. The messages will be stored in a durable store and can be received and processed when the receiving application comes back online.

**Scalability**

Imagine a scenario where you build a small proof-of-concept application to implement an asynchronous messaging channel between two applications. The implementation proves to be successful so the technique is used to connect other applications. This improves the operations of the business, so more

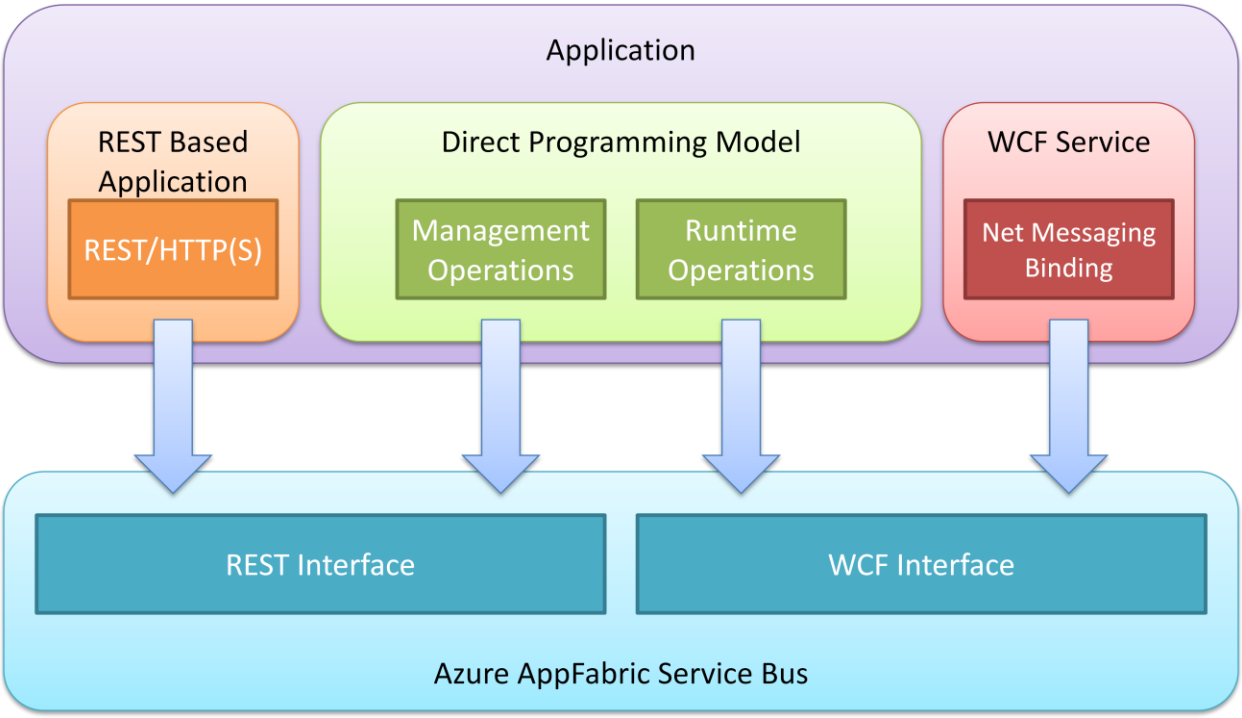
orders are generated, which leads to even more messages. There are also some processes that generate an immense amount of messages at the end of the trading day, and the end of the month. Other processes generate large amounts of traffic whenever the company places an advert on prime-time TV.

If you had chosen an on-premise application there would be a challenge in ensuring the production environment you built to host the proof-of-concept application can be scaled to handle the new load that is placed on it, and also be able to handle a projected load increase in the future. The predictable and sometimes unpredictable bursts of messages from end of day processes and advertizing campaigns could mean that you are required to invest in a substantial production environment that spends most of its time running at less than 5% of their capacity.

As AppFabric messaging services are hosted in Windows Azure datacenters they are able to scale automatically. Sudden increases in message load will be managed by the Azure hosting environment and additional resources allocated to handle the demand. As you pay for the capacity that you use you will incur less charges when the message load is lower and more for the few minutes a day that you actually need a high throughput capacity.

### *Programming Models*

AppFabric messaging provides four programming models that provide interfaces for applications to access the management and runtime functionality of the messaging artifacts.




**REST Interface**

The REST interface provides access to management and runtime functionality using representational state transfer over secure HTTP. This model allows applications built on non .NET technologies to interact with the AppFabric messaging artifacts.

**Direct Model**

The direct programming model provides a set of .NET classes that can be used to manage and use service bus messaging artifacts from .NET applications. The majority of the code samples in this chapter will use this model.

**WCF Model**

The WCF programming model involves the use of a WCF binding to create queued services using the service bus messaging infrastructure. The NetMessagingBinding class can be used by WCF services and WCF clients to send and receive messages using service bus queues, topics and subscriptions as the transport layer. This allows WCF services to integrate directly with the service bus messaging infrastructure.

**Azure AppFabric Application Model**

The Azure AppFabric application model was introduced in the Windows Azure AppFabric June 2011 CTP. The model allows developers to compose, deploy, manage and monitor applications in Azure AppFabric environments.

The AppFabric Application Designer allows developers to add queues, topics and subscriptions to an application and reference them from other services. This has the advantage of abstracting a lot of the code required to create references to the messaging artifacts and also ensure that they are provisioned when the application is deployed.

Future CTP releases of the AppFabric Applications technology will hopefully be available before it ships to production.

### *AppFabric Brokered Messaging Quota Limitations*

In the current AppFabric brokered messaging environment the following limitations have been set.

| | |
|---|---|
| Maximum Queue or Topic Size | 5 GB |
| Maximum message size | 256 Kb |
| Maximum message header size | 64 Kb |
| Maximum queues or topics per namespace | 10,000 |
| Maximum subscriptions per topic | 2,000 |
| Maximum number of SQL filters per topic | 2000 |
| Maximum number of correlation filters per topic | 100,000 |
| Maximum concurrent connections on a queue, topic or subscription | 100 |

Whilst many of these limitations will not impact the development of applications using the messaging services the message size limitation of 256 Kb may cause issues in some scenarios. It is worth noting that when using data contract serialization when constructing messages the binary serialization will be much more efficient that XML serialized messages used in other messaging systems. When working with large messages it is required to break down the data into a number of smaller messages and correlate the messages together. Techniques to do this are described later in the chapter.

## Direct Programming Model

The service bus programming model provides an extensive array of classes that can be used to manage and interact with the service bus messaging artifacts. The classes used for message related functionality are contained in the following namespaces.

- Microsoft.ServiceBus.Messaging
- Microsoft.ServiceBus.Messaging.Configuration

Classes in Microsoft.ServiceBus namespace are also used when creating clients to manage queues topics and subscriptions, and authenticate with the service bus. The messaging namespaces are included in the Microsoft.ServiceBus version 1.5 assembly which is present in the Azure AppFabric September 2011 release.

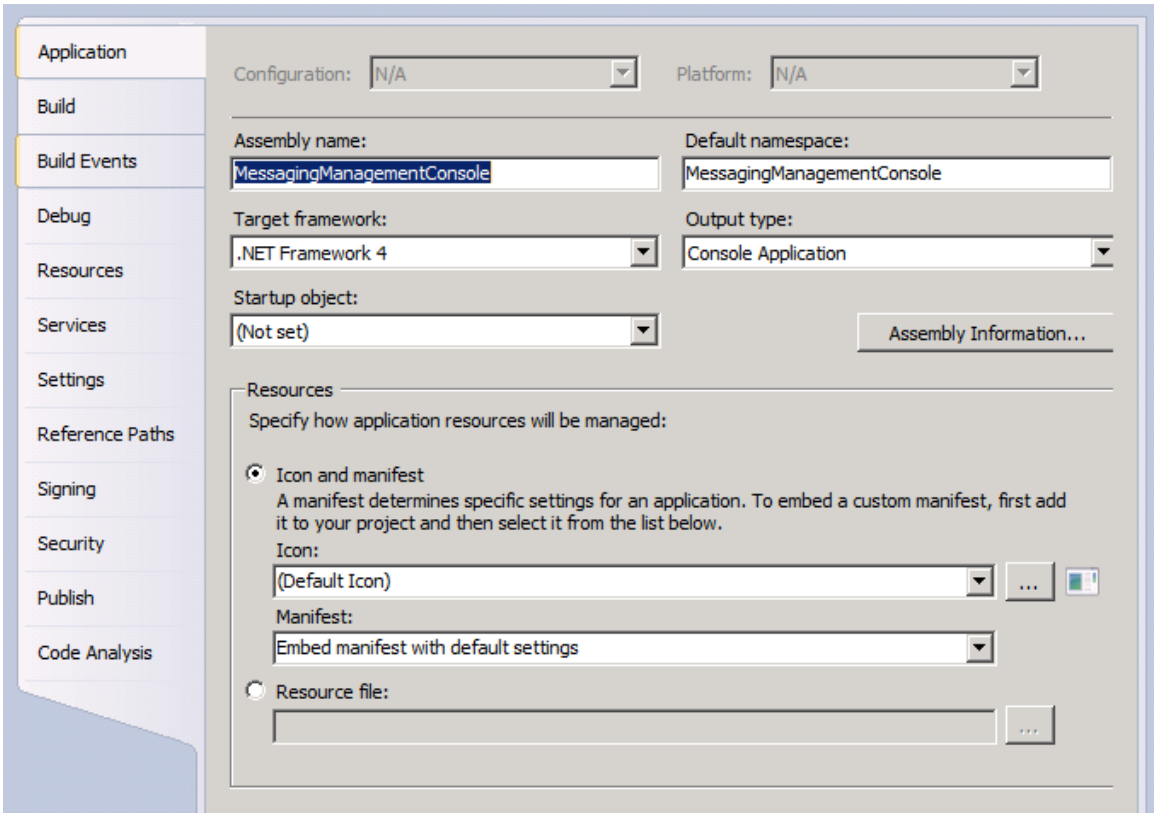
**A Note on the Code Examples**

The majority of the code examples in this chapter will use the synchronous versions of methods, and exception handling will typically not be used. Whilst this is not recommended practice in real-world projects it has the advantage of simplifying the code and allowing the samples to focus purely on the use of the service bus brokered messaging classes.

The use of asynchronous methods and exception handling, along with other coding best practices will be covered in detail elsewhere in this book.

**Project Configuration and Referenced Assemblies**

In order to access the references required for using the direct programming model from Windows applications the target framework of the project must be changed from .NET Framework 4 Client Profile to .NET Framework 4. The majority of samples in this chapter will use C# console applications in order to focus purely on the AppFabric programming model. The project properties for these console applications has been changed as shown in the screenshot below.




Once this is done the following references should be added to the project.

- Microsoft.ServiceBus

As AppFabric brokered messaging will often use WCF and data contract serialization classes internally it is often necessary to also add references to the following.

- System.ServiceModel
- System.Runtime.Serialization

### *Messaging Related Classes*

This section will provide an overview of the classes that will typically be used when developing message based applications using the direct programming model. At the time of writing the documentation for the messaging classes is fairly sparse, so this section can be used as a basic reference. The classes will be covered in much greater details later in the chapter.

#### Microsoft.ServiceBus Namespace

As well as containing many of the classes and WCF bindings for using the service bus relay service this namespace also contains a couple of classes that are used when calling the management and runtime messaging services.

**TokenProvider**

The TokenProvider class provides factory methods to create shared secret and SAML security tokens. When connecting to the service bus it is used to create a shared secret token using the issuer name and key for the account used to access the service bus.

**ServiceBusEnvironment**

The ServiceBusEnvironment class provides static methods and properties that return the URIs and settings for the service bus and access control environments. It is recommended to use these methods rather than hard coding the URI values. The URIs in CTP releases of the AppFabric SDKs will point to the CTP environment, appfabriclabs.com, the URIs in production releases will point to the production environment servicebus.windows.net.

**NamespaceManager**

The NamespaceManager class provides methods for creating, listing and deleting queues, topics and subscriptions in a specified service bus namespace. This class should be used for any management operations on service bus messaging entities.

#### Microsoft.ServiceBus.Messaging Namespace

This namespace contains the majority of the classes used when working with the AppFabric messaging services.

**BrokeredMessage**

This class contains constructors for creating messages from serializable objects and streams, and for creating empty messages. There is a typed method for deserializing messages or getting the message body stream. Various properties that make up the message header are exposed. All messages that pass through the service bus using the direct programming model are instances of BrokeredMessage.

**BrokeredMessageProperty**

This class provides a representation of the properties available on an instance of the BrokeredMessage class.

**MessagingFactory**

This class is a factory class that is used to create instances of clients for Queues, Topics and Subscriptions for a given service bus namespace. Instances of MessagingFactory are created by calling the static Create or BeginCreate methods specifying the address and credentials for the service bus namespace.

**MessagingFactorySettings**

This class contains settings that can be used when creating a MessagingFactory. Using an instance of this class when creating a MessagingFactory allows the default properties, such as OperationTimeout, to be changed.

**QueueClient, TopicClient, SubscriptionClient**

These classes provide client side connections to queues, topic and subscriptions hosted within the service bus namespace. They are created by calling the appropriate factory method in the MessagingFactory class. These client classes have internal message senders and receivers for message based operations. The QueueClient class contains functionality to send and receive messages; the TopicClient class has send functionality and the SubscriptionClient contains receive functionality.

**MessageReceiver, MessageSender**

These classes are used for sending and receiving messages from queues, topics and subscriptions. The MessageReceiver class contains functionality for completing, canceling, deferring and dead-lettering messages.

**MessageSession**

This class derives from the message receiver class and inherits the functionality required to receive messages form queues and subscriptions whilst providing additional support for receiving a correlated session of messages. Instances of the message session class are created by calling the AcceptSession method of the QueueClient class.

**QueueDescription, TopicDescription, SubscriptionDescription**

The description classes are used to specify the properties of queues, topics and subscriptions when they are created using the methods in the NamespaceManager class. Creating and specifying a description when creating an entity will allow the default properties to be overridden.

**CorrelationFilter, TrueFilter, FalseFilter, SqlFilter**

The filter classes are used for defining filter subscriptions between topics and subscriptions when using the publish-subscribe messaging pattern.

**RuleAction, SqlRuleAction, RuleDescription**

The rule action classes are used to modify the properties of a message based on rules when using the publish-subscribe messaging pattern in queues and topics.

**NetMessagingBinding**

The NetMessagingBinding is used by WCF services when exchanging messages with queues, topics and subscriptions.

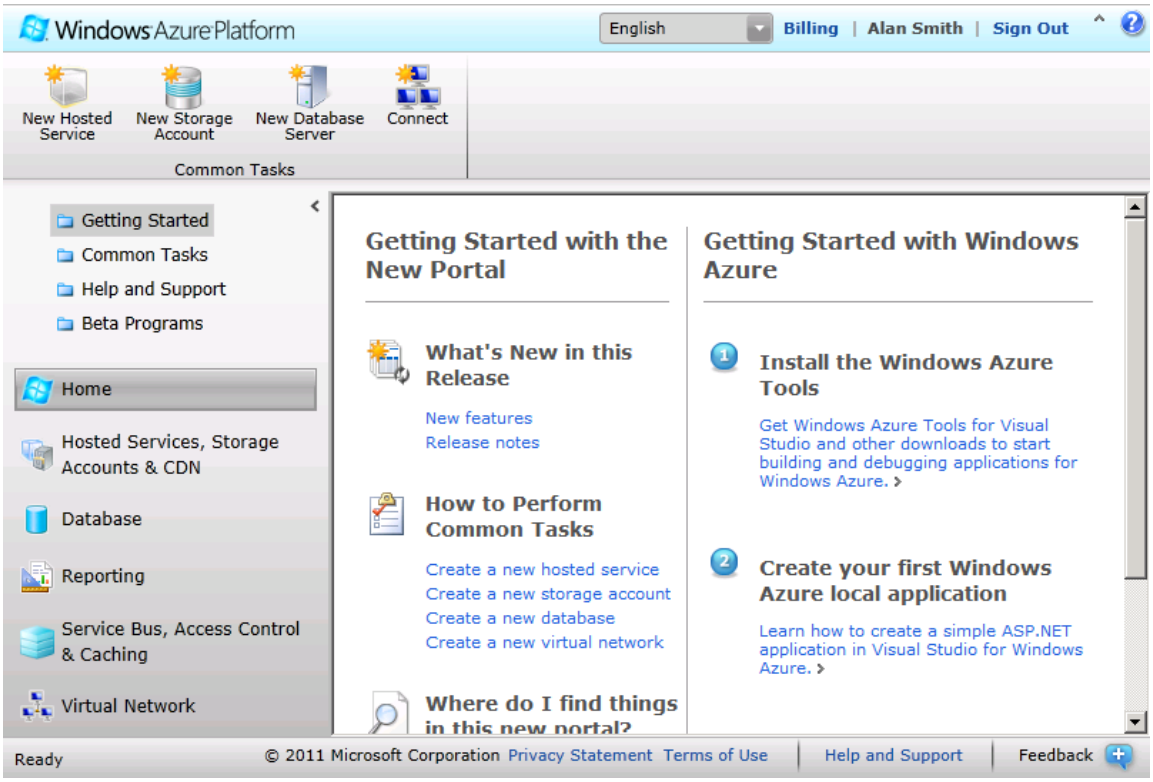
## Walkthrough: Simple Brokered Messaging

This walkthrough will use a basic example to show how to create a queue, send a message, and receive a message using the Azure AppFabric brokered messaging services using the management console and the direct programming model.

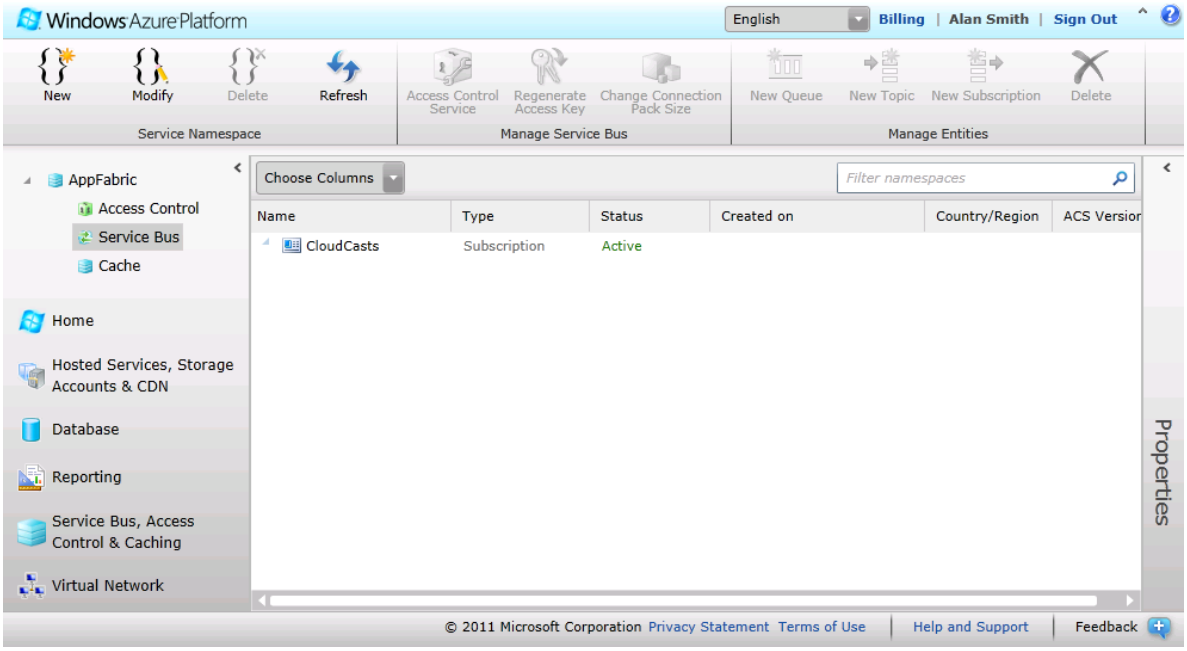
### Creating a Service Bus Namespace

The first step is to log onto the AppFabric management console and create a service bus namespace. If you have already done this you can skip this section.

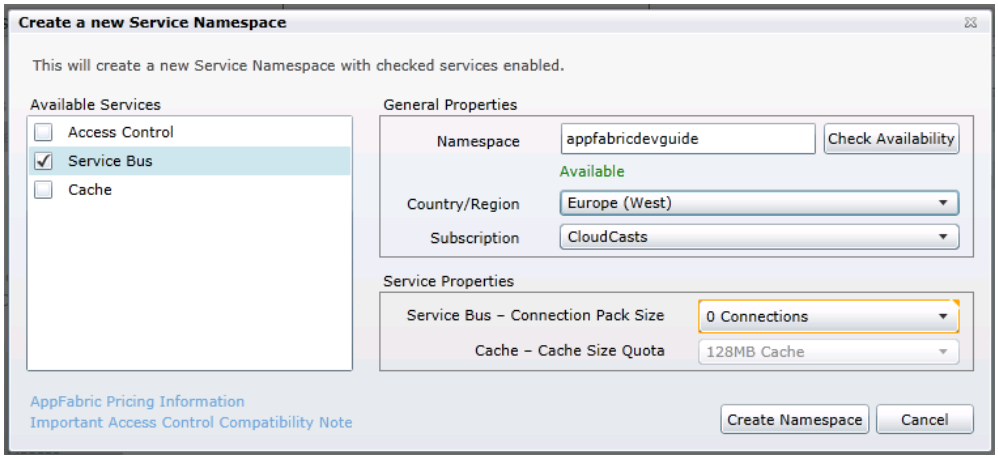
The Windows Azure Management console is located at https://windows.azure.com. Once a valid Live ID for a Windows Azure account has been provided the main management page will be displayed.



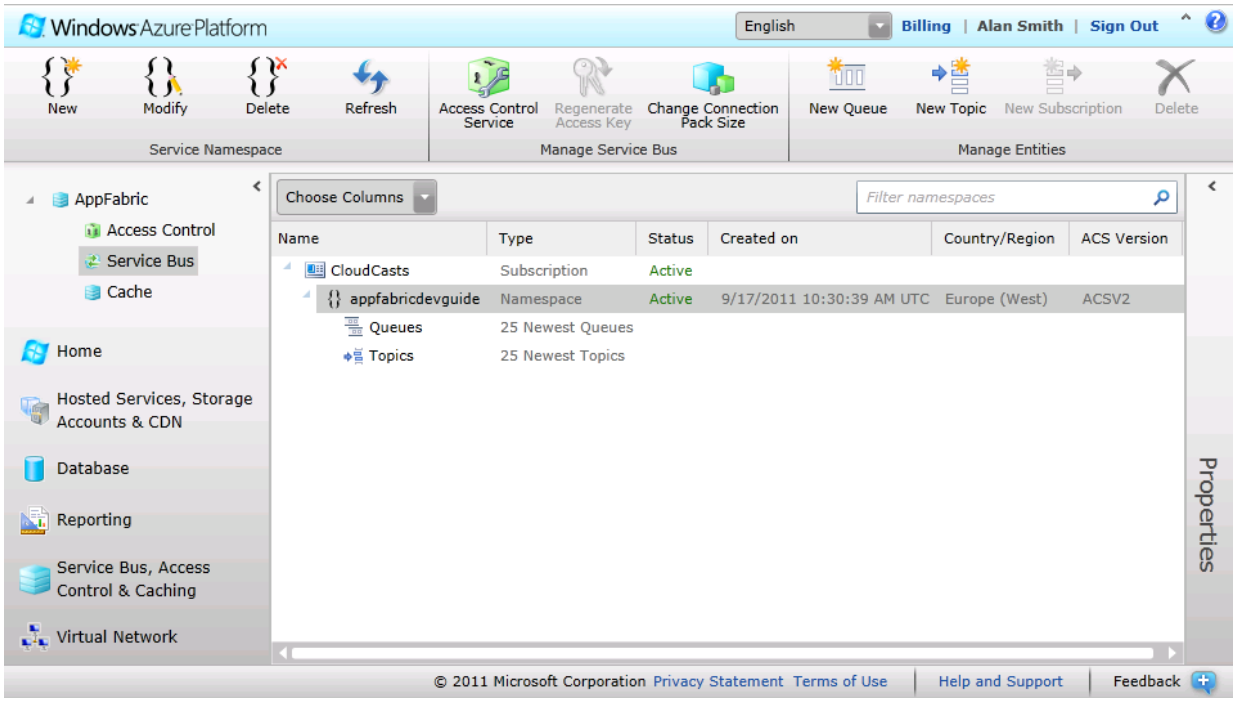

To get to the AppFabric management page, click the Service Bus, Access Control & Caching link. Clicking the Service Bus link will display the information relating to the service bus artifacts.




A namespace will need to be created before the service bus can be used. This namespace will be assigned to an Azure datacenter, but must be unique across all Windows Azure accounts. This is done by clicking on the New button in the Service Namespace section at the top left of the console.



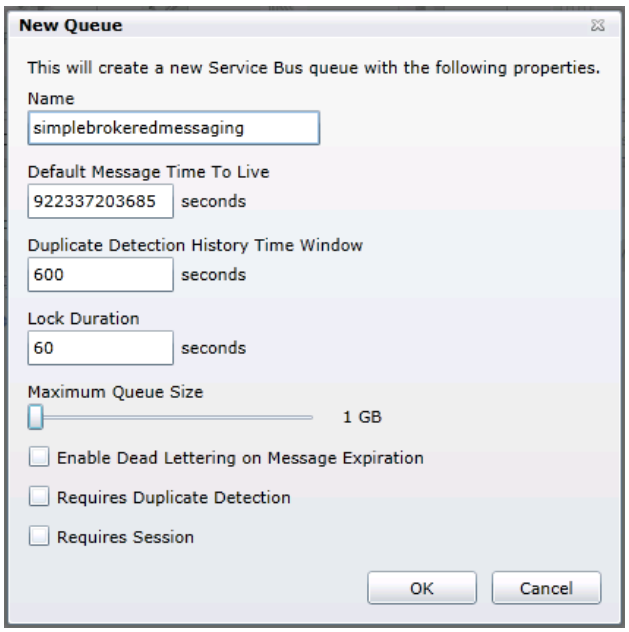

The name, data center location and any connection pack configuration is then entered; the name must be checked for availability to ensure that it is unique. Once this is done the namespace may take a few minutes to activate. Once activated, the namespace will be displayed in the management console.



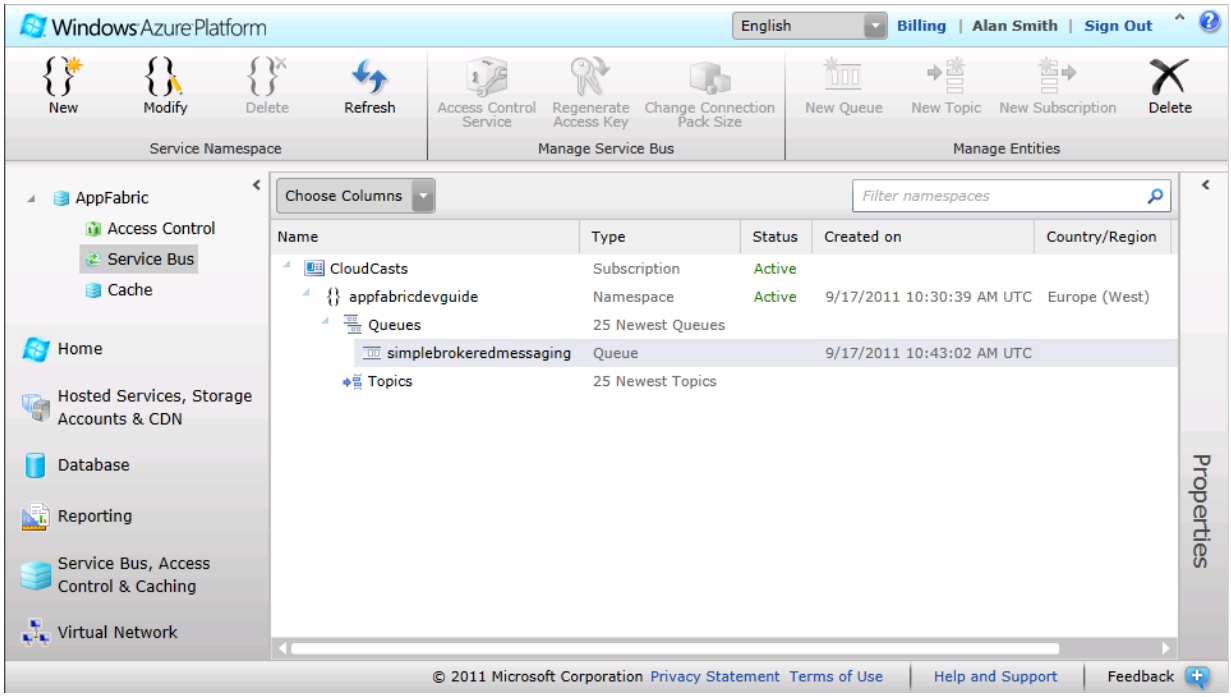

New service bus namespaces will be created to use Azure Access Control Service version 2 (ACSV2). Any namespaces created prior to the release of ACSV2 will use ACSV1. If you want to leverage the ACSV2 functionality you must create a new namespace.

**Creating a Queue**

Before any messages can be sent, a queue must be created in the service bus namespace. In this example the AppFabric management portal will be used to create the queue. Queue names must be unique within the service bus namespace. To create a queue, select the namespace and click the New Queue button.




In this example a queue named simplebrokeredmessaging is created, with the default settings. The queue is displayed in the management console as shown below.



**Creating the Project**

Now that the queue has been created, an application can be developed to use it. Create a Windows C# console application named SimpleBrokeredMessaging in Visual Studio 2010. Once this has been done the target framework should be set to .NET Framework 4.0 in the project properties, and references to the following assemblies added:

- Microsoft.ServiceBus
- System.Runtime.Serialization

**Setting the Account Details**

A class named AccountDetails can then be added to the project to store the service namespace, name and key of the account that will be used to access the service bus.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SimpleBrokeredMessaging
{
    class AccountDetails
    {
        public static string Namespace = "";
        public static string Name = "";
        public static string Key = "";
    }
}
```

The values for the static variables should be set to the appropriate values for the service bus account. These can be found in the default key property of the namespace in the AppFabric management portal. This class will be used in many of the code samples in the chapter to abstract the security details. Be aware that there is a security risk in placing credential information in code.

**Defining a Message Contract**

Messages can be created directly from serializable objects using the data contract serializer and this example will use a simple data contract to represent a pizza order. A class named PizzaOrder is added to the project and implemented as follows.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
```

```
namespace SimpleBrokeredMessaging
{
    [DataContract]
    public class PizzaOrder
    {
        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public string Pizza { get; set; }

        [DataMember]
        public int Quantity { get; set; }
    }
}
```

**Creating a Queue Client**

The next step is to create a queue client for the simplebrokeredmessaging queue. This involves the creation of a token provider with the appropriate credentials, a URI for the service bus, and a messaging factory. Once these have been created the messaging factory can be used to create a queue client.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;

namespace SimpleBrokeredMessaging
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Creating queue client...");

            // Create a token provider with the relevant credentials.
            TokenProvider credentials =
                TokenProvider.CreateSharedSecretTokenProvider
                (AccountDetails.Name, AccountDetails.Key);

            // Create a URI for the serivce bus.
            Uri serviceBusUri = ServiceBusEnvironment.CreateServiceUri
                ("sb", AccountDetails.Namespace, string.Empty);

            // Create a message factory for the service bus URI using the
            // credentials
            MessagingFactory factory = MessagingFactory.Create
                (serviceBusUri, credentials);

            // Create a queue client for the pizzaorders queue
```

```
            QueueClient queueClient =
                factory.CreateQueueClient("simplebrokeredmessaging");

            Console.WriteLine("Done!");
            Console.WriteLine();

        }
    }
}
```

**Creating and Send a Message**

The queue client can then be used to send a message to the queue. In order to do this an instance of the pizza order data contract will be created, which will then be used to create a brokered message. The brokered message is then sent to the queue by calling the send operation on the queue client.

```
// Create a queue client for the pizzaorders queue
QueueClient queueClient =
    factory.CreateQueueClient("simplebrokeredmessaging");

Console.WriteLine("Done!");
Console.WriteLine();

// Create a new pizza order.
PizzaOrder orderIn = new PizzaOrder()
{
    Name = "Alan",
    Pizza = "Hawaiian",
    Quantity = 1
};

// Create a brokered message based on the order.
BrokeredMessage orderInMsg = new BrokeredMessage(orderIn);

Console.WriteLine("Sending order for {0}...", orderIn.Name);

// Send the message to the queue.
queueClient.Send(orderInMsg);

Console.WriteLine("Done!");
Console.WriteLine();
```

**Receive a Message**

The next stage is to receive a message from the queue, this is done by calling the receive method on the queue client class. Sending and receiving messages is typically done in different applications, but to keep the implementation as simple as possible, the same application will be used.

```
// Send the message to the queue.
queueClient.Send(orderInMsg);

Console.WriteLine("Done!");
Console.WriteLine();

// Receive a message from the queue
Console.WriteLine("Receiving order...");
BrokeredMessage orderOutMsg = queueClient.Receive();

if (orderOutMsg != null)
{
    // Deserialize the message body to a pizza order.
    PizzaOrder orderOut = orderOutMsg.GetBody<PizzaOrder>();

    Console.WriteLine("Received order, {0} {1} for {2}",
        orderOut.Quantity, orderOut.Pizza, orderOut.Name);

    // Mark the order message as completed.
    orderOutMsg.Complete();
}
```

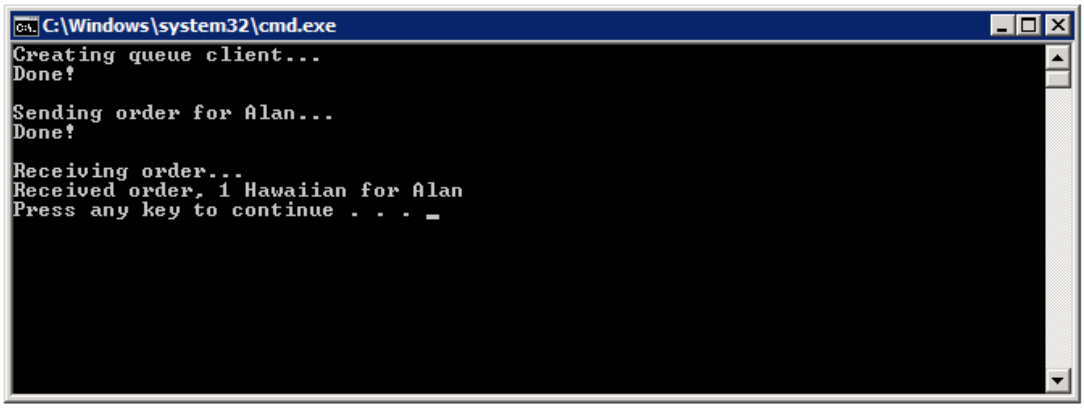
**Close Communication Objects**

As the communication objects used to connect to the AppFabric service bus will maintain open connections, and connections are a limited resource, these should be closed when no longer used. The following code will close the messaging factory, and all objects it created, so the queue client will also be closed.

```
if (orderOutMsg != null)
{
    // Deserialize the message body to a pizza order.
    PizzaOrder orderOut = orderOutMsg.GetBody<PizzaOrder>();

    Console.WriteLine("Received order, {0} {1} for {2}",
        orderOut.Quantity, orderOut.Pizza, orderOut.Name);

    // Mark the order message as completed.
    orderOutMsg.Complete();
}

// Close the message factory and everything it created.
factory.Close();
```

**Testing the Application**

The application can now be tested, if everything works well, the console should output the following.




**Summary**

This example showed a very simple way of creating interacting with an AppFabric queue to send and receive a message based on a data contract. Feel free to extend on this basic sample to explore the possibilities of AppFabric brokered messaging. You might like to try sending a number of messages to the queue, or using one application to send messages and another application to receive them.

### *Complete Code Listings*

The complete code listings for the walkthrough are shown below.

**AccountDetails.cs**

Account details have been removed.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SimpleBrokeredMessaging
{
    class AccountDetails
    {
        public static string Namespace = "";
        public static string Name = "";
        public static string Key = "";
    }
}
```

**PizzaOrder.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace SimpleBrokeredMessaging
{
    [DataContract]
    public class PizzaOrder
    {
        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public string Pizza { get; set; }

        [DataMember]
        public int Quantity { get; set; }
    }
}
```

**Program.cs**

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

```
using System.Linq;
using System.Text;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;

namespace SimpleBrokeredMessaging
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Creating queue client...");

            // Create a token provider with the relevant credentials.
            TokenProvider credentials =
                TokenProvider.CreateSharedSecretTokenProvider
                (AccountDetails.Name, AccountDetails.Key);

            // Create a URI for the serivce bus.
            Uri serviceBusUri = ServiceBusEnvironment.CreateServiceUri
                ("sb", AccountDetails.Namespace, string.Empty);

            // Create a message factory for the service bus URI using the
            // credentials
            MessagingFactory factory = MessagingFactory.Create
                (serviceBusUri, credentials);

            // Create a queue client for the pizzaorders queue
            QueueClient queueClient =
                factory.CreateQueueClient("simplebrokeredmessaging");

            Console.WriteLine("Done!");
            Console.WriteLine();

            // Create a new pizza order.
            PizzaOrder orderIn = new PizzaOrder()
            {
                Name = "Alan",
                Pizza = "Hawaiian",
                Quantity = 1
            };

            // Create a brokered message based on the order.
            BrokeredMessage orderInMsg = new BrokeredMessage(orderIn);

            Console.WriteLine("Sending order for {0}...", orderIn.Name);

            // Send the message to the queue.
            queueClient.Send(orderInMsg);

            Console.WriteLine("Done!");
            Console.WriteLine();

            // Receive a message from the queue
            Console.WriteLine("Receiving order...");
            BrokeredMessage orderOutMsg = queueClient.Receive();

            if (orderOutMsg != null)
```

```
            {
                // Deserialize the message body to a pizza order.
                PizzaOrder orderOut = orderOutMsg.GetBody<PizzaOrder>();

                Console.WriteLine("Received order, {0} {1} for {2}",
                    orderOut.Quantity, orderOut.Pizza, orderOut.Name);

                // Mark the order message as completed.
                orderOutMsg.Complete();
            }

            // Close the message factory and everything it created.
            factory.Close();

        }
    }
}
```

## Managing Messaging Artifacts

AppFabric queues topics and subscriptions are created in a service bus name space and accessed using the URI for the queue in that namespace. If a queue with the name orderqueue is created in the appfabricdevguide namespace in the AppFabric environment the URI for the queue will be as follows.

sb://appfabricdevguide.servicebus.windows.net/orderqueue

**Entity Immutability**

Once a queue, topic or subscription has been created in an AppFabric namespace it is immutable and cannot be modified. This means that it is not possible to change any of the properties on a messaging entity once created. For development and test purposes this does not present a problem as it's quick and easy to create and delete messaging entities.

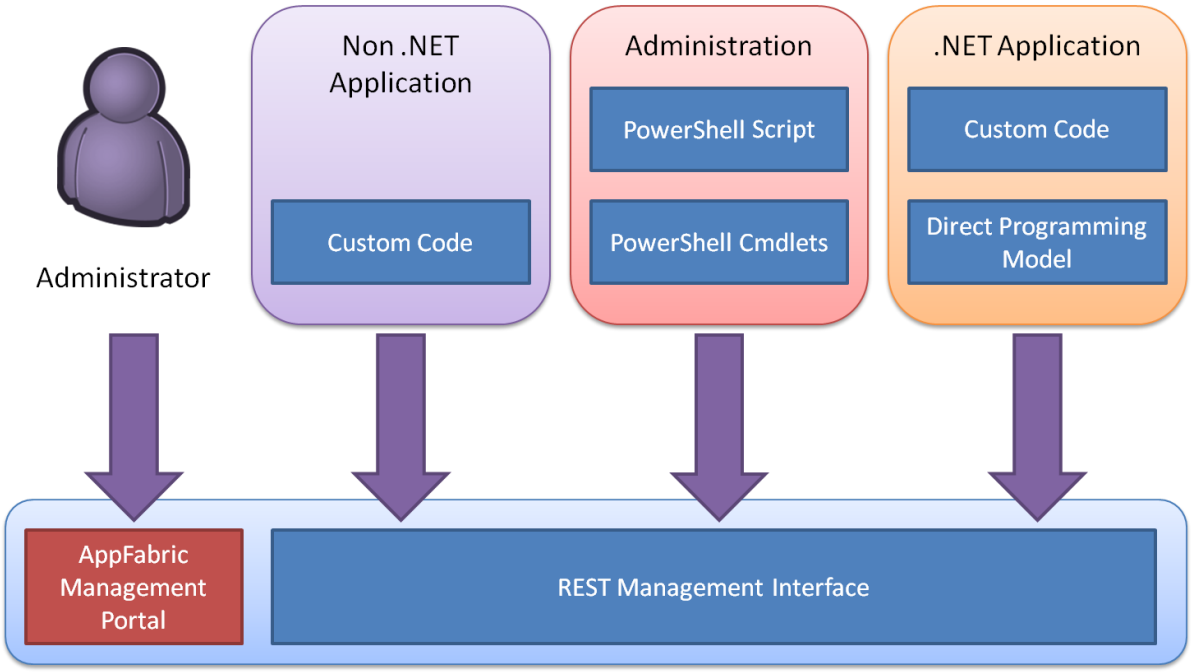
In production scenarios if the properties for a queue have not been set with appropriate properties and they need to be changed it will mean the queue will have to be deleted and recreated, or a new queue created and applications configured to communicate with it. Either of these scenarios could result in a break in communication between line-of-business applications.

When creating messaging artifacts is it wise to carefully consider the settings before they are used in production systems.

**Management Interfaces**

There are a number of options for managing AppFabric queues. .NET developers will typically opt for the direct programming model whilst administrators may want to use the sample PowerShell provider, for cross platform compatibility a REST model is provided. The management portal can also be used for basic management scenarios.

When using the AppFabric Application model the queue, topic and subscription services provide a provisioning mechanism that will create the artifacts in the service bus namespace when the application is deployed.




AppFabric Management Portal

The AppFabric management portal provides a quick and easy way to manage queues, topics and subscriptions in an AppFabric service bus namespace. The portal has some limitations in the number of artifacts that can be displayed in the interface, and can be time consuming to use when creating or deleting a large number of artifacts. This makes it most useful for managing small numbers of artifacts.

*Direct Programming Model*

The direct programming model provides a set of .NET classes that allow messaging artifacts to be managed from .NET applications. This allows .NET developers to manage artifacts programmatically and can be the best option to take when managing a large amount of entities or creating and deleting entities dynamically.

*REST Interface*

All management operations on the AppFabric service bus namespace take place through a REST interface. The direct programming model is built on this interface. The REST interface can also be used from non .NET applications.

*Sample PowerShell Provider*

The AppFabric samples contain a sample PowerShell provider that provides cmdlets for managing queues, topics, subscriptions and rules. The sample is located at:

[Samples Root]\ServiceBus\Scenarios\BrokeredMessaging\PowerShellProvider

*AppFabric Application Model*

The AppFabric application model that was released in the Azure AppFabric June CTP provides an option to provision queues, topics and subscriptions when an application is deployed to the AppFabric hosting environment. This allows developers to focus on the composition and creating of applications without having to spend time working with the provisioning or artifacts.

*Service Bus Explorer*

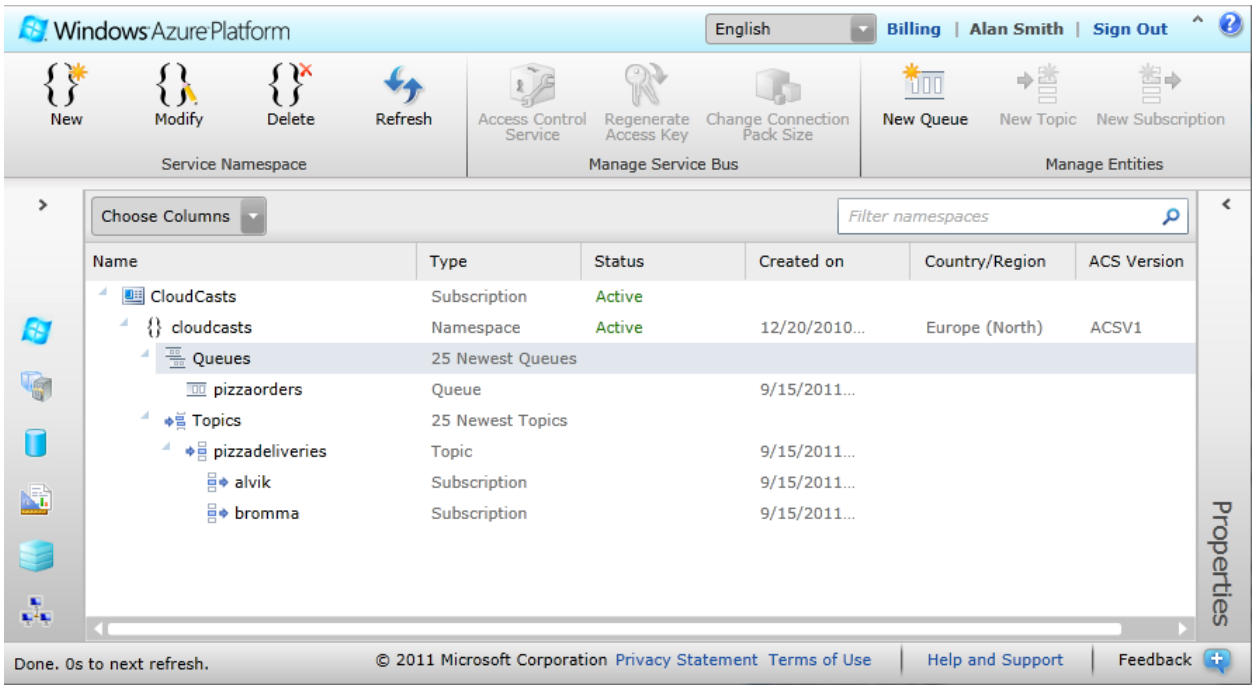
The Service Bus Explorer is an open source tool designed to manage queues, topics and subscriptions in the June CTP release of Azure AppFabric. As there have been significant changes made to the API between the CTP and the September 2011 the application will not work with this release. Hopefully the application will be updated for the latest release.

The link for the current Service Bus Explorer release at the time of writing is:

http://archive.msdn.microsoft.com/appfabriccat/Release/ProjectReleases.aspx?ReleaseId=5672

### *Using the AppFabric Management Portal*

The AppFabric management portal provides an interface for the creation and deleting of queues, topics and subscriptions. A screenshot of the interface is shown below.




For simple scenarios the AppFabric portal is a quick way to create the messaging entities required and remove them after use. The AppFabric Management Portal currently has the following limitations when displaying topics, queues and subscriptions.

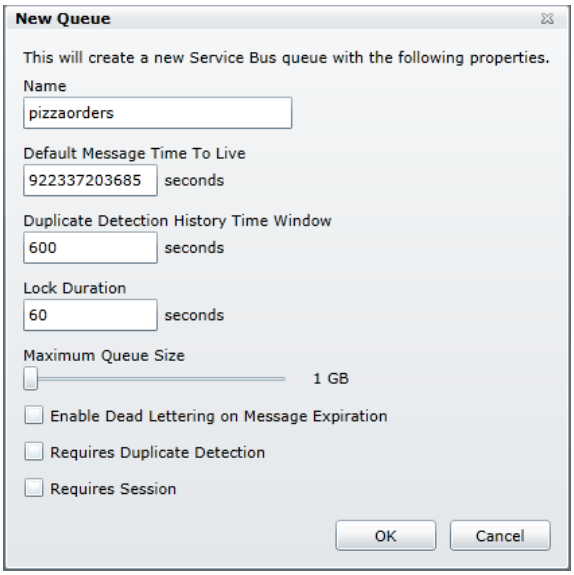
| | |
|---|---|
| Maximum number of queues per namespace | 25 |
| Maximum number of topics per namespace | 25 |
| Maximum number of subscriptions per topic | 5 |

There is also currently no way to view or set the MaxDeliveryCount property of queues and subscriptions using this interface.

When developing applications that use a number of queues, topics and subscriptions, of that require properties of these entities to be set to specific values a scripted or programmatic approach will be more effective. This is especially true when deleting a number of messaging entities.

**Creating a Queue**

Queues are created under a service bus namespace. When a namespace is selected the New Queue button becomes active. Clicking the New Queue button will show the following dialog box.

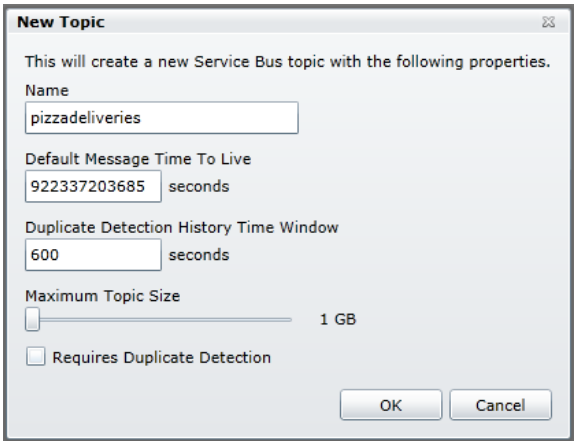



The name of the queue must be specified, and must be valid. The properties can either be left with the default values or assigned new values depending on the requirements of the system.

Descriptions of these properties will be provided later in the chapter.

**Creating a Topic**

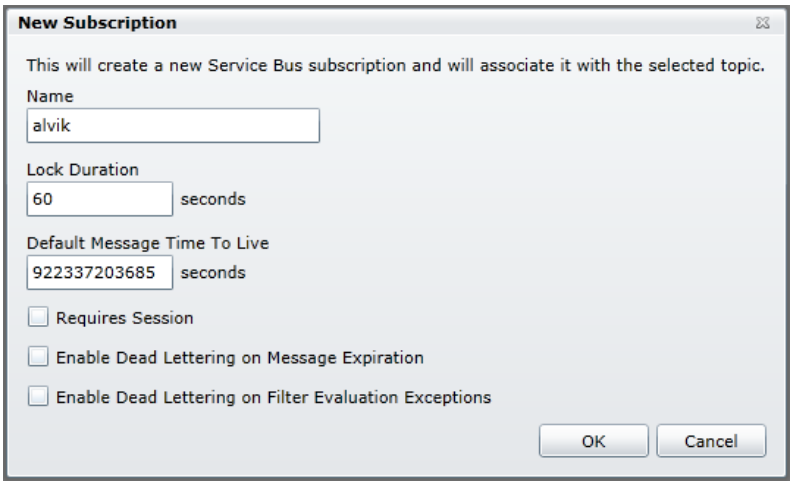
Topics are created under a service bus namespace. When a namespace is selected the New Topic button is enabled. Clicking the New Topic button will show the following dialog box.




The properties available on a topic are the ones applicable to the enqueueing of messages.

**Creating a Subscription**

Subscriptions are created under a topic. When a topic is selected the New Subscription button is enabled. Clicking the New Topic button will show the following dialog box.




The properties available on a subscription are the ones applicable to the storing and dequeuing of messages.

Although subscriptions can be created using the AppFabric management portal there is currently no option to specify a subscription filter or action. This means that any subscriptions created in the portal will subscribe to all messages that are enqueued to the topic. If subscriptions with filters are to be created, which is often the most common scenario; the direct programming model should be used.

### *NamespaceManager Class*

The NamespaceManager class provides management functionality for creating, listing, and deleting messaging artifacts. An instance of the class is initialized for a specific service bus namespace with credentials that allow service bus management operations to be performed.

**Constructors**

The NamespaceManager class provides four constructors for creating new instances of the class.

```
public NamespaceManager(string address, TokenProvider tokenProvider)

public NamespaceManager(Uri address, TokenProvider tokenProvider)

public NamespaceManager(string address, NamespaceManagerSettings settings)

public NamespaceManager(Uri address, NamespaceManagerSettings settings)
```

When choosing a constructor the main choice is weather to supply a TokenProvider or a NamespaceManagerSettings object. The NamespaceManagerSettings class is a simple class that provides a TokenProvider and an OperationTimeout as properties. Choosing the constructors with a NamespaceManagerSettings parameter allows the OperationTimeout to be changed from its default value of 60 seconds. This timeout value controls the timeout for message entity management functions.

**Useful Methods**

There are a number of methods available for creating and deleting messaging entities and for retrieving information about the entities present in the service bus namespace. The class also provides begin and end implementations for all these methods to allow for asynchronous invocation. Note that there are no methods to update messaging entities, as once they have been created their properties are immutable.

*CreateQueue, CreateTopic, CreateSubscription*

These methods create messaging entities in the service bus namespace represented by the NamespaceManager. Each method can either take a string as a parameter for the name of the entity, or a description object for the entity. Passing a name will result in default properties being set for the entity, passing a description object will allow the properties to be overridden. These methods all return a description object that contains the properties of the created entity.

*DeleteQueue, DeleteTopic, DeleteSubscription*

These methods will delete the specified queue, topic or subscription from the service bus namespace. Deleting a topic will also delete all subscriptions associates with that topic.

*GetQueue, GetTopic, GetSubscription*

These methods return the description class for a messaging entity. The entity is specified by name, and the description class contains the properties of the entity. It should be noted that these methods do not get the actual entity, just the entity properties, which are read-only, and cannot be modified.

*GetQueues, GetTopics, GetSubscriptions, GetRules*

The GetQueues and GetTopics methods return an enumerable collection of description objects for all the messaging entities of the relevant type in the service bus namespace. GetSubscriptions returns descriptions of all subscriptions in a specified topic. GetRules retunes a description of all the rules in a specified subscription.

*QueueExists, TopicExists, SubscriptionExists*

These methods return a boolean that specifies if a messaging entity with the specified name exists.

## Using NamespaceManager

This section will show examples of how the NamespaceManager class can be used to manage messaging entities in the AppFabric service bus. The creating of a NamespaceManager will be demonstrated, and then the creation, listing and deletion of queues, topics and subscriptions.

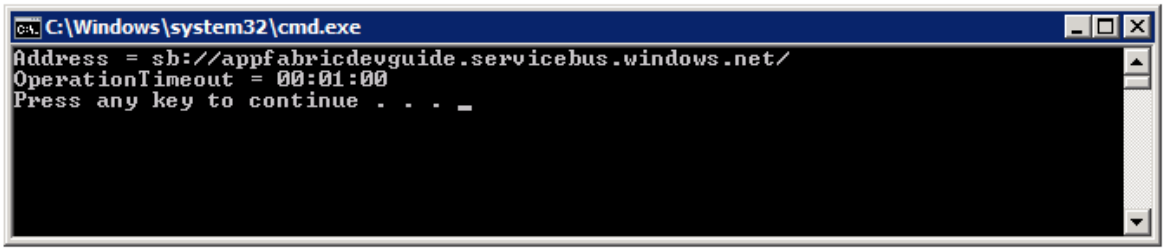
Note that the code used in these examples does not use asynchronous methods or exception handling, the intention is to focus purely on the  use of the API calls.

**Creating a NamespaceManager**

The following code will initialize a NamespaceManager for the specified service bus name space using the credentials and account key for the owner. The address and operational timeout of the client will then be displayed. Note that the namespace and credentials are stored in the AccountDetails class, and their values hidden.

```
// Create a token provider with the relevant credentials.
TokenProvider credentials =
    TokenProvider.CreateSharedSecretTokenProvider
    (AccountDetails.Name, AccountDetails.Key);

// Create a URI for the service bus.
Uri serviceBusUri = ServiceBusEnvironment.CreateServiceUri
    ("sb", AccountDetails.Namespace, string.Empty);


// Create a NamespaceManager for the specified namespace
// using the specified credentials.
NamespaceManager namespaceManager = new NamespaceManager(serviceBusUri, credentials);

Console.WriteLine("Address = {0}", namespaceManager.Address);
Console.WriteLine("OperationTimeout = {0}",
    namespaceManager.Settings.OperationTimeout);
```

The following output is generated when the code is executed.




It is recommended to use the ServiceBusEnvironment.CreateServiceUri class to generate the service bus URI as hard coding the value will require modifications if the service bus URI changes. This typically happens when moving applications between the appfabriclabs environment and production hosting environments.

The default timeout for operations against the service bus is one minute. In scenarios where the client will be used in a user interface it may be optimal to reduce this value to provide a quicker response to the client in the event of a failed operation. This can be done by setting the property directly or using one of the other constructors.

```
// Create a token provider with the relevant credentials.
TokenProvider credentials =
    TokenProvider.CreateSharedSecretTokenProvider
    (AccountDetails.Name, AccountDetails.Key);

// Create a URI for the service bus.
Uri serviceBusUri = ServiceBusEnvironment.CreateServiceUri
    ("sb", AccountDetails.Namespace, string.Empty);


NamespaceManagerSettings settings = new NamespaceManagerSettings()
{
    TokenProvider = credentials,
    OperationTimeout = TimeSpan.FromSeconds(10)
};

// Create a NamespaceManager for the specified namespace
// using the specified settings.
NamespaceManager namespaceManager = new NamespaceManager(serviceBusUri, settings);

Console.WriteLine("Address = {0}", namespaceManager.Address);
Console.WriteLine("OperationTimeout = {0}",
    namespaceManager.Settings.OperationTimeout);
```

In other scenarios where network connectivity may not be optimal it may be necessary to increase the operation timeout.

## Managing Queues

Queues, topics and subscriptions are managed programmatically by calling methods on the NamespaceManager class. As the techniques for managing queues are very similar to that of topics and subscriptions queue management will be covered in detail, then additional coverage for topics and subscriptions provided where appropriate.

### Creating Queues

The CreateQueue method of the NamespaceManager is overloaded and provides two options for creating queues.

```
public QueueDescription CreateQueue(string path)

public QueueDescription CreateQueue(QueueDescription description)
```

The first method will create a queue with the address of the NamespaceManager and path specified. The queue created will have the default property values. If the default property values are to be overridden, a QueueDescription can be created and the appropriate values set. This can then be used to create the queue using the second method.

Attempting to create queue with a path that already exists in the service bus namespace will result in a MessagingEntityAlreadyExistsException exception being thrown.

### QueueDescription Class

The QueueDescription class is a data contract class that provides properties which can be set when creating a queue. Setting any of these properties will override their default values in the new queue.

### Useful QueueDescription Properties

*DefaultMessageTimeToLive*

The DefaultMessageTimeToLive is a TimeSpan property that sets the default expiration timeout for messages that are added to the queue. The default value is TimeSpan.MaxValue, which is almost 30,000 years, meaning that messages will never expire during the expected lifetime of the application. If this value is set for a queue, any messages placed on that queue that have not been assigned a TimeToLive value will default to the value set for the queue. If the sending application has set the TimeToLive value for the message the value will not be changed.

*DuplicateDetectionHistoryTimeWindow*

If duplicate detection is enabled for a queue the DuplicateDetectionHistoryTimeWindow value will specify how long the MessageId values for the received messages will be retained in order to check for

duplicate messages. The property is a TimeSpan with a default value of 10 minutes and a maximum allowed value of 7 days.

*EnableDeadLetteringOnMessageExpiration*

By default any messages that have been on the queue for longer than the assigned time to live will expire and be removed from the queue. Setting EnableDeadLetteringOnMessageExpiration will cause expired messages to be moved to the dead-letter queue instead of being removed from the queue. This will allow the messages to be processed at a later point in time.

Be aware that expired messages on the dead letter queue will increase the size of the queue, which could result in the queue failing to receive new messages if expired messages are not de-queued from the dead letter queue.

*IsReadOnly*

When an instance of the QueueDescription class is created using the default constructor the properties can be modified. When a QueueDescription for a queue that is present in a service bus namespace is returned the properties can't be modified. The IsReadOnly property indicates weather the queue description properties can be modified.

*LockDuration*

LockDuration is a TimeSpan property that determines the maximum time a message will remain in the locked state when received using the peek-lock receive mode. The default value is 30 seconds; the maximum allowed value is 5 minutes. If the receiver fails to complete, abandon, defer or dead-letter the message within this time the message will become visible on the queue for other receivers to receive.

If it is expected that the receiving application may take longer than 30 seconds to determine if a message can be successfully processed or not the LockDuration of the queue should be set to an appropriate value. Be aware that it is no possible for the receiving application to specify the lock duration when dequeuing a message, and once a queue is created the lock duration cannot be changed. In production systems care should be taken to ensure a suitable lock duration has been set.

*MaxDeliveryCount*

The MaxDeliveryCount property specifies the maximum number of times a message can be received by a receiving application before the message is automatically place on the dead letter queue. This is used to prevent the repeated processing of "poison messages" causing bottle necks in a message channel.

The default setting is 10, the minimum value allowed value 1, the maximum allowed value is int.MaxValue, (2,147,483,647). Setting the maximum value effectively turns off the dead lettering of poison messages.

*MaxSizeInMegabytes*

The MaxSizeInMegabytes property defines the maximum size a queue can reach before new messages are rejected from the queue. The default setting is 1,024, which is 1 GB, and the current maximum allowable value is 5120, which is 5 GB. The maximum size of a queue in GB is used as a multiplier when calculating the billing for entity hours in the service bus, a queue with a maximum size of 4 GB will be billed at 4 times the price of one with a maximum size of 1 GB.

*MessageCount*

The MessageCount property returns a count of the number of messages on the queue.

*RequiresDuplicateDetection*

*When set to true, the RequiresDuplicateDetection property will ensure that all enqueued messages with a duplicate value of the MessageId property within the DuplicateDetectionHistoryTimeWindow will be ignored.*

*RequiresSession*

The RequiresSession property specifies that messages sent to the queue most have a value set for their SessionId property. Sessions are used for correlating related messages into transactions.

*SizeInBytes*

The SizeInBytes returns the current size of the queue.

**Examples**

The following code shows how to create two queues using the two overloaded methods.

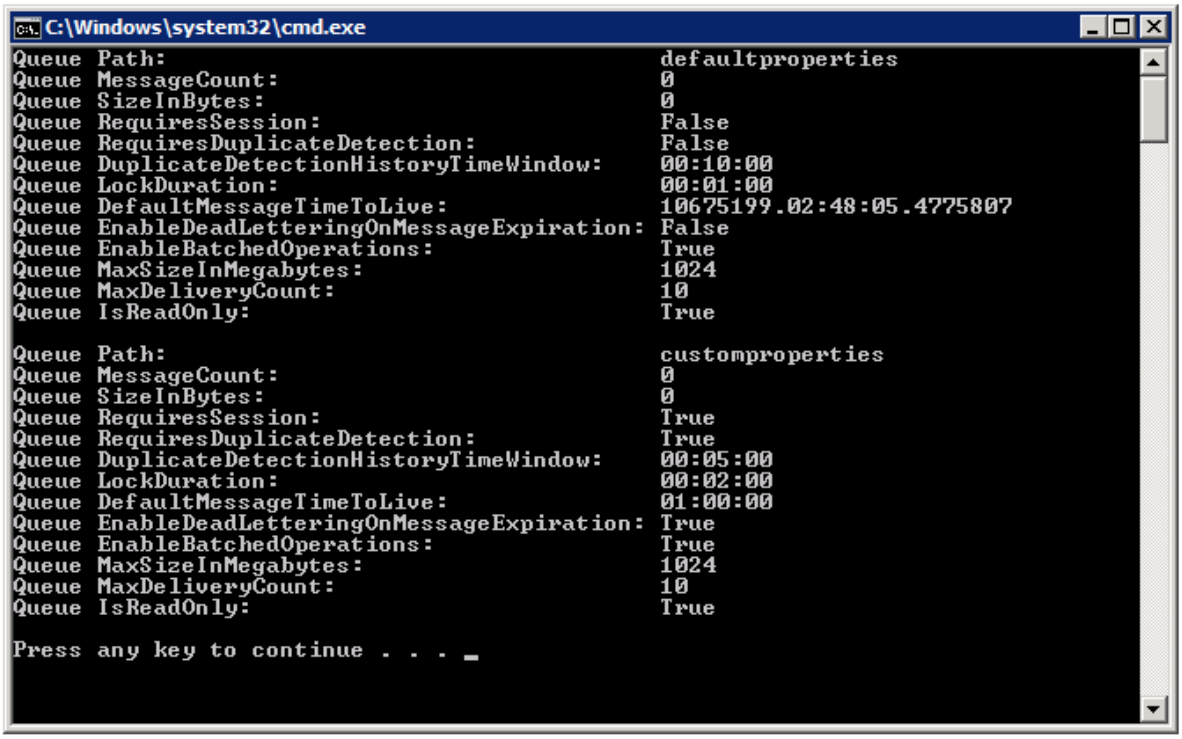
```
// Create a token provider with the relevant credentials.
TokenProvider credentials =
    TokenProvider.CreateSharedSecretTokenProvider
    (AccountDetails.Name, AccountDetails.Key);

// Create a URI for the serivce bus.
Uri serviceBusUri = ServiceBusEnvironment.CreateServiceUri
    ("sb", AccountDetails.Namespace, string.Empty);

// Create a ServiceBusNamespaceClient for the specified namespace
// using the specified credentials.
NamespaceManager namespaceManager = new NamespaceManager(serviceBusUri, credentials);

// Create a queue with the default properties.
QueueDescription defaultQueueDescription =
namespaceManager.CreateQueue("defaultproperties");
WriteQueueDescription(defaultQueueDescription);

// Create a custom queue description.
QueueDescription customDescription = new QueueDescription("customproperties")
{
    DefaultMessageTimeToLive = TimeSpan.FromHours(1),
```

```
    EnableDeadLetteringOnMessageExpiration = true,
    RequiresDuplicateDetection = true,
    DuplicateDetectionHistoryTimeWindow = TimeSpan.FromMinutes(5),
    LockDuration = TimeSpan.FromMinutes(2),
    RequiresSession = true
};

// Create a queue using the queue description.
QueueDescription customQueueDescription =
namespaceManager.CreateQueue(customDescription);
WriteQueueDescription(customQueueDescription);
```

The WriteQueueDescription method is implemented as follows.

```
static void WriteQueueDescription(QueueDescription queueDescription)
{
    Console.WriteLine("Queue Path:                                    {0}",
        queueDescription.Path);
    Console.WriteLine("Queue MessageCount:                            {0}",
        queueDescription.MessageCount);
    Console.WriteLine("Queue SizeInBytes:                             {0}",
        queueDescription.SizeInBytes);
    Console.WriteLine("Queue RequiresSession:                         {0}",
        queueDescription.RequiresSession);
    Console.WriteLine("Queue RequiresDuplicateDetection:              {0}",
        queueDescription.RequiresDuplicateDetection);
    Console.WriteLine("Queue DuplicateDetectionHistoryTimeWindow:     {0}",
        queueDescription.DuplicateDetectionHistoryTimeWindow);
    Console.WriteLine("Queue LockDuration:                            {0}",
        queueDescription.LockDuration);
    Console.WriteLine("Queue DefaultMessageTimeToLive:                {0}",
        queueDescription.DefaultMessageTimeToLive);
    Console.WriteLine("Queue EnableDeadLetteringOnMessageExpiration: {0}",
        queueDescription.EnableDeadLetteringOnMessageExpiration);
    Console.WriteLine("Queue EnableBatchedOperations:                 {0}",
        queueDescription.EnableBatchedOperations);
    Console.WriteLine("Queue MaxSizeInMegabytes:                      {0}",
        queueDescription.MaxSizeInMegabytes);
    Console.WriteLine("Queue MaxDeliveryCount:                        {0}",
        queueDescription.MaxDeliveryCount);
    Console.WriteLine("Queue IsReadOnly:                              {0}",
        queueDescription.IsReadOnly);
    Console.WriteLine();
}
```

When the code is run the following output is displayed.




**Listing Queues**

The following code will retrieve a list of queues from a service bus namespace and output the queue details.

```
IEnumerable<QueueDescription> queueDescriptions = namespaceManager.GetQueues();

foreach (QueueDescription queueDescription in queueDescriptions)
{
    WriteQueueDescription(queueDescription);
}
```

**Deleting Queues**

Queues can be deleted using the DeleteQueue method of NamespaceManager, specifying the path of the queue. Attempting to delete a queue that does not exist will result in a MessagingEntityNotFoundException being thrown.

The following code shows how to delete all the queues in a service bus namespace.

```
IEnumerable<QueueDescription> queueDescriptions = namespaceManager.GetQueues();

foreach (QueueDescription queueDescription in queueDescriptions)
{
    Console.Write("Deleting {0}...", queueDescription.Path);
    namespaceManager.DeleteQueue(queueDescription.Path);
    Console.WriteLine("Done!");
}
```

As messaging entities are billed on an hourly basis it would be wise for developers to have a quick an easy way or removing a number of queues and topics form a namespace when a development or testing session is complete.

## Managing Topics and Subscriptions

The techniques for managing topics in an AppFabric service bus namespace are very similar to those used for queues.

### Creating Topics

Topics are created in a similar way to queues; the two overloaded CreateTopic methods are shown below.

```
public TopicDescription CreateTopic(string path)

public TopicDescription CreateTopic(TopicDescription description)
```

The TopicDescription class provides the properties that can be set when creating a topic. These properties are used in the same way as the properties relevant for enqueueing messages in the QueueDescription class.

### Creating Subscriptions

Subscriptions are created inside an existing topic. The NamespaceManager class provides an overloaded CreateSubscription method.

```
public SubscriptionDescription CreateSubscription(string topicPath, string name)

public SubscriptionDescription CreateSubscription(SubscriptionDescription description)

public SubscriptionDescription CreateSubscription
    (string topicPath, string name, Filter filter)
```

```
public SubscriptionDescription CreateSubscription
    (SubscriptionDescription description, Filter filter)

public SubscriptionDescription CreateSubscription
    (string topicPath, string name, RuleDescription ruleDescription)

public SubscriptionDescription CreateSubscription
    (SubscriptionDescription description, RuleDescription ruleDescription)
```

If a subscription is created with no specified filter it will subscribe to all messages.

```
// Create a NamespaceManager
NamespaceManager namespaceMgr = new NamespaceManager(serviceBusUri, credentials);

// Create a subscription on the orders topic that subscribes to all orders
namespaceMgr.CreateSubscription("ordertopic", "allorders");
```

If a filter is specified the subscriptions will subscribe to messages where a property in the message properties dictionary matches the filter. In this example a property named “region” will be added to the message properties dictionary and assigned the value of the country region code of the order. The order messages will be routed to the topic for the appropriate region.

```
// Create a NamespaceManager
NamespaceManager namespaceMgr = new NamespaceManager(serviceBusUri, credentials);

// Create subscriptions on the order topic for the sales regions.
namespaceMgr.CreateSubscription
    ("ordertopic", "usorders", new SqlFilter("region = 'US'"));

namespaceMgr.CreateSubscription
    ("ordertopic", "canadaorders", new SqlFilter("region = 'CA'"));

namespaceMgr.CreateSubscription
    ("ordertopic", "ukorders", new SqlFilter("region = 'GB'"));

namespaceMgr.CreateSubscription
    ("ordertopic", "australiaorders", new SqlFilter("region = 'AU'"));

namespaceMgr.CreateSubscription
    ("ordertopic", "germanyorders", new SqlFilter("region = 'DE'"));

namespaceMgr.CreateSubscription
    ("ordertopic", "franceorders", new SqlFilter("region = 'FR'"));
```

## Queue, Topic and Subscription Properties Summarized

The following table provides a summary of the properties available on queues, topics and subscriptions. In general enqueue related properties are available on queues and topics, and dequeue related properties on queues and subscriptions.

| Property | Default | Max Value | Applicable to |
|---|---|---|---|
| DefaultMessageTimeToLive | TimeSpan.MaxValue | TimeSpan.MaxValue | Queue, Topic, Subscription |
| DuplicateDetectionHistoryTimeWindow | 00:10:00 | 7 days | Queue, Topic |
| EnableBatchedOperations | True | | Queue, Topic, Subscription |
| EnableDeadLetteringOnFilterEvaluationExceptions | False | | Subscription |
| EnableDeadLetteringOnMessageExpiration | False | | Queue, Subscription |
| ExtensionData | | | Queue, Topic, Subscription |
| IsReadOnly | False | | Queue, Topic, Subscription |
| LockDuration | 00:01:00 | 00:05:00 | Queue, Subscription |
| MaxDeliveryCount | 10 | Int.MaxValue | Queue, Subscription |
| MaxSizeInMegabytes | 1024 | 5120 | Queue, Topic |
| MessageCount | | | Queue, Subscription |
| Name | | 50 characters | Subscription |
| Path | | 260 characters | Queue, Topic |
| RequiresDuplicateDetection | False | | Queue, Topic |

| | | | |
|---|---|---|---|
| RequiresSession | False | | Queue, Subscription |
| SizeInBytes | | | Queue, Topic |
| TopicPath | | 260 characters | Subscription |

## *Walkthrough: Creating a Simple Queue Management Tool*

When creating applications that use AppFabric messaging it makes sense to have a separate application or console to manage the messaging artifacts rather than creating them in the applications that send and receive messages.

This section will take a walk through the creation of a simple console application to manage queues in an AppFabric service bus namespace. This application is a help when creating basic messaging applications but it does not provide support for setting properties on queues, or managing topics and subscriptions.

### Creating the Solution

The solution is created as a C# console application named SimpleQueueManagementConsole. The target framework of the project is changed from .NET Framework 4 Client Profile to .NET Framework 4 and references to the following assemblies added.

- Microsoft.ServiceBus
- System.Configuration
- System.Runtime.Serialization

### Adding Configuration Settings

The management console must connect to a service bus namespace using a set of credentials. To prompting the user for credentials they will be placed in the app.config file.

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>

  <appSettings>
    <add key="NameSpace" value="(enter value)"/>
    <add key="CredentialName" value="(enter value)"/>
    <add key="CredentialKey" value="(enter value)"/>
  </appSettings>

</configuration>
```

You will have to set the appropriate values for your AppFabric service bus account. Be aware that storing these credentials in a configuration file is a potential security risk.

**Adding a ManagementHelper Class**

The next step is to add a helper class named ManagementHelper that will initialize a NamespaceManager member variable with the appropriate settings in the constructor.

```
namespace SimpleQueueManagementConsole
{
    class ManagementHelper
    {
        private NamespaceManager m_NamespaceManager;

        public ManagementHelper()
        {
            // Retrieve the account details from the configuration file.
            string nameSpace = ConfigurationManager.AppSettings["NameSpace"];
            string credentialName = ConfigurationManager.AppSettings["CredentialName"];
            string credentialKey = ConfigurationManager.AppSettings["CredentialKey"];


            // Create a token provider with the relevant credentials.
            TokenProvider credentials =
                TokenProvider.CreateSharedSecretTokenProvider
                (credentialName, credentialKey);


            // create a Uri for the service bus using the specified namespace
            Uri sbUri = ServiceBusEnvironment.CreateServiceUri
                ("sb", nameSpace, string.Empty);

            // Create a ServiceBusNamespaceClient for the specified namespace
            // using the specified credentials.
            m_NamespaceManager = new NamespaceManager (sbUri, credentials);

            // Set the timeout to 15 seconds.
            m_NamespaceManager.Settings.OperationTimeout = TimeSpan.FromSeconds(15);

            // Output the client address.
            Console.WriteLine("Service bus address {0}", m_NamespaceManager.Address);
        }
```

Once this is done methods can be added to create, delete, get and list queues.

```
public void CreateQueue(string queuePath)
{
    Console.Write("Creating queue {0}...", queuePath);
    m_NamespaceManager.CreateQueue(queuePath);
    Console.WriteLine("Done!");
}
```

```
public void DeleteQueue(string queuePath)
{
    Console.Write("Deleting queue {0}...", queuePath);
    m_NamespaceManager.DeleteQueue(queuePath);
    Console.WriteLine("Done!");
}

public void GetQueue(string queuePath)
{
    QueueDescription queueDescription = m_NamespaceManager.GetQueue(queuePath);

    Console.WriteLine("Queue Path:                                      {0}",
        queueDescription.Path);
    Console.WriteLine("Queue MessageCount:                              {0}",
        queueDescription.MessageCount);
    Console.WriteLine("Queue SizeInBytes:                               {0}",
        queueDescription.SizeInBytes);
    Console.WriteLine("Queue RequiresSession:                           {0}",
        queueDescription.RequiresSession);
    Console.WriteLine("Queue RequiresDuplicateDetection:                {0}",
        queueDescription.RequiresDuplicateDetection);
    Console.WriteLine("Queue DuplicateDetectionHistoryTimeWindow:       {0}",
        queueDescription.DuplicateDetectionHistoryTimeWindow);
    Console.WriteLine("Queue LockDuration:                              {0}",
        queueDescription.LockDuration);
    Console.WriteLine("Queue DefaultMessageTimeToLive:                  {0}",
        queueDescription.DefaultMessageTimeToLive);
    Console.WriteLine("Queue EnableDeadLetteringOnMessageExpiration: {0}",
        queueDescription.EnableDeadLetteringOnMessageExpiration);
    Console.WriteLine("Queue EnableBatchedOperations:                   {0}",
        queueDescription.EnableBatchedOperations);
    Console.WriteLine("Queue MaxSizeInMegabytes:                        {0}",
        queueDescription.MaxSizeInMegabytes);
    Console.WriteLine("Queue MaxDeliveryCount:                          {0}",
        queueDescription.MaxDeliveryCount);
    Console.WriteLine("Queue IsReadOnly:                                {0}",
        queueDescription.IsReadOnly);
}

public void ListQueues()
{
    IEnumerable<QueueDescription> queueDescriptions = m_NamespaceManager.GetQueues();
    Console.WriteLine("Lisitng queues...");
    foreach (QueueDescription queueDescription in queueDescriptions)
    {
        Console.WriteLine("\t{0}", queueDescription.Path);
    }
    Console.WriteLine("Done!");
}
```

**Processing the Commands**

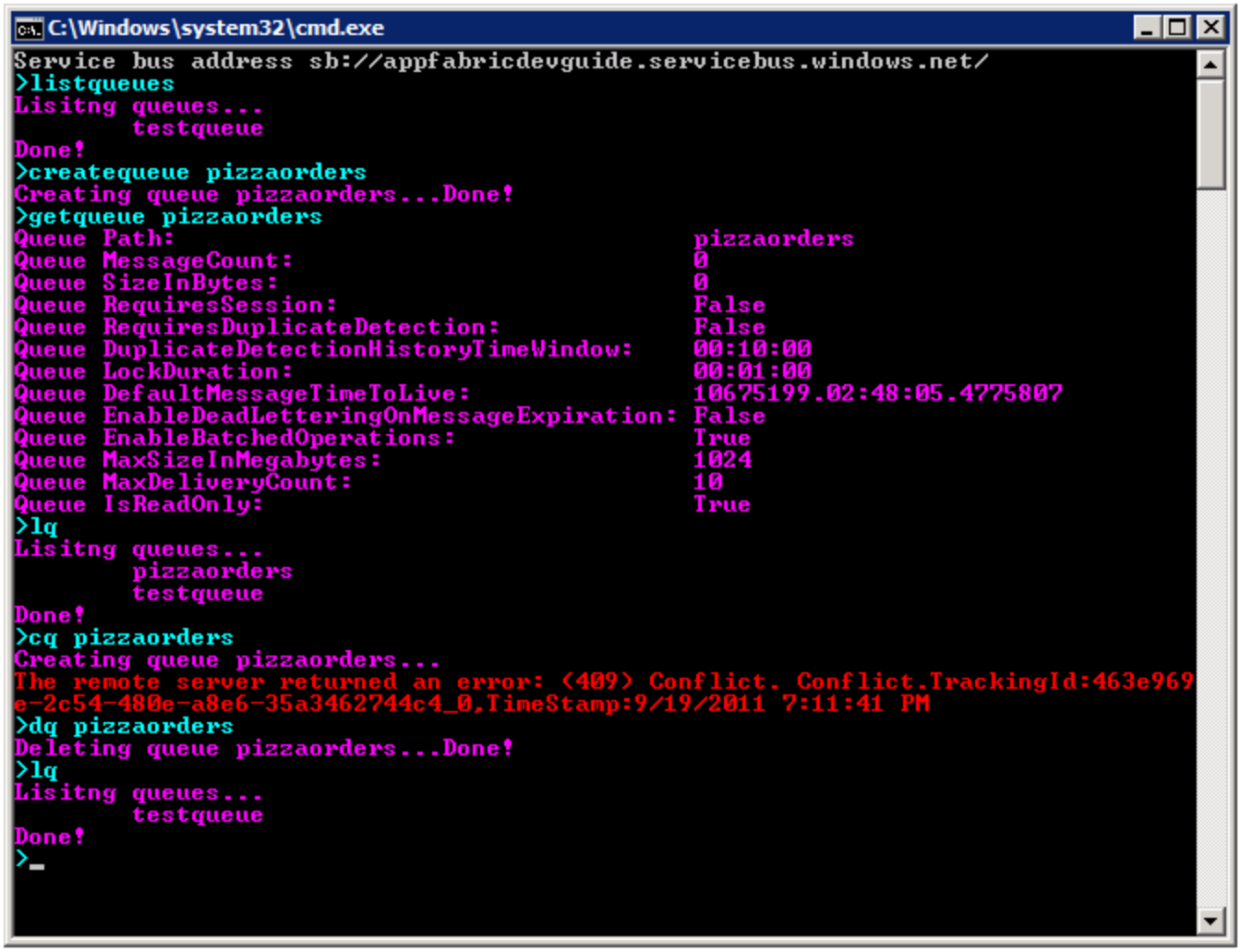
The console application will process the commands that the user enters and call the appropriate methods on the helper class. The following code in the Main method will implement this.

```
static void Main(string[] args)
{
    ManagementHelper helper = new ManagementHelper();

    bool done = false;
    do
    {
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.Write(">");
        string commandLine = Console.ReadLine();
        Console.ForegroundColor = ConsoleColor.Magenta;
        string[] commands = commandLine.Split(' ');

        try
        {
            if (commands.Length > 0)
            {
                switch (commands[0])
                {
                    case "createqueue":
                    case "cq":
                        if (commands.Length > 1)
                        {
                            helper.CreateQueue(commands[1]);
                        }
                        else
                        {
                            Console.ForegroundColor = ConsoleColor.Yellow;
                            Console.WriteLine("Queue path not specified.");
                        }
                        break;
                    case "listqueues":
                    case "lq":
                        helper.ListQueues();
                        break;
                    case "getqueue":
                    case "gq":
                        if (commands.Length > 1)
                        {
                            helper.GetQueue(commands[1]);
                        }
                        else
                        {
                            Console.ForegroundColor = ConsoleColor.Yellow;
                            Console.WriteLine("Queue path not specified.");
                        }
                        break;
                    case "deletequeue":
                    case "dq":
                        if (commands.Length > 1)
                        {
                            helper.DeleteQueue(commands[1]);
                        }
```

```
                        else
                        {
                            Console.ForegroundColor = ConsoleColor.Yellow;
                            Console.WriteLine("Queue path not specified.");
                        }
                        break;
                    case "exit":
                        done = true;
                        break;
                    default:
                        break;
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine();
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(ex.Message);
        }

    } while (!done);

}
```

**Testing the Console**

The queue management console can now be tested.




## Complete Code Listings

The complete listings for the two classes are provided below.

**app.config**

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>

  <appSettings>
    <add key="NameSpace" value=""/>
    <add key="CredentialName" value=""/>
    <add key="CredentialKey" value=""/>
  </appSettings>

</configuration>
```

**ManagementHelper.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Configuration;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;

namespace SimpleQueueManagementConsole
{
    class ManagementHelper
    {
        private NamespaceManager m_NamespaceManager;

        public ManagementHelper()
        {
            // Retrieve the account details from the configuration file.
            string nameSpace = ConfigurationManager.AppSettings["NameSpace"];
            string credentialName = ConfigurationManager.AppSettings["CredentialName"];
            string credentialKey = ConfigurationManager.AppSettings["CredentialKey"];


            // Create a token provider with the relevant credentials.
            TokenProvider credentials =
                TokenProvider.CreateSharedSecretTokenProvider
                (credentialName, credentialKey);


            // create a Uri for the service bus using the specified namespace
            Uri sbUri = ServiceBusEnvironment.CreateServiceUri
                ("sb", nameSpace, string.Empty);

            // Create a ServiceBusNamespaceClient for the specified namespace
            // using the specified credentials.
            m_NamespaceManager = new NamespaceManager(sbUri, credentials);

            // Set the timeout to 15 seconds.
            m_NamespaceManager.Settings.OperationTimeout = TimeSpan.FromSeconds(15);

            // Output the client address.
            Console.WriteLine("Service bus address {0}", m_NamespaceManager.Address);
        }



        public void CreateQueue(string queuePath)
        {
            Console.Write("Creating queue {0}...", queuePath);
            m_NamespaceManager.CreateQueue(queuePath);
            Console.WriteLine("Done!");
        }

        public void DeleteQueue(string queuePath)
        {
```

```
            Console.Write("Deleting queue {0}...", queuePath);
            m_NamespaceManager.DeleteQueue(queuePath);
            Console.WriteLine("Done!");
        }

        public void GetQueue(string queuePath)
        {
            QueueDescription queueDescription = m_NamespaceManager.GetQueue(queuePath);

            Console.WriteLine("Queue Path:                                  {0}",
                queueDescription.Path);
            Console.WriteLine("Queue MessageCount:                          {0}",
                queueDescription.MessageCount);
            Console.WriteLine("Queue SizeInBytes:                           {0}",
                queueDescription.SizeInBytes);
            Console.WriteLine("Queue RequiresSession:                       {0}",
                queueDescription.RequiresSession);
            Console.WriteLine("Queue RequiresDuplicateDetection:            {0}",
                queueDescription.RequiresDuplicateDetection);
            Console.WriteLine("Queue DuplicateDetectionHistoryTimeWindow:   {0}",
                queueDescription.DuplicateDetectionHistoryTimeWindow);
            Console.WriteLine("Queue LockDuration:                          {0}",
                queueDescription.LockDuration);
            Console.WriteLine("Queue DefaultMessageTimeToLive:              {0}",
                queueDescription.DefaultMessageTimeToLive);
            Console.WriteLine("Queue EnableDeadLetteringOnMessageExpiration: {0}",
                queueDescription.EnableDeadLetteringOnMessageExpiration);
            Console.WriteLine("Queue EnableBatchedOperations:               {0}",
                queueDescription.EnableBatchedOperations);
            Console.WriteLine("Queue MaxSizeInMegabytes:                    {0}",
                queueDescription.MaxSizeInMegabytes);
            Console.WriteLine("Queue MaxDeliveryCount:                      {0}",
                queueDescription.MaxDeliveryCount);
            Console.WriteLine("Queue IsReadOnly:                            {0}",
                queueDescription.IsReadOnly);
        }

        public void ListQueues()
        {
            IEnumerable<QueueDescription> queueDescriptions =
m_NamespaceManager.GetQueues();
            Console.WriteLine("Lisitng queues...");
            foreach (QueueDescription queueDescription in queueDescriptions)
            {
                Console.WriteLine("\t{0}", queueDescription.Path);
            }
            Console.WriteLine("Done!");
        }




    }

}
```

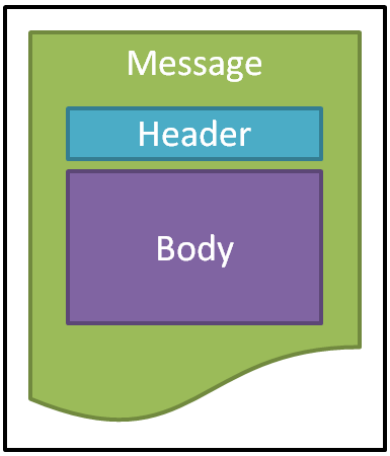
**Program.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SimpleQueueManagementConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            ManagementHelper helper = new ManagementHelper();

            bool done = false;
            do
            {
                Console.ForegroundColor = ConsoleColor.Cyan;
                Console.Write(">");
                string commandLine = Console.ReadLine();
                Console.ForegroundColor = ConsoleColor.Magenta;
                string[] commands = commandLine.Split(' ');

                try
                {
                    if (commands.Length > 0)
                    {
                        switch (commands[0])
                        {
                            case "createqueue":
                            case "cq":
                                if (commands.Length > 1)
                                {
                                    helper.CreateQueue(commands[1]);
                                }
                                else
                                {
                                    Console.ForegroundColor = ConsoleColor.Yellow;
                                    Console.WriteLine("Queue path not specified.");
                                }
                                break;
                            case "listqueues":
                            case "lq":
                                helper.ListQueues();
                                break;
                            case "getqueue":
                            case "gq":
                                if (commands.Length > 1)
                                {
                                    helper.GetQueue(commands[1]);
                                }
                                else
                                {
```

```
                                    Console.ForegroundColor = ConsoleColor.Yellow;
                                    Console.WriteLine("Queue path not specified.");
                                }
                                break;
                            case "deletequeue":
                            case "dq":
                                if (commands.Length > 1)
                                {
                                    helper.DeleteQueue(commands[1]);
                                }
                                else
                                {
                                    Console.ForegroundColor = ConsoleColor.Yellow;
                                    Console.WriteLine("Queue path not specified.");
                                }
                                break;
                            case "exit":
                                done = true;
                                break;
                            default:
                                break;
                        }
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine();
                    Console.ForegroundColor = ConsoleColor.Red;
                    Console.WriteLine(ex.Message);
                }

            } while (!done);

        }
    }
}
```

## Messages

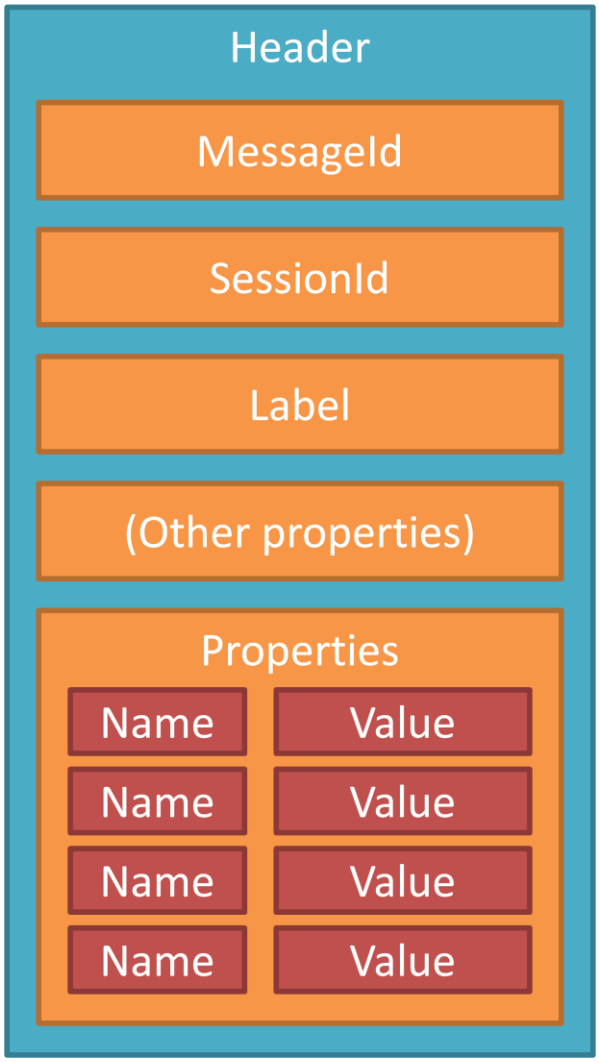
All messages used in the AppFabric service bus brokered messaging services are instances of the BrokeredMessage class. Messages are made up of a header and a body. The message header contains properties and contextual information about the message. The message body contains the payload or data of the message.




**Message Header**

The message header contains a number of fixed properties that are used by the service bus messaging entities and by sending and receiving applications to handle the processing of messages. Some of these properties are used by the messaging entities, others are application specific.

The header also contains a string, object dictionary, named Properties, which can be used to store a number of name-value pairs that can be used for routing messages between topics and subscriptions and also by sending and receiving applications for application specific logic.




The maximum allowed size for a message header is 64 kb.

Be aware that “message properties” could possibly refer to the fixed properties of the message, such as MessageId, and Label, and also to the name-value pairs in the Properties dictionary. I will endeavor to use “message properties collection” when referring to the name-value properties collection.

**Message Body**

The message body contains the serialized data that is transmitted in the message. This is typically the serialized state of an object, or a data stream. In some cases the message body can be empty.

**Message Immutability**

BizTalk developers are familiar with the concept of immutable messages; the same goes in AppFabric brokered messaging. Once an instance of the BrokeredMessage class has been created it is not possible to set or modify the message body content.

**Message Size Limitations**

The AppFabric service bus brokered messaging services have a limit on the size of message that can be placed on a queue or a topic. The current limit on the message size is 256 KB, which includes the message body and header. There is, however, no limit in the size of message that can be created using the BrokeredMessage class. The handling of large messages in the AppFabric brokered messaging services will be covered later in this chapter.

### *BrokeredMessage Class*

All messages in AppFabric service bus messaging are contained in an instance of the BrokeredMessage class. In order to understand and leverage the functionality of AppFabric messaging it is required to have a good understanding of this class.

BrokeredMessage provides a number of properties that are used by the AppFabric messaging artifacts and the applications that send and receive messages. There are four constructors that can be used to create messages from serializable objects, streams, and messages with no body content. Message bodies are immutable once created. The typed GetBody<T> method is used to deserialize the message body to the appropriate object. There are also a number of methods that are used to change the state of messages to set them as completed, abandoned, deferred or dead lettered, these operations also have asynchronous counterparts.

#### Useful Properties

The section provides details of useful properties provided by the BrokeredMessage class.

**ContentType**

The ContentType property is a string field that can be set by the sending application to provide information to the receiving application regarding the content of the message. Applications are free to set this to any value.

**CorrelationId**

The CorrelationId property can be used in correlation filters to route messages and is more performant that using SQL filters.

**DeliveryCount**

The DeliveryCount specifies how many times a message has been received from a queue or topic. When a message is received using the PeekLock mode and Complete is not called within the lock timeout or Abandon is called the message will become visible again with the DeliveryCount property incremented. When processing messages the DeliveryCount property can be checked by the receiving application to prevent poison messages from causing repeated processing failures.

**EnqueuedTimeUtc**

The time at which the message was placed on the queue by the sending application.

**ExpiresAtUtc**

The time at which the message is set to expire. This is a read-only property message expiration should be set using the TimeToLive property.

**Label**

The Label property is similar to the content type property in that it is a string property that is used by sending and receiving applications to set a value used in application specific message processing.

**LockedUntilUtc**

This property will specify when the lock on a message will be released if it has been received using the peek-lock message mode.

**LockToken**

This property is a unique identifier that is used to identify a locked message on a queue. This property is only available when a message has been received using the peek-lock message mode, and can be used by the QueueClient and SubscriptionClient classes to perform actions on locked messages.

**MessageId**

MessageId is a string property used to uniquely identify a message. When a new message is created it will be assigned a new unique identifier for the MessageId property. The sending application can assign its own value to the MessageId property of a message, as duplicate detection is based on this property it can sometimes be useful to do this.

The MessageId has a maximum allowed length of 128 bytes.

**Properties**

The Properties property is a string-object dictionary used for storing application specific message properties. Message properties are typically used for routing messages using topics and subscriptions. If the message body contains values that will be used for routing the message they can be promoted into the message properties.

**ReplyTo**

The ReplyTo property is used in request-response and duplex communication. A sending application can assign a value to this property that the receiving application can use to determine the queue that a response message should be sent on.

**ReplyToSessionId**

The ReplyTo property is used in request-response and duplex communication. A sending application can assign a value to this property that the receiving application can use to determine the value of the SessionId property that should be set on a response message.

**ScheduledEnqueueTimeUtc**

The ScheduledEnqueueTimeUtc property can be set to a time in the future in order to delay the delivery of a message that is placed on a queue.

**SequenceNumber**

The internal sequence number for the message that is assigned by the service bus queue or topic. The property is only available on received messages and is scoped to the queue or topic the message was sent to.

**SessionId**

The CorrelationId property of the message allows a single string value to be set in the message header that can be used in subscription filters to subscribe to messages based on this property.

**Size**

The Size property represents the size of the body in bytes. Note that the physical size of a message includes the message header as well as the message body.

**TimeToLive**

A TimeSpan property of the time a message will remain in the system without being dead lettered or discarded.

**To**

Specified the intended recipient of the message.

## Useful Methods

The following section describes the commonly used methods of the BrokeredMessage class. Asynchronous variants are provided on many of these methods.

**Abandon**

Calling Abandon will release the ownership locks that were placed on the message when it was received. Abandoning messages is useful if an application cannot process a message but wants to leave the message present on the queue. If a message has been received using the PeekLock receive mode this method can be called.

**Complete**

Once a message has been successfully processed calling Complete indicates that the message should be deleted from the queue. If Complete is not called the message will remain in the queue and become visible on the queue when the lock duration timeout expires. If a message has been received using the PeekLock receive mode this method can be called.

**DeadLetter**

The DeadLetter method will abandon processing of a massage and place it on the $DeadLetterQueue sub queue of the queue or subscription that it was received from (confirm subscription). If a message

has been received using the PeekLock receive mode and has not been confirmed or abandoned this method can be called.

**Defer**

The Defer method is used when a message has been received using the PeekLock receive mode, and the processing of the message is to be deferred to a later point in time.

**GetBody<T>**

The GetBody typed method will deserialize the message body into an object of the type specified. The deserialization can use either the default DataContractSerializer or a binary XmlObjectSerializer.

### *Creating Messages*

The BrokeredMessage class provides four constructors that are used to create messages with no body content, or create messages from serializable objects or streams. Once a message is created the message body content cannot be changed.

```
// Create a message with no body content.
public BrokeredMessage ()

// Create a message setting the body content to the contents of a stream.
public BrokeredMessage (Stream, bool)

// Create a message setting the body content to a serializable object.
public BrokeredMessage (object)

// Create a message setting the body content to a serializable object
// using an XML serializer.
public BrokeredMessage (object, XmlObjectSerializer)
```

#### Creating Messages from Serializable Objects

A simple way to create a massage is shown below. As the string class is a serializable object the message will be created with the text as the message body.

```
BrokeredMessage msg = new BrokeredMessage ("Hello, world!");
```

In messaging scenarios it is more common to use classes attributed as data contracts to represent business objects. This is especially useful when using AppFabric messaging together with WCF services as the same data contracts can be used for both technologies. The code below shows a simple data contract for a pizza order.

```
[DataContract]
public class PizzaOrder
{
    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public string Pizza { get; set; }

    [DataMember]
    public int Quantity { get; set; }
}
```

The following code shows how an instance of the PizzaOrder data contract class can be created and then used to construct an instance of the BrokeredMessage class.

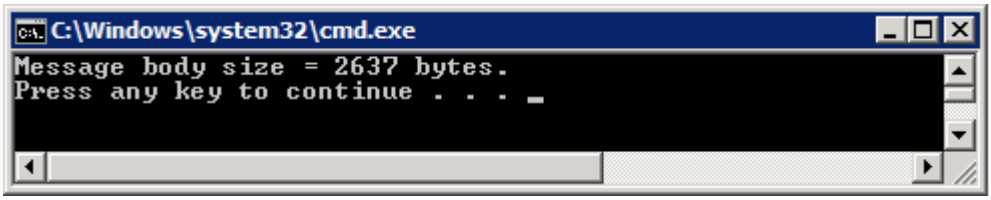
```
// Create a new pizza order.
PizzaOrder order = new PizzaOrder()
{
    Name = "Alan",
    Pizza = "Hawaiian",
    Quantity = 3
};

// Create a brokered message based on the order.
BrokeredMessage orderMsg = new BrokeredMessage (order);
```

## Creating Messages from Streams

Another useful way to create messages is by using streams. The following example shows how to create a message with a photo as the message body. The boolean value in the method specifies whether the message has ownership of the stream. If this is set to true the stream will be closed when the message is closed, if set to false the stream will be left open.

```
// Create a file stream for the photo.
FileStream photo = new FileStream(@"C:\Photos\AlanSmith.jpg", FileMode.Open);

// Create a message from the stream giving the message ownership of the stream.
BrokeredMessage msg = new BrokeredMessage (photo, true);

// Output the message size.
Console.WriteLine("Message body size = {0} bytes.", msg.Size);
```

The following output is displayed, the Size property refers to the body of the message (AlanSmith.jpg is 2,637 bytes), not the total message size, which will include additional data in the header.




When creating the message in this way the message size will be set to the size of the file, but the file stream will not be read into memory until the message is sent or deserialized. Be aware that there is no size limitation when creating messages, but there is a size limitation on sending a massage.

Note that the value returned by the Size property only indicates the size of the message body, there will also be data present in the message header. Once the message has been sent the Size property will return the size of both the message body and header.

### Creating Command and Event Messages

The Enterprise Integration Patterns website defines message construction patterns for command messages, document messages and event messages. The previous examples of message construction have been examples of document messages. It is possible to create messages that do not have any body content. When implementing command message and event message patterns there may be no need for the message to contain any body content.

The Enterprise Integration Patterns website provides a description of the Document Message pattern here.

The Enterprise Integration Patterns website provides a description of the Command Message pattern here.

The Enterprise Integration Patterns website provides a description of the Event Message pattern here.

The following example shows how an event message can be created to specify that a specific order has been completed.

```
// Construct a message with no body content.
BrokeredMessage eventMsg = new BrokeredMessage ();

// Set the appropriate message properties.
eventMsg.ContentType = "Event";
eventMsg.CorrelationId = "OD1345";
eventMsg.Label = "Processed";
```

### *Deserializing Messages*

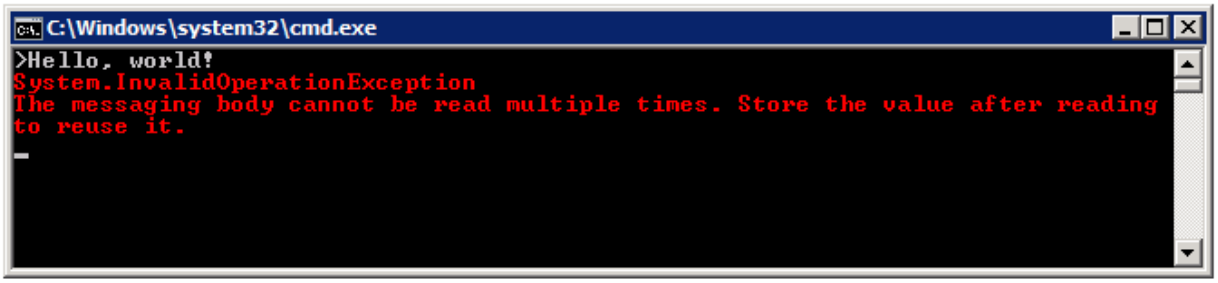
When a receiving application has dequeued a message from a queue or subscription the message body content will usually be deserialized into an object, or read as a stream. The BrokeredMessage class provides a typed GetBody<T> method that will deserialize the content of the message onto an object of the specified type. The following code shows how a message containing a pizza order can be deserialized into a PizzaOrder object.

```
// Create a new pizza order.
PizzaOrder orderIn = new PizzaOrder()
{
    Name = "Alan",
    Pizza = "Hawaiian",
    Quantity = 3
};

// Create a brokered message based on the order.
BrokeredMessage orderMsg = new BrokeredMessage(orderIn);

// Deserialize the content of the order message.
PizzaOrder orderOut = orderMsg.GetBody<PizzaOrder>();
Console.WriteLine("{0} {1} for {2}", orderOut.Quantity, orderOut.Pizza, orderOut.Name);
```

It is important to note that the message body content is based on a forward only stream, and once the content is read it cannot be read again. The following code sample shows an example of this.

```
// Create a BrokeredMessage from a string.
BrokeredMessage simpleMsg = new BrokeredMessage("Hello, world!");

try
{
    // Deserialize the message
    string simpleMessageText1 = simpleMsg.GetBody<string>();
    Console.WriteLine(">" + simpleMessageText1);

    // Deserialize the message again
    string simpleMessageText2 = simpleMsg.GetBody<string>();
    Console.WriteLine(">" + simpleMessageText2);
}
catch (Exception ex)
{
    // Display any exception
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(ex.GetType());
    Console.WriteLine(ex.Message);
}
```

The resulting output in the console is shown below.



### *Using Message Properties*

The BrokeredMessage class provides a number of properties that can be used by applications to that can be used for routing messages and implementing application specific message processing logic. Setting message properties means that they are available in the message header and can be used in rules and actions when using topics and subscribers. It also means that the properties are available to the receiving application without deserializing the message body, which can improve performance. Unlike the message body stream these properties can be read multiple times in a receiving application.

Some message properties will be used by queues, topics and subscriptions when processing messages, other properties are only used by application specific logic in the sending and receiving applications. When a message is created the message properties will be set to the following default values.

| **Property** | **Default Value** |
|---|---|
| ContentType | Null |
| CorrelationId | Null |
| EnqueuedTimeUtc | DateTime.MinValue (Not readable before send.) |
| ExpiresAtUtc | DateTime.MaxValue |
| Label | Null |
| MessageId | A string representation of a new GUID. |
| Properties | New IDictionary <string, object> |
| ReplyTo | Null |
| ReplyToSessionId | Null |
| ScheduledEnqueueTimeUtc | DateTime.MinValue |
| SessionId | Null |
| Size | The size of the message body in bytes. |
| TimeToLive | TimeSpan.MaxValue |
| To | Null |

**MessageId**

The MessageId property is used to uniquely identify a message in the AppFabric messaging services. Then a message is created string representation of a new GUID will be assigned to this property, ensuring that its value will be unique. The MessageId property is used by AppFabric queues and topics when detecting duplicate messages.

As all messages created in AppFabric will have unique values by default a sending application may need to overwrite the default MessageId value with another value to ensure that duplicate message detection produces the desired results. An example of using this technique to handle duplicate messages is shown in the Exploring AppFabric Messaging Features section.

```
// Create a new pizza order.
PizzaOrder orderIn = new PizzaOrder()
{
    Name = "Alan",
    Pizza = "Hawaiian",
```

```
    Quantity = 1
};

// Create a new brokered message from the RFID tag.
BrokeredMessage tagReadMsg = new BrokeredMessage (rfidTag);

// Set the Message Id to the ID of the RFID tag.
tagReadMsg.MessageId = rfidTag.TagId;
```

**ContentType**

Applications have the option of specifying the content type for messages. The ContentType property is a string field that can be set by the sending application to provide information to the receiving application regarding the content of the message. Applications are free to set this to any value; however it would be recommended to use pre-defined MIME types.

The following example shows how the content type property of a message can be set to the mime type for a JPEG image.

```
// Create a file stream for the photo.
FileStream photo = new FileStream(@"C:\Photos\AlanSmith.jpg", FileMode.Open);

// Create a message from the stream giving the message ownership of the stream.
BrokeredMessage msg = BrokeredMessage.CreateMessage(photo, true);

// Set the content type of the message.
msg.ContentType = "image/jpeg";
```

**Label**

Label is a string field that can be used by sending and receiving applications to implement application specific logic. Setting this value will have no effect on the way that the messages are processed by queues topics and subscriptions.

The following code will create a message and label it as a test message. A receiving application can examine the value of the Label property and process it accordingly.

```
// Create a brokered message based on the order.
BrokeredMessage orderInMsg = new BrokeredMessage(orderIn);

// Use the labelproperty to indicate that this is a test message.
orderInMsg.Label = "TestMessage";
```

**Properties**

The properties property is a generic IDictionary<string, Object> class that can be used to store multiple message properties that are used in filters in subscriptions to subscribe to messages.

The following code will create a message and add items to the property collection.

```
// Create a message from the order.
BrokeredMessage orderMsg = new BrokeredMessage(order);

// Set message properties from values in the order.
orderMsg.Properties.Add("loyalty", order.HasLoyltyCard);
orderMsg.Properties.Add("items", order.Items);
orderMsg.Properties.Add("value", order.Value);
orderMsg.Properties.Add("region", order.Region);
```

**CorrelationId**

The CorrelationId property of the message allows a single string value to be set in the message header that can be used in subscription filters to subscribe to messages based on this property. This is more efficient than adding an item to the message properties collection when only one string value is requires in the filter.

The following code will create a message and set the CorrelationId property to the region of the order.

```
// Create a message from the order.
BrokeredMessage orderMsg = new BrokeredMessage(order);

// Set the CorrelationId to the region.
orderMsg.CorrelationId = order.Region
```

**TimeToLive**

The TimeToLive property of a message determines the value that will be set for the ExpiresAtUtc property of the message when it is sent.

The following code will create an order message and set the message to expire 30 days from when it is sent to the queue.

```
// Create a message from the order.
BrokeredMessage orderMsg = new BrokeredMessage(order);

// Set the message time to live to 30 days.
orderMsg.TimeToLive = TimeSpan.FromDays(30);
```

**SessionId**

SessionId is a string property that is used when messages are to be correlated by a receiving application. This can be when a group of messages related to each other are to be sent in a batch, but also when a request-response messaging pattern is used.

If a number of messages need to be grouped together and received by the same instance of a receiving application in a load balanced environment assigning them with the same value of SessionId and receiving them as a session will ensure this happens.

The following code will send a batch of pending order messages, all assigned with the same value for SessionId.

```
List<PizzaOrder> Orders = OrderManager.GetPendingOrders();

string orderBatchId = Guid.NewGuid().ToString();

foreach (PizzaOrder order in Orders)
{
    // Create a brokered message based on the order.
    BrokeredMessage orderMsg = new BrokeredMessage(orderIn);

    // Set the SessionId.
    orderMsg.SessionId = orderBatchId;

    // Send the message.
    queueClient.Send(orderMsg);
}
```

**ScheduledEnqueueTimeUtc**

Setting the ScheduledEnqueueTimeUtc of a message allows the message to be sent to a queue where it will be stored durably until the specified time. It will then be enqueued and will can be received by a receiving application.

The following code will create on order for a pizza that will be available for processing one hour from the time at which it was sent to the queue.

```
// Create a new pizza order.
PizzaOrder orderIn = new PizzaOrder()
{
    Name = "Alan",
    Pizza = "Hawaiian",
    Quantity = 1
```

```
};

// Create a brokered message based on the order.
BrokeredMessage orderInMsg = new BrokeredMessage(orderIn);

// Schedule the order to be one hour from now.
orderInMsg.ScheduledEnqueueTimeUtc = DateTime.UtcNow.AddHours(1);

// Send the message to the queue.
queueClient.Send(orderInMsg);
```

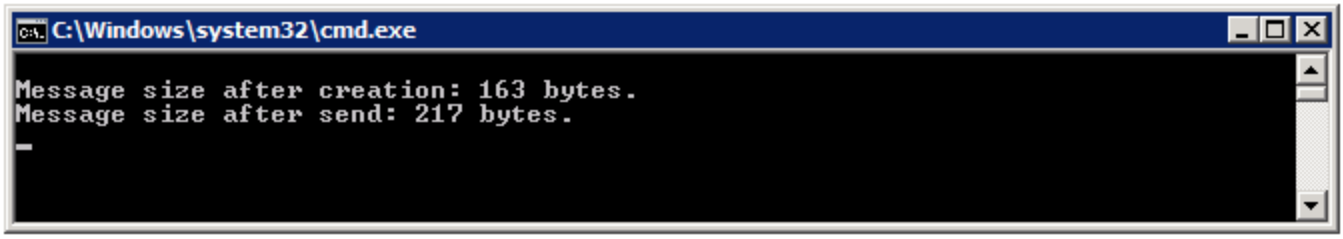
**Size**

The Size property of the message can be used to determine the combined size of the message header and body in bytes. There are, however, some issues with using this property to determine an accurate value for the size of the message due to the way that it is calculated.

In order to illustrate this, the following code was written to create a new message and send it to the queue. The size property of the message is written to the console before and after the message has been sent.

```
// Create a new pizza order.
PizzaOrder orderIn = new PizzaOrder()
{
    Name = "Alan",
    Pizza = "Hawaiian",
    Quantity = 1
};

// Create a brokered message based on the order.
BrokeredMessage orderInMsg = new BrokeredMessage(orderIn);

Console.WriteLine("Message size after creation: {0} bytes.", orderInMsg.Size);

// Send the message to the queue.
queueClient.Send(orderInMsg);

Console.WriteLine("Message size after send: {0} bytes.", orderInMsg.Size);
```

When the code is executed the output is as follows.

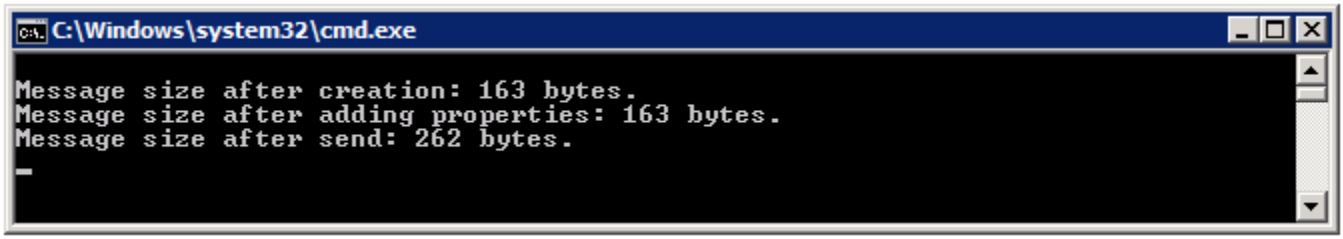



This shows a difference in message size of 54 bytes. The reason for this is that before the message has been sent the size property returns only the size of the message body. When the message is sent the message header is serialized, and after sending the message it will be available in the size property.

This is further illustrated by adding code to add items to the message property collection in the message header.

```
// Create a new pizza order.
PizzaOrder orderIn = new PizzaOrder()
{
    Name = "Alan",
    Pizza = "Hawaiian",
    Quantity = 1
};

// Create a brokered message based on the order.
BrokeredMessage orderInMsg = new BrokeredMessage(orderIn);

Console.WriteLine("Message size after creation: {0} bytes.", orderInMsg.Size);

// Add properties to the message
orderInMsg.Properties.Add("name", orderIn.Name);
orderInMsg.Properties.Add("pizzas", orderIn.Quantity);
orderInMsg.Properties.Add("branch", "Bromma");

Console.WriteLine("Message size after adding properties: {0} bytes.", orderInMsg.Size);

// Send the message to the queue.
queueClient.Send(orderInMsg);

Console.WriteLine("Message size after send: {0} bytes.", orderInMsg.Size);
```

When the code is executed the following results are displayed.



The value returned by the size property of the message remains unchanged after items are added to the message properties collection. Once the message has been sent its size is 262 bytes, 45 bytes more than the message that was sent without items in the properties collection.

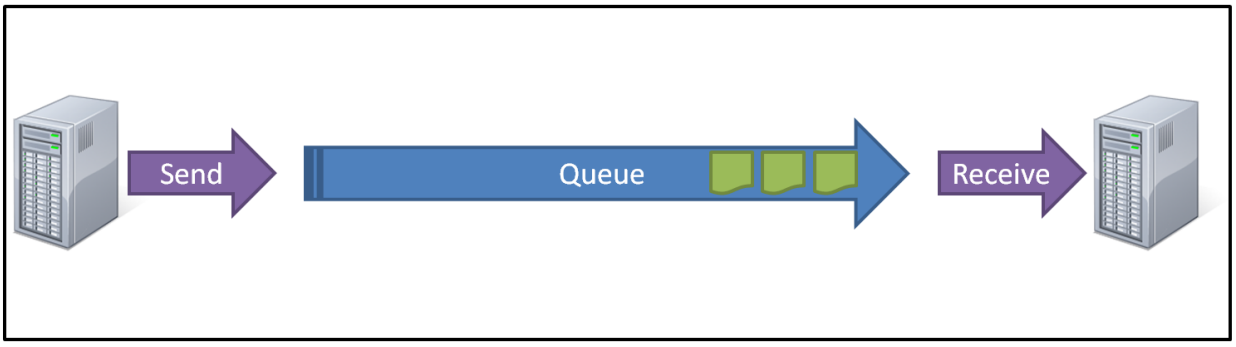
To summarize this, the size property will only return the body size of the message before the message is sent. The header size is calculated during the send operation, and is then used to calculate the value of the Size property.

- Before Send: Size = Body Size
- After Send: Size = Header Size + Body Size

As there is a 256KB limit in the size of a message that can be sent to the service bus, it is recommended that sending applications verify the size of the messages before sending then, especially if another application may be sending data items that could exceed this size. Carful thought to the size of the message header must be taken into account, and a margin allowed for any variations. As the maximum size of the message header is 64 Kb it can be assumed that a message with a body of up to 192 Kb will not be too large to send.

## Using Queues

AppFabric service bus queues provide a traditional first in first out (FIFO) model for exchanging messages. Messages are typically sent to a queue from a sending application and received by a receiving application. Although the AppFabric messaging infrastructure is hosted in Windows Azure data center the sending and receiving applications can be cloud-based or on-premise. This makes AppFabric queues a versatile communication mechanism.




Whilst AppFabric queues have some similarities with the queues found in Windows Azure Storage services they provide more sophisticated features that make them a much better choice when implementing enterprise applications. AppFabric queues support sessions, dead lettering, message deferral, duplicate detection and message expiration.

This section will provide an overview of the classes used to connect to AppFabric queues and send and receive messages. Many of the topics covered in this section can be applied when sending messages to topics and receiving messages from subscriptions.

### *MessageClientEntity Class*

The MessageClientEntity class provides an abstract class for client classes that will interact with messaging entities in a service bus namespace. The following classes are derived from MessageClientEntity.

- MessageReceiver
- MessageSender
- MessagingFactory
- QueueClient
- SubscriptionClient
- TopicClient
- MessageSession

These classes are thread safe and there is a slight overhead when initializing the connection to the service bus. This makes it most efficient to reuse the objects as much as possible; however holding unused connections to the service bus can be a drain of resources.

The MessageClientEntity class is similar in nature to the CommunicationObject class used in WCF, and provides an implementation of the connection between client applications and the service bus. Any class that derives from MessageClientEntity will create a connection to the service bus when first used. As service bus connections are limited for messaging entities, and billed for on an hourly basis it is important to close these connections appropriately.

### *MessagingFactory Class*

The MessagingFactory is a factory class used to create clients for queues, topics and subscriptions. MessagingFactory provides a static overloaded Create method that is used to create a message factory for a specified service bus namespace and credentials. Once created the MessagingFactory can be used to create clients for queues, topics and subscriptions within that namespace.

**Creating a MessagingFactory**

The static overloaded Create method returns an in initialized instance of the MessagingFactory class. The four implementation of this method are shown below.

```
public static MessagingFactory Create(string address, TokenProvider tokenProvider)

public static MessagingFactory Create(Uri address, TokenProvider tokenProvider)

public static MessagingFactory Create(string address, MessagingFactorySettings
    factorySettings)

public static MessagingFactory Create(Uri address, MessagingFactorySettings
    factorySettings)
```

Details of the service bus namespace and credentials are required for this operation and can be passed in a number of ways. The MessagingFactorySettings can also be used to override the default operational timeout of 60 seconds.

The simplest way to create a MessagingFactory instance is to create a TokenProvider with the appropriate credentials and a Uri with the appropriate namespace, then call the static Create method. The following code will create an instance of the messaging factory with the default settings using a namespace and credential stored in a class named AccountDetails.

```
// Create a token provider with the relevant credentials.
TokenProvider credentials =
    TokenProvider.CreateSharedSecretTokenProvider
    (AccountDetails.Name, AccountDetails.Key);

// Create a URI for the serivce bus.
Uri serviceBusUri = ServiceBusEnvironment.CreateServiceUri
    ("sb", AccountDetails.Namespace, string.Empty);

// Create a message factory for the service bus URI using the
// credentials
MessagingFactory factory = MessagingFactory.Create
    (serviceBusUri, credentials);
```

There are two settings that can be specified when creating a MessagingFactory that will affect its runtime behavior. The MessagingFactorySettings class exposes these settings as properties and can be specified as a parameter when creating an instance of MessagingFactory. Once a MessagingFactory instance has been created it is not possible to alter, or even view, these settings.

*OperationTimeout*

This is a TimeSpan property with a default value of 60 seconds. The OperationTimeout that is specified when the MessagingFactory is created will be used as the operation timeout for all communication objects that are created with the factory methods.

*NetMessagingTransportSettings.BatchFlushInterval*

This is a TimeSpan property with a default value of 20ms. According to the documentation this property "Gets or sets the batch flush interval". The use of this property will be investigated and detailed in a future release of this book.

The following code will create a MessagingFactory instance with a reduced OperationTimeout and increased BatchFlushInterval.

```
// Create a token provider with the relevant credentials.
TokenProvider credentials =
    TokenProvider.CreateSharedSecretTokenProvider
    (AccountDetails.Name, AccountDetails.Key);

// Create a URI for the serivce bus.
Uri serviceBusUri = ServiceBusEnvironment.CreateServiceUri
    ("sb", AccountDetails.Namespace, string.Empty);

// Create a MessagingFactorySettings instance with the appropriate
// settings.
MessagingFactorySettings factorySettings = new MessagingFactorySettings()
{
    TokenProvider = credentials,
    OperationTimeout = TimeSpan.FromSeconds(10),
    NetMessagingTransportSettings = new NetMessagingTransportSettings()
    {
        BatchFlushInterval = TimeSpan.FromMilliseconds(100)
    }
};

// Create a message factory for the service bus URI using the
// credentials
MessagingFactory customFactory = MessagingFactory.Create
    (serviceBusUri, factorySettings);
```

**Creating Clients**

The MessagingFactory class provides three methods for creating clients for messaging entities, CreateQueueClient, CreateSubscriptionClient and CreateTopicClient. There are also asynchronous versions of these methods available. These methods will return a client object for the specified queue, topic or subscription.

The method definitions for creating clients are show below.

```
public QueueClient CreateQueueClient(string path)

public QueueClient CreateQueueClient(string path, ReceiveMode receiveMode)

public TopicClient CreateTopicClient(string path)

public SubscriptionClient CreateSubscriptionClient(string topicPath, string name)

public SubscriptionClient CreateSubscriptionClient(string topicPath, string name,
    ReceiveMode receiveMode)
```

Queue clients are created by specifying the path of the queue, and optionally the receive mode. The receive mode can be either PeekLock or ReceiveAndDelete, and will default to PeekLock if not specified explicitly. Topic clients are created by specifying the path of the topic. Subscription clients are created by specifying the path of the topic that contains the topic that contains the subscription and the subscription name, with an option of specifying the receive mode. The same rules apply for the default receive mode when creating subscription clients.

### *QueueClient Class*

The QueueClient class is an abstract class used to send and receive messages to an AppFabric service bus queue. The QueueClient class derives from MessageClientEntity and uses internal instances of the MessageSender and MessageReceiver classes for the send and receive operations.

**Massage Send and Receive Operations**

The following methods are provided for sending and receiving messages.

```
public void Send(BrokeredMessage message)

public BrokeredMessage Receive()

public BrokeredMessage Receive(TimeSpan serverWaitTime)

public BrokeredMessage Receive(long sequenceNumber)
```

The Send method is self explanatory, it will enqueue the specified BrokeredMessage instance on the queue that the QueueClient is connected to.

The Receive method provides three overloads; the first will receive a message from the queue. If no message is present the method will block the flow of execution until a message is available, or until the operation timeout expires. If no message is received during this time the method will return null.

The operation timeout of the QueueClient is set to the operation timeout of the MessagingFactory that created the queue client, which will have a default value of 60 seconds. If a different timeout for a receive operation is required this can be specified as a parameter in the second overloaded method.

The third overloaded method allows a sequence number of a specific message to be specified. Receiving messages based on the sequence number is used when a message has been deferred by a receiving application.

**Message State Operations**

The QueueClient class provides a number of methods that can be used to change the state of messages received in the PeekLock receive mode. The BrokeredMessage class also provides similar methods that allow message state to be changed.

```
public void Complete(Guid lockToken)

public void Abandon(Guid lockToken)

public void DeadLetter(Guid lockToken)

public void DeadLetter(Guid lockToken, string deadLetterReason,
    string deadLetterErrorDescription)

public void Defer(Guid lockToken)
```

Each of these methods requires the LockToken of the message to be specified. The LockToken is a unique identifier that is assigned as a message property when a message is received in the PeekLock receive mode. One of the DeadLetter implementations allows error information to be added to the deadlettered message.

## *Sending Messages*

Messages can be sent to queues or topics and the same techniques are used for both scenarios. The MessageSender class is used internally by the QueueClient and TopicClient classes for sending messages.

### Understanding Message Streams

The BrokeredMessage class uses an internal stream that is used to read the message body content. Once read this stream cannot be rewound, meaning that once created a message can only be sent once. In scenarios where the same data needs to be sent multiple times it is required to create one message instance for each send. If a send operation fails or times out, and data is to be resent then the message should be created again.

### Sending Messages Synchronously

Messages are sent to Queues and Topics in a synchronous fashion using the Send method. The method takes BrokeredMessage as a parameter. An example of the code used to create a QueueClient, and send a message is shown below.

```
// Create a queue client for the pizzaorders queue
QueueClient queueClient = factory.CreateQueueClient("pizzaorders");

// Create a new pizza order.
PizzaOrder orderIn = new PizzaOrder()
{
    Name = "Alan",
    Pizza = "Hawaiian",
    Quantity = 1
};

// Create a brokered message based on the order.
BrokeredMessage orderInMsg = new BrokeredMessage(orderIn);

// Send the message to the queue.
queueClient.Send(orderInMsg);

// Close the client
queueClient.Close();
```

### Sending Messages Asynchronously

As the AppFabric service bus infrastructure is hosted in an Azure data center there is likely to be a significant latency using the synchronous Send method. This latency could affect the throughput of an application that is attempting to send a number of messages in a short time interval.

When designing applications it is important to take into account the effect that the location of your application will have on the latency of messaging operations. When preparing for and presenting a session at the Microsoft campus in Redmond in 2008 using the AppFabric service bus, which at the time was called BizTalk Services, I experienced a 600 ms latency in Stockholm, a 100 ms latency in Kirkland (a short drive from the Microsoft campus), and a 50ms latency during the presentation on the Microsoft campus. If you are performance testing your application close to the datacenter hosting the messaging infrastructure your results will be very different to what your customer may experience from a different location.

A short non-scientific test of sending messages to a queue hosted in the Europe (West) data center, which is located in Amsterdam, from a client in Stockholm gave the following results.

- Sending synchronously – About 14 messages per second
- Sending asynchronously –About 520 messages per second

Bear in mind that this was a quick "coffee break" test that shows asynchronous sending is about 35 times faster than synchronous sending. When using service bus in a real-world scenario where performance is an issue, robust performance testing should be undertaken.

The following code sample shows how to send a message using the asynchronous BeginSend method.

```
// Create a new BrokeredMessage from the order data contract.
BrokeredMessage orderMsg = new BrokeredMessage(order);

// Send the order.
//queueClient.Send(orderMsg);
queueClient.BeginSend(orderMsg, OnSendComplete,
    new Tuple<QueueClient, string>(queueClient, orderMsg.MessageId));
messageSendPerSecCounter.Increment();

Console.WriteLine("Sent order: {0}", order.OrderNumber);
```

The following code shows the implementation of the OnSendComplete method.

```
public static void OnSendComplete(IAsyncResult result)
{
    Tuple<QueueClient, string> stateInfo = (Tuple<QueueClient, string>)result.AsyncState;

    QueueClient queueClient = stateInfo.Item1;
    string messageId = stateInfo.Item2;

    try
    {
        // Complete Asynchronous Message Send process
        queueClient.EndSend(result);
    }
    catch (Exception e)
```

```
    {
        Console.WriteLine("Error for message = {0} - {1}", messageId, e.ToString());
    }
}
```

### *Receiving Messages*

Messages can be received from queues or subscriptions and the same techniques are used for both scenarios. The MessageReceiver class is used internally by the QueueClient and SubscriptionClient classes for receiving messages. Messages can also be received as sessions using the MessageSession class, which derives from MessageReceiver.

There are three techniques that can be used for receiving messages.

- Receive and Delete Receive Mode
- Peek Lock Receive Mode
- Transactional Receives

**Receive and Delete Receive Mode**

The receive and delete mode is the simplest way of receiving messages from a queue. When using this mode messages will be received from the queue and marked as complete in one operation, causing them to be deleted from the queue. Although this is the simplest way of receiving messages it is not the most optimal. If the receiving application fails to process the message successfully it will be lost. Once a message is received using this mode it is not possible to defer, dead-letter or abandon processing of the message. Using the receive and delete mode provides at-least-once processing semantics.

The following code shows how to create a queue client using the receive and delete receive mode, and then receive any messages that are on the queue.

```
// Use the MessagingFactory to create a queue client for the orderqueue.
QueueClient queueClient = factory.CreateQueueClient
    ("orderqueue", ReceiveMode.ReceiveAndDelete);

// Receive messages from the queue with a 10 second timeout.
while (true)
{
    // Receive a message using a 10 second timeout
    BrokeredMessage msg = queueClient.Receive(TimeSpan.FromSeconds(10));

    if (msg != null)
    {
        // Deserialize the message body to an order data contract.
        Order order = msg.GetBody<Order>();

        // Output the order.
        Console.WriteLine("{0} {1} {2} {3} {4} ${5}",
            order.OrderNumber,
            order.Customer.FirstName,
            order.Customer.LastName,
            order.ShipTo.City,
            order.ShipTo.Province,
            order.Total);
    }
```

```
    else
    {
        // No message has been received, we will poll for more messages.
        Console.WriteLine("Polling, polling, polling...");
    }
}
```

**Peek Lock Receive Mode**

The peak-lock receive mode uses a two-phase protocol to receive messages from queues and subscriptions. It is the default receive mode for queue and subscription clients. When a message is received it is placed into a locked state on the queue and will remain in that state until the state is changed, or a lock timeout occurs, at which point it will become visible again.

Whilst the message is in the locked state the receiving application can perform one, and only one, of the following actions.

- Complete – The message is marked as complete; this should be done when the message has been processed successfully.
- Abandon – The message is unlocked and available for receiving again, this should be done when the processing of a message has failed and the failure was not due to the message content.
- Defer – The message remains on the queue in a deferred state and can be received at a later point in time using the message sequence id. This should be done when a receiving application will process the message at a later point in time.
- DeadLetter – The message is places on a deadletter sub-queue and can be received and processed at a later point in time. This should typically be done when the processing of a message has failed due to the message content.

The following code shows how a message can be received using the default peek lock receive mode, and then completed if processed successfully, or abandoned if an error occurred adding the order to the database.

```
// Use the MessagingFactory to create a queue client for the orderqueue.
QueueClient queueClient = factory.CreateQueueClient("orderqueue");


// Receive messages from the queue with a 10 second timeout.
while (true)
{
    // Receive a message using a 10 second timeout
    BrokeredMessage msg = queueClient.Receive(TimeSpan.FromSeconds(10));

    if (msg != null)
    {
        // Deserialize the message body to an order data contract.
        Order order = msg.GetBody<Order>();
```

```
        // Output the order.
        Console.WriteLine("{0} {1} {2} {3} {4} ${5}",
            order.OrderNumber,
            order.Customer.FirstName,
            order.Customer.LastName,
            order.ShipTo.City,
            order.ShipTo.Province,
            order.Total);

        // Update the database
        try
        {
            // Add the order to the database.
            OrdersData.AddOrder(order);

            // Mark the message as complete.
            msg.Complete();
        }
        catch (Exception ex)
        {
            Console.WriteLine("Exception: {0}", ex.Message);

            // Something went wrong, abandon the message.
            msg.Abandon();
        }
    }
    else
    {
        // No message has been received, we will poll for more messages.
        Console.WriteLine("Polling, polling, polling...");
    }
}
```

Note that the above solution could fail repeatedly for the same message if something in the content of the message was causing the exception when adding the order to the database. If this was to happen the message would automatically be deadlettered when the DeliveryCount of the message reached the MaxDeliveryCount of the queue. The default MaxDeliveryCount for a queue is 10.

**Received Message Lock Timeout**

The lock timeout is set on the queue as the LockDuration property, and has a default value of 60 seconds and a maximum value of 5 minutes. Unlike Azure Storage queues it is not possible for a receiving application to specify a value for the lock timeout when receiving a massage, nor is it possible to change the LockDuration property on a queue once it has been created. Care should be taken when deciding on a value for the LockDuration property of a queue created for use in production systems as it can have significant impact on the processing of messages in receiving applications.

If the processing time of a message in a receiving application exceeds the lock timeout, the message will become visible to other applications, and data duplication can occur if messages are processed more than once. An exception will be thrown in the receiving application if it attempts to call complete on a message when the lock timeout has expired. The receiving application can check when the lock timeout expires by checking the LockedUntilUtc property of the received message. Bear in mind that this property will be based on the setting of UTC time at the Azure data center, and there may be a slight time difference between that time and the time on the computer hosting the receiving application.

**Poison Messages**

If a message is received using peek-lock mode and the processing of the message fails the receiving application can call the Abandon message to release the lock and make it available for receiving again. If there is something about that particular message that is causing processing to fail the message will be stuck in an endless loop, and will be received and abandoned multiple times. Messages that cause this kind of failure are known as poison messages, and receiving applications should include the capability to identify and handle them accordingly.

The BrokeredMessage class provides a DeliveryCount property that identifies the number of times a message has been received from a queue or subscription. A receiving application could have a delivery count threshold where any message received with a DeliveryCount greater than that value is assumed to be a poison message, and can be placed on a dead-letter queue. The queue also contains a MaxDeliveryCount property that can automatically deadletter messages that have been received a specified number of times.

## Polling Consumers, Event Driven Consumers and 'Sticky Polling'

The enterprise integration patterns book defines two patterns for the way in which messaging endpoints can consume messages from a messaging channel.

**Polling Consumer**

Polling consumers will poll a message channel until a message becomes available. BizTalk developers are familiar with the Polling consumer pattern as it is commonly used in adapters to receive messages from line-of-business applications. The CloudQueue.GetMessage in the Windows Azure storage client library operates on the same principle. If a message is not present on the queue the method returns null, and the application will wait a while before polling the queue again.

As a polling interval is used to delay between each polling operation developers typically have a compromise between message receive latency and the load placed on the message channel. Reducing the polling interval will reduce the receive latency, but also increase the load.

| |
|---|
| The Enterprise Integration Patterns website provides a description of the Polling Consumer pattern here. |

**Event-Driven Consumer**

In an event driven consumer an event is created in the system when a message received, or is ready to be received. Using this pattern can greatly reduce the latency in messaging systems as an event can be raised as soon as a message is available, and there is no polling taking place on the channel.

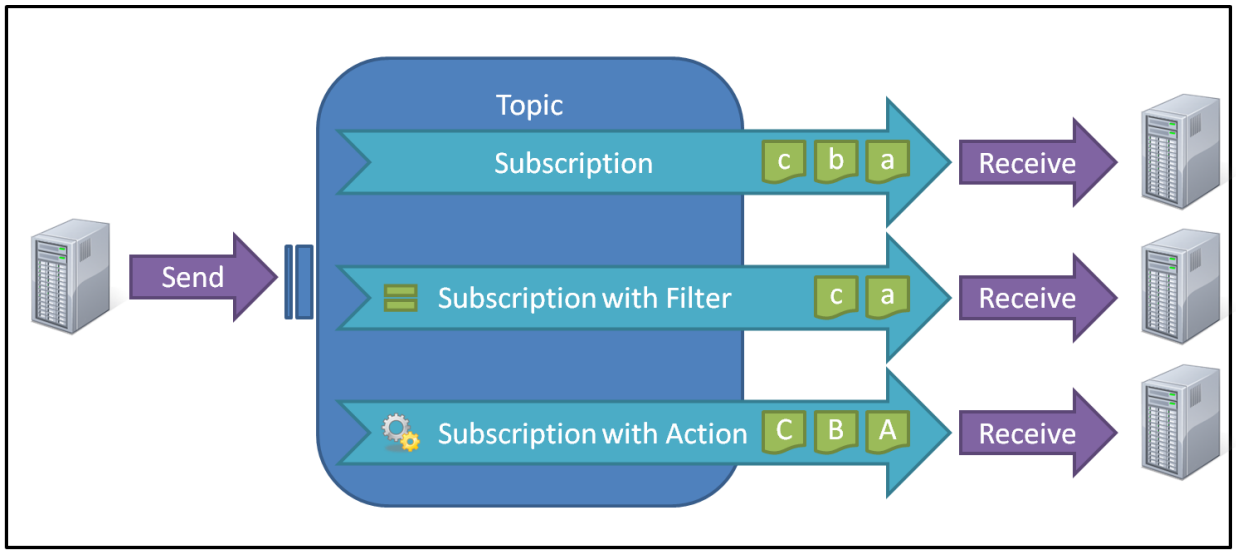
The Enterprise Integration Patterns website provides a description of the Event-Driven Consumer pattern here.

**"Sticky Polling"**

Strictly speaking, when receiving messages from queues and subscriptions, the MessageReceiver and MessageSession class are use the polling consumer pattern. However, they also provide the low-latency for detecting and receiving messages of an event driven consumer. When the Receive method is called on a MessageReceiver a poll operation will start on the queue. Instead of returning immediately the operation will keep the poll operation open until either a message is received, or a timeout expires. If a message is received the method will receive the message immediately and return it to the calling application. This "sticky polling" technique provides a very low receive latency without having to resort to a short polling interval.

## Using Topics and Subscriptions

AppFabric messaging provides a publish-subscribe messaging implementation that can be used in scenarios where an application sends a message that is to be received by one or more subscribers. Whilst queues are used for creating point-to-point messaging channels, topics and subscriptions are used for publish-subscribe channels.



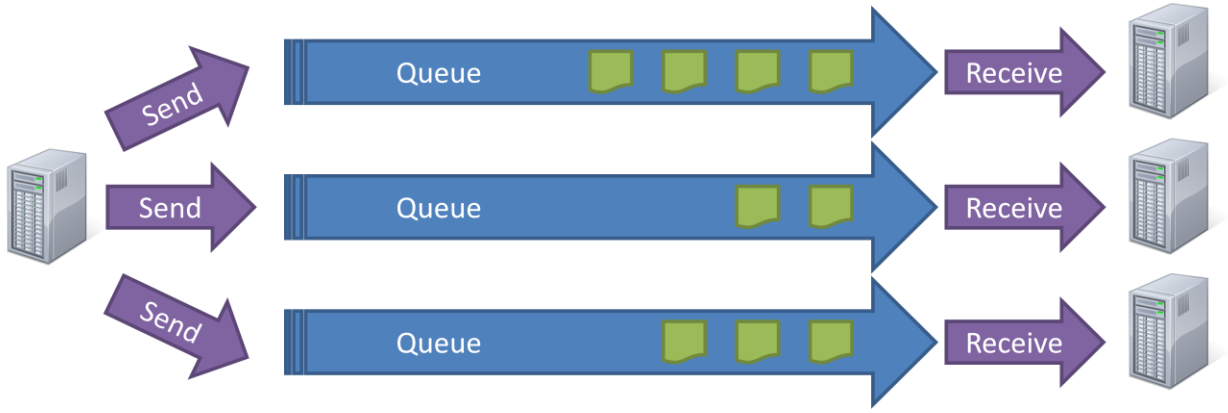
### *Publish-Subscribe Messaging*

In AppFabric messaging a queue is used to implement a point-to-point message channel. This ensures that messages sent on the message channel will only be received by one receiver. Point-to-point message channels are most useful when an application will send messages to another application.

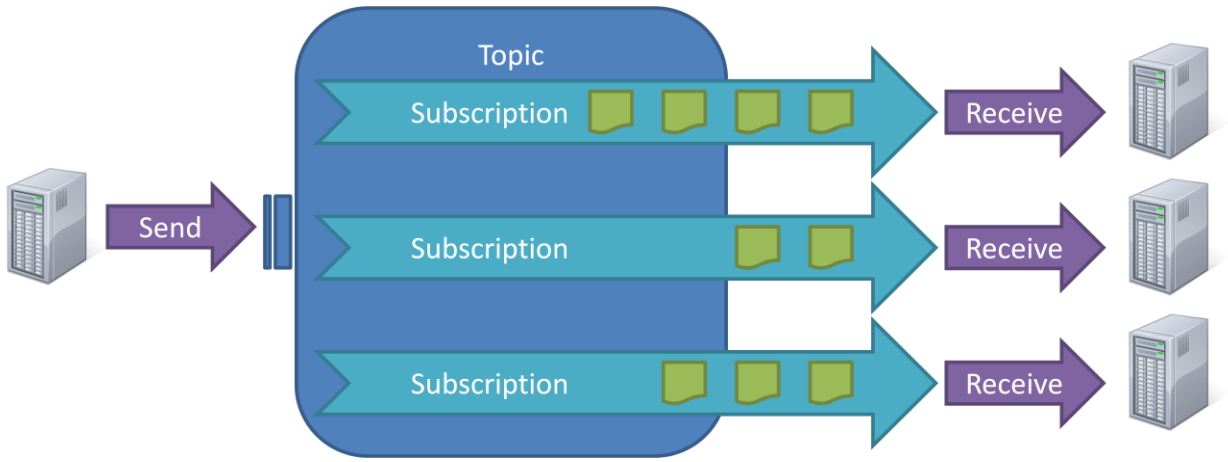
The Enterprise Integration Patterns website provides a description of the Point-to-Point Channel pattern here.

In some scenarios a message that is sent by an application will need to be received by several other applications. Consider an order processing scenario where order messages are created by a point of sale application in a store. The order message will need to be sent to the line-of-business application that handles the finance aspects of the organization. It will also need to be sent to the warehouse application that will need to be aware that products have been sold so it can restock those products. If the customer has a loyalty card the order details will need to be sent to an application that assimilates information about the customers purchasing habits.

If point-to-point channels was used for this scenario the point of sale terminal would have the responsibility of sending the order messages to the various systems. If systems were later added or modified this would mean modifying the point of sale application. In scenarios where many line of business systems are exchanging messages with many other systems the point-to-point channels pattern can become unmanageable.




Publish-subscribe channels provide a way of removing the sender of the responsibility of sending messages to specific applications. The logic that decides where the messages will be sent is contained within the message channel. Publish-subscribe channels also allow for one input message result in a number of output messages being sent to different systems based on this routing logic.

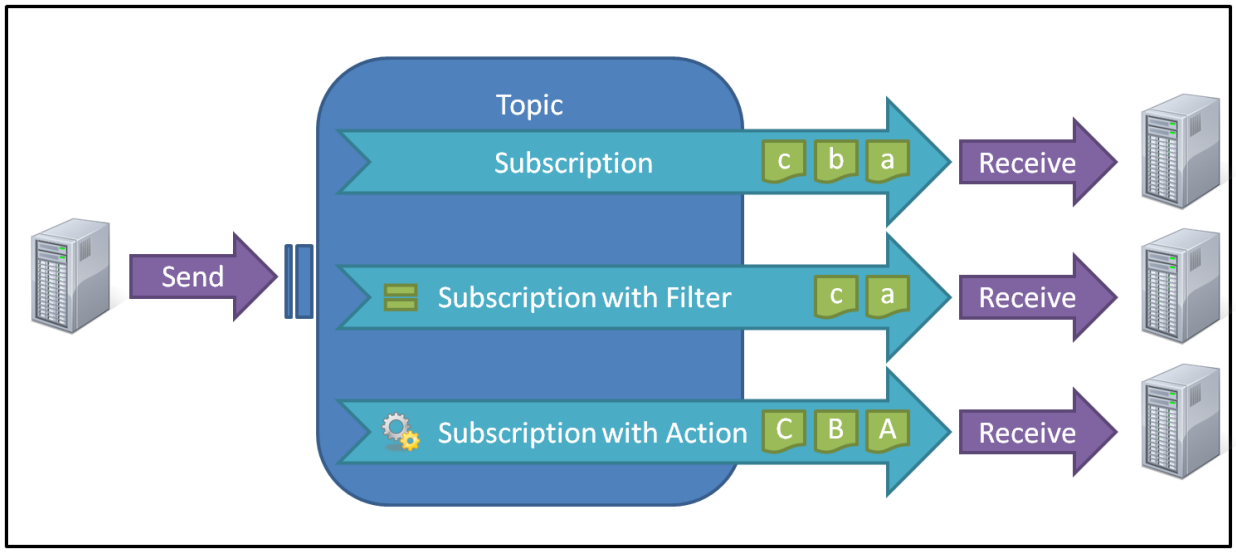



The Enterprise Integration Patterns website provides a description of the Publish-Subscribe Channel pattern here.

It is possible to create a point-to-point message channel using a topic with one subscription. This may be desirable in scenarios where changes are likely to be made to systems running in production. Be aware that the entity hour pricing model for AppFabric brokered messaging services may make this more expensive than using a single queue, and there may be a small additional overhead in message processing.

### *Topics and Subscriptions*

In AppFabric topics and subscriptions are used to implement publish subscribe channels.




**Topics**

Topics are similar to the enqueueing end of a queue. Applications send messages to topics in exactly the same way that they send them to queues by using a MessageSender object. Topics provide no interface for dequeuing messages, instead they have a collection of zero or more subscriptions that will subscribe to the messages sent to the topic based on filter rules. Topics provide support for message expiration, deadlettering and duplicate detection. The techniques for sending messages to queues can be used to send messages to topics. These are discussed in the Sending Messages section of Using Queues.

**Subscriptions**

Subscriptions are similar to the dequeuing end of a queue. Applications receive messages from subscriptions in exactly the same way that they receive them from queues by using a QueueClient or MessageSession object. A topic can have zero or more subscriptions, but a subscription can only belong to one topic. Subscriptions provide no external interface for enqueueing messages, internally messages are enqueued on the subscription inside the publish-subscribe channel based on routing logic in the subscription filters.

Subscriptions provide support for message expiration, dead lettering and message sessions. As topics provide support for duplicate message detection there is no need for this to be implemented on subscriptions as any duplicate messages should be detected by the topic before reaching the subscription. The techniques for receiving messages from queues can be used to receive messages from subscriptions. These are discussed in the Receiving Messages section of Using Queues.

**Filter Expressions**

Filter expressions are used to determine which of the messages sent to the topic the subscription will subscribe to. There are currently four types of filter that can be added to a subscription.

- SqlFilter – Subscribes to messages based on a T-SQL like expression based using values in the message property dictionary
- CorrelationFilter – Subscribes to messages based on the value of the CorrelationId property of the message
- True Filter– Messages are always subscribed to
- False Filter – Messages are never subscribed to

## Walkthrough: Introducing Topics and Subscriptions

The following walkthrough will show how topics and subscriptions can be used to implement a simple publish-subscribe messaging channel. The next walkthrough will build on this sample and explore the use of filters on subscriptions.

If you want to run through the sample create a new console application, set the Target Framework to .NET 4.0 and add the following assemblies.

- Microsoft.ServiceBus
- System.Runtime.Serialization

**Creating the Data Contract**

The application will send order messages through the publish-subscribe channel. The following data contract is used to represent an order. This is created as a separate class in the project.

```
[DataContract]
class Order
{
    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public DateTime OrderDate { get; set; }

    [DataMember]
    public int Items { get; set; }

    [DataMember]
    public double Value { get; set; }

    [DataMember]
    public string Priority { get; set; }

    [DataMember]
    public string Region { get; set; }

    [DataMember]
    public bool HasLoyltyCard { get; set; }
}
```

**Adding Account Details**

An AccountDetails class is also added and the values set for the namespace and credentials.

```
class AccountDetails
{
    public static string Namespace = "";
    public static string Name = "";
    public static string Key = "";
}
```

**Creating Static NamespaceManager and MessagingFactory**

The next step is to create and initialize a NamespaceManager and MessagingFactory as static variables. This allows the objects to be reused and saves the overhead of recreating them.

```
class Program
{
    static NamespaceManager NamespaceMgr;
    static MessagingFactory Factory;
```

```
    static void Main(string[] args)
    {
        CreateManagerAndFactory();
    }

    static void CreateManagerAndFactory()
    {
        // Create a token provider with the relevant credentials.
        TokenProvider credentials =
            TokenProvider.CreateSharedSecretTokenProvider
            (AccountDetails.Name, AccountDetails.Key);

        // Create a URI for the serivce bus.
        Uri serviceBusUri = ServiceBusEnvironment.CreateServiceUri
            ("sb", AccountDetails.Namespace, string.Empty);

        // Create the NamespaceManager
        NamespaceMgr = new NamespaceManager(serviceBusUri, credentials);

        // Create the MessagingFactory
        Factory = MessagingFactory.Create(serviceBusUri, credentials);









        // Close the MessagingFactory.
        Factory.Close();

    }
}
```

**Creating a Topic and Adding Subscriptions**

When this is done a topic and some can be created in the service bus namespace. The following code will delete the topic, and then re-create it. As subscriptions belong to a topic any subscriptions we add to the topic will be deleted with that topic. Three subscriptions are then added to the topic without specifying any filters.

Please not that at the time of writing Azure customers are not billed for using brokered messaging services. When the billing model is in place repeatedly deleting and recreating messaging entities could lead to a recorded consumption of a large number of entity hours.

```
static void CreateTopicsAndSubscriptions()
{
    // If the topic exists, delete it.
    if (NamespaceMgr.TopicExists("ordertopic"))
    {
        NamespaceMgr.DeleteTopic("ordertopic");
    }

    // Create the topic.
    NamespaceMgr.CreateTopic("ordertopic");

    // Create three subscriptions in the topic.
    NamespaceMgr.CreateSubscription("ordertopic", "subscription1");
    NamespaceMgr.CreateSubscription("ordertopic", "subscription2");
    NamespaceMgr.CreateSubscription("ordertopic", "subscription3");
}
```

**Implementing a SendOrder Method**

As we will be sending several messages to the topic a static method is added to send message using a TopicClient.

```
static void SendOrder(TopicClient topicClient, Order order)
{
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.Write("Sending {0}...", order.Name);

    // Create a message from the order.
    BrokeredMessage orderMsg = new BrokeredMessage(order);

    // Send the message.
    topicClient.Send(orderMsg);

    Console.WriteLine("Done!");
}
```

**Receiving Messages form Subscriptions**

As several subscriptions have been added to the topic it makes sense to add method to loop through all the subscriptions on a topic and receive all the messages from each subscription. As the messages will have already been sent to the topic we can use a short timeout for receiving them from the subscriptions.

```
private static void ReceiveFromSubscriptions (string topicPath)
{
    Console.ForegroundColor = ConsoleColor.Magenta;
    Console.WriteLine("Receiving from topic {0} subscriptions.", topicPath);

    // Loop through the subscriptions in a topic.
    foreach (SubscriptionDescription subDescription in
        NamespaceMgr.GetSubscriptions(topicPath))
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("  Receiving from subscription {0}...", subDescription.Name);

        // Create a SubscriptionClient
        SubscriptionClient subClient =
            Factory.CreateSubscriptionClient(topicPath, subDescription.Name);

        Console.ForegroundColor = ConsoleColor.Green;


        // Receive all the massages form the subscription.
        while (true)
        {
            // Receive any message with a one second timeout.
            BrokeredMessage msg = subClient.Receive(TimeSpan.FromSeconds(1));
            if (msg != null)
            {
                // Deserialize the message body to an order.
                Order order = msg.GetBody<Order>();
                Console.WriteLine("    Received {0}", order.Name);

                // Mark the message as complete.
                msg.Complete();
            }
            else
            {
                break;
            }
        }

        // Close the SubscriptionClient.
        subClient.Close();
    }
    Console.ResetColor();
}
```

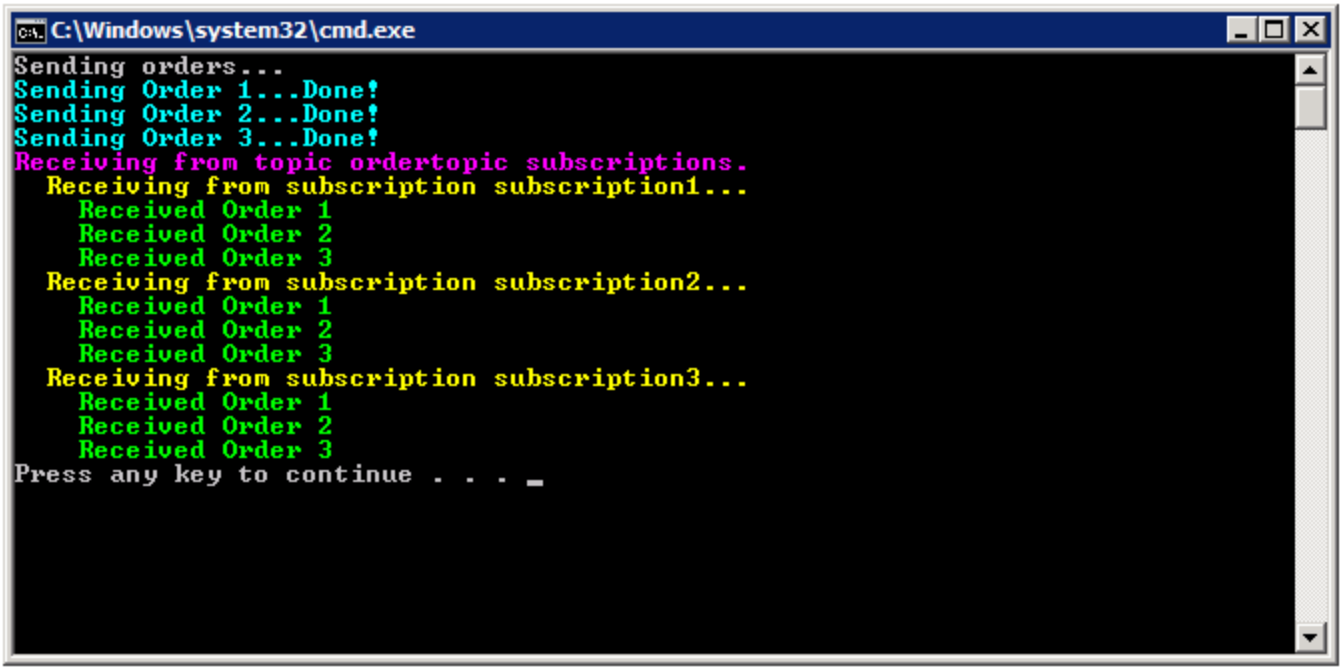
**Putting it All Together**

We can now add code to the main method to create the topic and subscriptions, send three order messages to the topic, and then receive messages from all the subscriptions.

```
static void Main(string[] args)
{
    CreateManagerAndFactory();

    CreateTopicsAndSubscriptions();

    // Create a TopicClient for ordertopic.
    TopicClient orderTopicclient = Factory.CreateTopicClient("ordertopic");

    // Send three orders.
    Console.WriteLine("Sending orders...");
    SendOrder(orderTopicclient, new Order() { Name = "Order 1" });
    SendOrder(orderTopicclient, new Order() { Name = "Order 2" });
    SendOrder(orderTopicclient, new Order() { Name = "Order 3" });

    // Close the TopicClient.
    orderTopicclient.Close();

    // Receive all messages from the ordertopic subscriptions.
    ReceiveFromSubscriptions("ordertopic");

    // Close the MessagingFactory.
    Factory.Close();

}
```

**Testing the Application**

When the application is tested the following output is displayed in the console.




A copy of each of the three order messages sent to the topic was received by the three subscriptions.

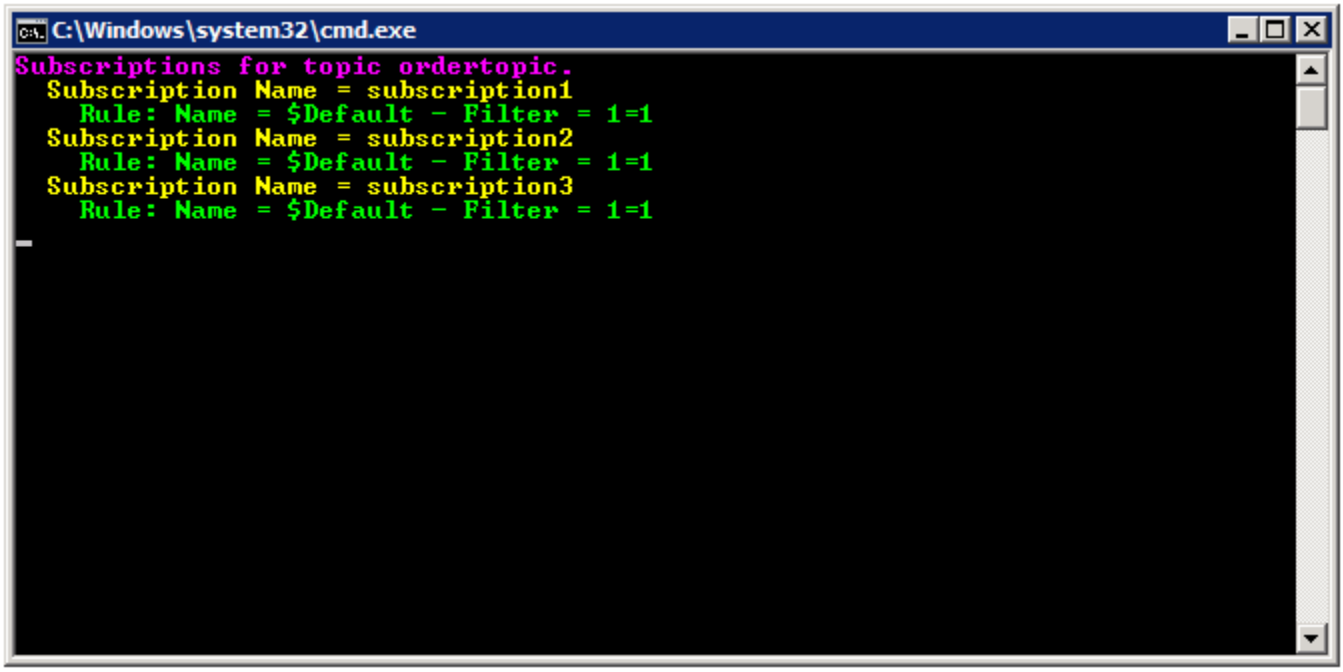
**Exploring Default Filters**

When the subscriptions were created in the, no filter was defined for them, and they subscribe to all messages that are sent to the topic. To understand why this happens we need to dig a little deeper into the rules on those subscriptions. The following method will loop through the subscriptions in a topic and display any rules that belong to the subscription.

```
private static void OutputSubscriptionsAndRules(string topicPath)
{
    Console.ForegroundColor = ConsoleColor.Magenta;
    Console.WriteLine("Subscriptions for topic {0}.", topicPath);

    // Loop through the subscriptions in a topic.
    foreach (SubscriptionDescription subDescription in
        NamespaceMgr.GetSubscriptions(topicPath))
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("  Subscription Name = {0}", subDescription.Name);
        Console.ForegroundColor = ConsoleColor.Green;

        // Loop through the rules in each subscription.
        foreach (RuleDescription rule in NamespaceMgr.GetRules(topicPath,
            subDescription.Name))
        {
            // Cast the rule to a SqlFilter and display the name and expression.
            Console.WriteLine("    Rule: Name = {0} - SqlExpression = {1}", rule.Name,
                (rule.Filter as SqlFilter).SqlExpression);
        }
    }
```

```
}
```

When this code is executed the results are as follows.




This shows that each subscription has been created with a default rule with a SQL filter expression of "1 =1". The filters are actually instances of the TrueFilter class, which derives from SqlFilter and sets the SQL expression to "1=1". The FalseFilter class also derives from SqlFilter and sets the SQL expression to "1=0". These filters can be used to cause a subscription to subscribe to all messages on the topic, or no messages.

### *Filtering Messages*

So far we have only looked at using topics and subscriptions to broadcast a message to all the subscriptions on a topic. In the store checkout scenario mentioned earlier there was a line of business application that processed orders from customers who had a loyalty card with the store chain. As this application is only interested in processing orders from customers who have a loyalty card it would make sense to have some kind of rule that only routed messages to that application if the order was from a customer with a card.

Routing messages in this way is known as content-based routing.

The Enterprise Integration Patterns website provides a description of the Content-Based Router pattern here.

In order to implement this pattern using topics and subscriptions we need to promote message properties and define rules using filter expressions.

### Promoting Message Properties

In this scenario a simple data contract is used to represent an order. As a reminder the code for this is shown below.

```
[DataContract]
class Order
{
    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public DateTime OrderDate { get; set; }

    [DataMember]
    public int Items { get; set; }

    [DataMember]
    public double Value { get; set; }

    [DataMember]
    public string Priority { get; set; }

    [DataMember]
    public string Region { get; set; }

    [DataMember]
    public bool HasLoyltyCard { get; set; }
}
```

When a message is created using an instance of this data contract the order details are serialized into the body of the message. When routing messages it would not be efficient to parse the body of the message when deciding which subscriptions to route the messages to. The message could be large and contain a lot of fields, the message could be encrypted, or the message could just contain a binary stream of data.

When evaluating the rules that will decide where to route a message the names values in the Properties dictionary collection or the CorrelationId property of the message are used. If we want to route messages based on any of their content, that information needs to be added to the Properties collection in the message. This is known as property promotion, and is a familiar concept for BizTalk developers.

The Properties property of the BrokeredMessage class is a string-object dictionary collection that can be used to add values from the message, or from the sending application, that are evaluated in the filter expressions in the subscriptions when a message is enqueued on a topic. This can be confusing when

talking about message properties and the message properties collection; I will try and use properties collection to refer to the string-object dictionary collection.

```
public IDictionary<string, object> Properties
```

String-object property pairs are added to this collection as follows.

```
// Create a message from the order.
BrokeredMessage orderMsg = new BrokeredMessage(order);

// Add string-object property pairs to the collection.
orderMsg.Properties.Add("region", "USA");
orderMsg.Properties.Add("value", 1.99);
```

Subscriptions can then be added to topics with a specified filter as follows.

```
NamespaceMgr.CreateSubscription ("topic", "usaSub", new SqlFilter("region = 'USA'"));
```

Case sensitivity is used in string comparisons for the values in the property collection, but not for the names.

```
// This subscription will subscribe the order message.
NamespaceMgr.CreateSubscription ("topic", "usaSub", new SqlFilter("REGION = 'USA'"));

// This subscription will not.
NamespaceMgr.CreateSubscription ("topic", "usaSub", new SqlFilter("region = 'usa'"));
```

The SendOrder method in the example used in the previous walkthrough can be modified to promote the properties in the order data contract that will be used to route the messages to the appropriate subscriptions.

```
static void SendOrder(TopicClient topicClient, Order order)
{
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.Write("Sending {0}...", order.Name);

    // Create a message from the order.
    BrokeredMessage orderMsg = new BrokeredMessage(order);

    //// Promote properties.
    orderMsg.Properties.Add("loyalty", order.HasLoyltyCard);
    orderMsg.Properties.Add("items", order.Items);
    orderMsg.Properties.Add("value", order.Value);
    orderMsg.Properties.Add("region", order.Region);

    // Send the message.
    topicClient.Send(orderMsg);
    Console.WriteLine("Done!");
}
```

## Defining Rules

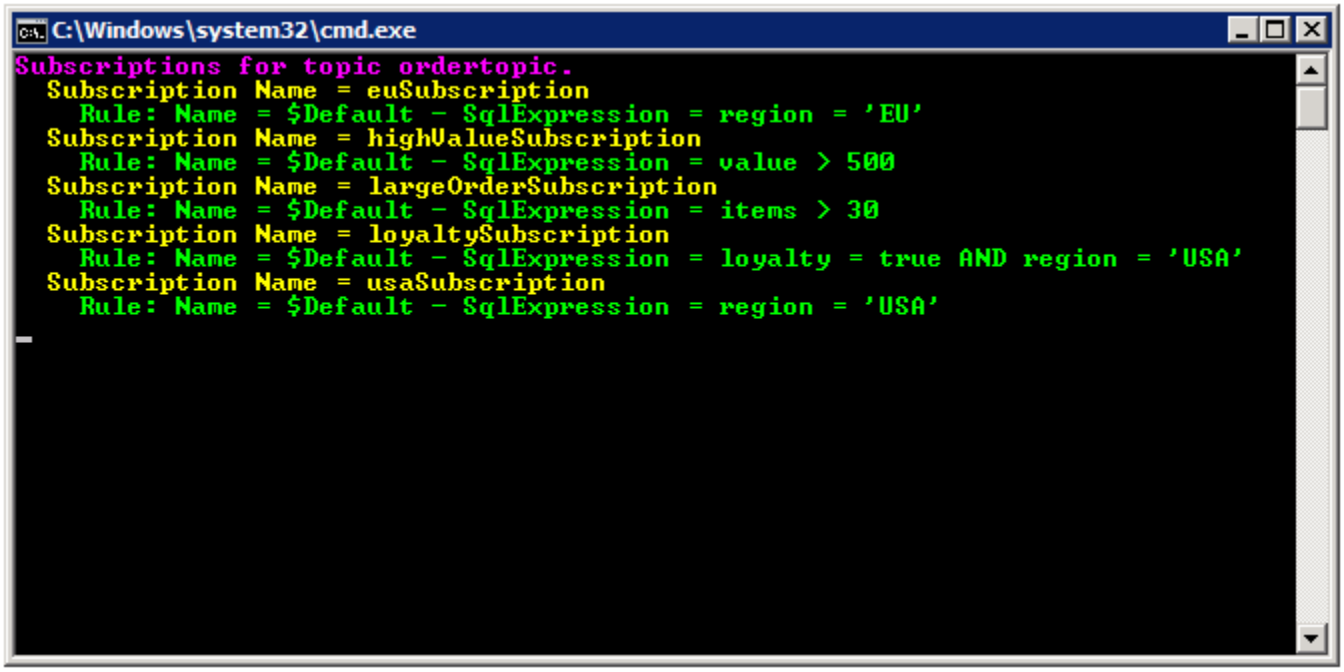
In order for messages sent to topics to be routed to the appropriate subscriptions rules must be added to the subscriptions using filter expressions. If the rules for the filter expression evaluate as true a copy of the message will be routed to the subscription. In the previous example it appeared that no rules were defined for the subscriptions added to the topic, but what was actually happening was default rules were created for those subscriptions.

**Filter Expressions**

If a subscription is to subscribe to messages based on values in the message properties a rule containing a filter expression must be specified for the subscription. This can be done when the subscriptions are added to the topic on the following way. At present the SqlFilter class is used to define filter expressions using a SQL92 like expression.

```
// Create three subscriptions in the topic.
//NamespaceMgr.CreateSubscription("ordertopic", "subscription1");
//NamespaceMgr.CreateSubscription("ordertopic", "subscription2");
//NamespaceMgr.CreateSubscription("ordertopic", "subscription3");

// Create subscripions using values set in the message properties collection.
NamespaceMgr.CreateSubscription ("ordertopic", "usaSubscription",
    new SqlFilter("region = 'USA'"));

NamespaceMgr.CreateSubscription  ("ordertopic", "euSubscription",
    new SqlFilter("region = 'EU'"));

NamespaceMgr.CreateSubscription  ("ordertopic", "largeOrderSubscription",
    new SqlFilter("items > 30"));

NamespaceMgr.CreateSubscription  ("ordertopic", "highValueSubscription",
    new SqlFilter("value > 500"));

NamespaceMgr.CreateSubscription  ("ordertopic", "loyaltySubscription",
    new SqlFilter("loyalty = true AND region = 'USA'"));
```

When the OutputSubscriptionsAndRules method is used to display the subscriptions for the ordertopic topic the output is as follows.



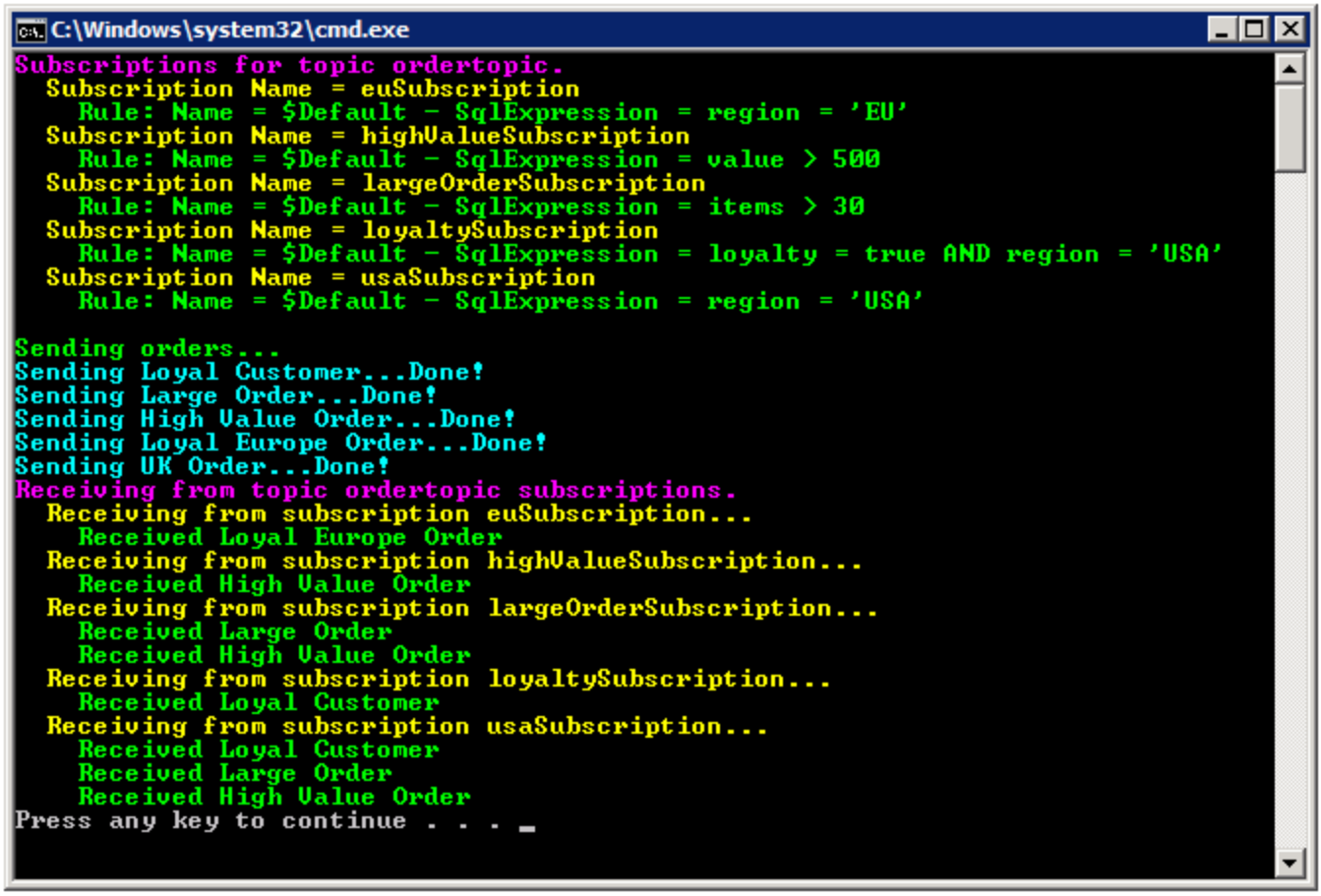

By specifying a SqlFilter when adding the subscription the default rule will be created using that filter expression, instead of the TrueFilter.

With these subscriptions created we can now send a set of messages to the topic and examine the results. The following changes  will call the SendOrder method to send five orders with different property values to the topic. The SendOrder method will promote the properties using the code shown earlier. The ReceiveFromSubscriptions method will then receive all messages from all subscriptions in the topic.

```
// Send three orders.
Console.WriteLine("Sending orders...");
//SendOrder(orderTopicclient, new Order() { Name = "Order 1" });
//SendOrder(orderTopicclient, new Order() { Name = "Order 2" });
//SendOrder(orderTopicclient, new Order() { Name = "Order 3" });

// Send five orders with different properties.
SendOrder(orderTopicclient, new Order() { Name = "Loyal Customer", Value = 19.99,
    Region = "USA", Items = 1, HasLoyltyCard = true });

SendOrder(orderTopicclient, new Order() { Name = "Large Order", Value = 49.99,
    Region = "USA", Items = 50, HasLoyltyCard = false });

SendOrder(orderTopicclient, new Order() { Name = "High Value Order", Value = 749.45,
    Region = "USA", Items = 45, HasLoyltyCard = false });

SendOrder(orderTopicclient, new Order() { Name = "Loyal Europe Order", Value = 49.45,
    Region = "EU", Items = 3, HasLoyltyCard = true });

SendOrder(orderTopicclient, new Order() { Name = "UK Order", Value = 49.45,
    Region = "UK", Items = 3, HasLoyltyCard = false });

// Close the TopicClient.
orderTopicclient.Close();

// Receive all messages from the ordertopic subscriptions.
ReceiveFromSubscriptions("ordertopic");
```

The SendOrder method will promote the properties using the code shown earlier. The ReceiveFromSubscriptions method will then receive all messages from all subscriptions in the topic.

When this code is run the output is as follows.




The results show that the order messages are successfully routed to the appropriate subscribers based on the values in their promoted properties. A summary of those values is provided below.

| Name | Value | Region | Items | HasLoyaltyCard |
|---|---|---|---|---|
| Loyal Customer | 19.99 | USA | 1 | true |
| Large Order | 49.99 | USA | 50 | false |
| High Value Order | 749.45 | USA | 45 | false |
| Loyal Europe Order | 49.45 | EU | 3 | true |
| UK Order | 49.45 | UK | 3 | false |

But what happened to the UK order? The order was enqueued to the topic bit the properties of that order did not match any of the filter subscriptions, so it was not subscribed to by any of the subscriptions.

### Publish-Subscribe Model in Topics and Subscriptions

The previous examples highlighted the golden rules about publish-subscribe message channels.

- When a message is published a copy of the message is routed to all matching subscribers.

The following rules about topics and subscriptions also hold true.

- If is only one subscriber then the publish-subscribe channel acts like a point-to-point channel.
- If there are multiple subscribers they will all receive messages.
- If there are no subscribers the message will be ignored.

If you are new to the publish-subscribe model you will probably find this fairly intuitive. If you are a seasoned BizTalk developer and an expert in publish-subscribe messaging you want to question this. In BizTalk Server if a message is published and there are no subscribers the message will be suspended in the message box database and errors published in the event log, “The published message could not be routed because no subscribers were found”, or “published not subscribed” as BizTalkers call it. BizTalk treats all messages as potentially critical business data, even if nothing subscribes to the message. In some scenarios these messages should be suspended or deadlettered, in others they can be ignored. As present there is no way to deadletter these “published not subscribed” messages in AppFabric topics.

### Correlation Filters

It is also possible to use a correlation filter to subscribe to messages based on the value of the CorrelationId of the message. The BrokeredMessage class exposes the following property.

```
public string CorrelationId
```

The following code will create a subscription for UK orders using the CorrelationFilter class and correlating on the value “UK”.

```
NamespaceMgr.CreateSubscription  ("ordertopic", "highValueSubscription",
    new SqlFilter("value > 500"));

NamespaceMgr.CreateSubscription  ("ordertopic", "loyaltySubscription",
    new SqlFilter("loyalty = true AND region = 'USA'"));

NamespaceMgr.CreateSubscription("ordertopic", "ukSubscription",
    new CorrelationFilter("UK"));
```

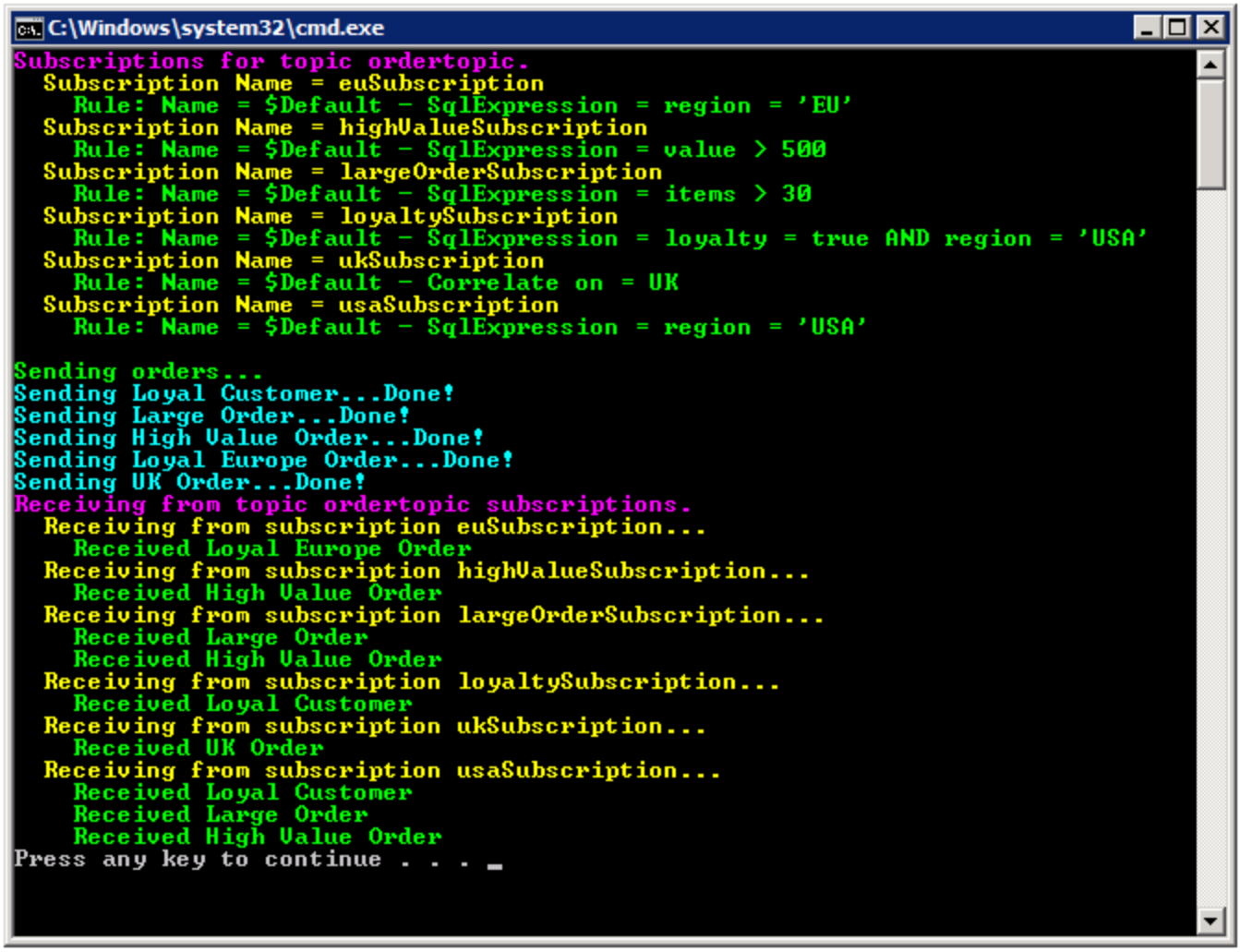
In order to examine the filters used in subscriptions the code in the OutputSubscriptionsAndRules method will need to be changed to take account of different filter types.

```
private static void OutputSubscriptionsAndRules(string topicPath)
{
    Console.ForegroundColor = ConsoleColor.Magenta;
    Console.WriteLine("Subscriptions for topic {0}.", topicPath);

    // Loop through the subscriptions in a topic.
    foreach (SubscriptionDescription subDescription in
        NamespaceMgr.GetSubscriptions(topicPath))
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("  Subscription Name = {0}", subDescription.Name);
        Console.ForegroundColor = ConsoleColor.Green;

        // Loop through the rules in each subscription.
        foreach (RuleDescription rule in NamespaceMgr.GetRules(topicPath,
            subDescription.Name))
        {
            if (rule.Filter is SqlFilter)
            {
                // Cast the rule to a SqlFilter and display the name and expression.
                Console.WriteLine("    Rule: Name = {0} - SqlExpression = {1}",
                    rule.Name, (rule.Filter as SqlFilter).SqlExpression);
            }
            if (rule.Filter is CorrelationFilter)
            {
                // Cast the rule to a SqlFilter and display the name and expression.
                Console.WriteLine("    Rule: Name = {0} - Correlate on = {1}", rule.Name,
                    (rule.Filter as CorrelationFilter).CorrelationId);
            }
        }
    }
}
```

In order for the message to be subscribed to by a correlation filter the CorrelationId property is set to the region for the order.

```
static void SendOrder(TopicClient topicClient, Order order)
{
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.Write("Sending {0}...", order.Name);

    // Create a message from the order.
    BrokeredMessage orderMsg = new BrokeredMessage(order);

    // Promote properties.
    orderMsg.Properties.Add("loyalty", order.HasLoyltyCard);
    orderMsg.Properties.Add("items", order.Items);
    orderMsg.Properties.Add("value", order.Value);
    orderMsg.Properties.Add("region", order.Region);

    // Set the CorrelationId to the region.
    orderMsg.CorrelationId = order.Region;

    // Send the message.
    topicClient.Send(orderMsg);
    Console.WriteLine("Done!");
}
```

When the code is executed the results are as follows.




We can see that the filter on the ukSubscription is correlating on the value of UK, and that the UK order has been correctly routed to the ukSubscription.

Correlation filters have the advantage in that they are easy to create, have a lower message header overhead, and are evaluated quicker in the brokered messaging service. There are some limitations in that only string equality comparisons can be used, so they cannot be used for numeric comparisons.