Microsoft Azure Storage Essentials

Harness the power of Microsoft Azure services to build efficient cloud solutions

Chukri Soueidi
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'Working with Queues'
- A synopsis of the book’s content
- More information on Microsoft Azure Storage Essentials
About the Author

Chukri Soueidi is a software developer based in Beirut, Lebanon. With an experience that spans more than 8 years, he specializes in web development and Microsoft technologies. He currently works at the American University of Beirut as a software engineer and systems analyst, developing new software solutions for the university and its medical center.

He was awarded the Microsoft Most Valued Professional (MVP) award for 2 years 2014 and 2015 for his contributions to the technical communities of Visual C# and .NET. The MVP award is an annual award that recognizes exceptional technology community leaders worldwide, who actively share their high-quality and real-world expertise with other communities. With fewer than 5,000 awardees worldwide, Microsoft MVPs represent a highly selected group of experts.

He is heavily involved in local developer communities in Beirut, coaching on the latest technologies, and he is also a regular speaker at the major evangelism events held by Microsoft, Lebanon. He conducted several workshops for university students and professionals on topics of software development and gaming technologies. Chukri is also a technical trainer and has been working as an instructor at the regional learning centers.

Away from his laptop, Chukri is an avid marathon runner. He enjoys exploring new places and cooking with his beloved ones and friends. You can always reach him at chukrisoueidi@outlook.com.
Microsoft Azure Storage provides a set of services that can be used by applications for data management and storage. By providing scalable, durable, and highly available services, this platform is remarkable for its design and rich features that can help you achieve all types of data management tasks, from the simplest to the most complex. For developers, it provides a rapid development environment that supports small to large-scale applications, and enables new scenarios on the cloud, server, mobile, and web. For IT professionals, it reduces the complexity of managing, protecting, and replicating data.

*Microsoft Azure Storage Essentials* is a jump-start book for developing applications using the different storage services. This book will teach you about the different characteristics of each service and how to utilize them in applications and perform basic programming operations on them.

The storage services are exposed through REST APIs which make them accessible to any HTTP enabled application; these APIs allow for platform specific client libraries to be implemented on top of them. This book will be focusing on Windows Azure Client Library for .NET using the C# language. We will deal with storing and retrieving unstructured data with blobs, and then move on to tables to insert and update entities in a structured NoSQL fashion. Then, we will explore Queues as a reliable messaging service; after that, we will show file storage that allows migrating legacy applications' data to the cloud; to end with, we will learn about transient fault handling and service monitoring.

This book is an introduction to Microsoft Azure Storage and also to developing applications that make use of these data management services.
What this book covers

Chapter 1, *An Introduction to Microsoft Azure Storage Services*, introduces the Azure Storage service and all the options associated with storage accounts.

Chapter 2, *Developing Against Storage*, provides an overview of how to manage the services and utilize them programmatically using many familiar development platforms.

Chapter 3, *Working with Blobs*, examines the blob storage service and the two different types of blobs. It then, moves on to how to create and manage blobs.

Chapter 4, *Working with Tables*, looks at the Table storage service as an alternative database solution that is designed for the purpose of storing massive amounts of flat data.

Chapter 5, *Designing Scalable and Performant Tables*, provides a deeper dive into tables, in terms of design patterns and tackling the need for speedy read and write operations.

Chapter 6, *Working with Queues*, introduces the Queue storage service as a messaging solution for communications between different apps.

Chapter 7, *Working with Azure File Service*, allows you to create mounted shares on virtual machines that can accessed using the SMB protocol or simple REST APIs.

Chapter 8, *Transient Fault Handling and Analytics*, deals with transient fault handling and retry policies, and also examines Storage Analytics that comes with storage services.
So far, we have seen two services by Azure Storage: blobs and tables. While the previous services provide persistent storage options, this chapter will introduce you to a different non-persistent storage service, queue storage. Azure Queues provide a reliable system to store messages that need to be consumed afterwards by other tiers. In this chapter, we will explore the characteristics of queues and the way messages are inserted, and then consumed.

The need for queues

Queues are data structures that enable you to queue and de-queue messages into them for later retrieval; they allow you to create a backlog of work. A queue-centric pattern in applications can be simply described as follows: an application uses a queue to keep messages or chunks of data in a quick and ordered way; on the other hand a backend tier(s) of the same or different application is following up on these messages and processing them gradually.

Queues are used in a wide range of applications and they prove to be very effective in addressing many business requirements, such as increasing availability and reliability of systems; reducing latency; load-leveling the workload and deferring processing at peak times; avoiding bottlenecks of IO resources; and distributing communication between heterogeneous platforms.

Increasing the availability and reliability

Applications that are dependent on other services, such as databases or public APIs, need a way to avoid any service interruptions of these external components and not to halt waiting for failed requests. By using Queues, you can loosely couple your components, allowing your app to save its operations without being directly affected by database server availability for instance.
Reducing latency and avoiding IO bottlenecks

Queues can be useful any time you are doing time-consuming work. For example, say you want to create different thumbnails and sizes for any newly uploaded image on your website. It might be unpleasant to keep the user waiting for your background image processing job to run and save the different files on your server. Using queues, you can put this operation in a queue message and send a success "upload" message to your user, and then a worker service will continue the image processing job in the background.

Load leveling

Many applications pass through peak times during which the amount and volume of requests exceeds the capacity that can be handled by its infrastructure. Usually, you can resolve this in two ways: either you purchase additional resources to make sure your infrastructure can handle the worst-case scenarios; or you can simply use queues to keep a backlog of work that needs to be processed later, allowing your application to stay available even during peak times.

Passing messages between Azure web roles and worker roles

Another offering by Azure is Cloud Services; it allows you to deploy applications that can be scaled without the overhead of administration and load balancing. These applications will run on Azure VMs and can be of two types: web roles (web apps) and worker roles (for background processing). In many cases these two roles need to communicate and pass messages between each other and a perfect solution for that would be to use Azure Queues.

The Queue storage structure

Azure Queues offers a simple REST-based service for storing messages that can be accessed and consumed from anywhere via authorized HTTP or HTTPS requests, providing a reliable messaging system for multiple dependent services. It shares the same programming model with the previously discussed tables and blobs, creating a uniform experience for developers.
The service contains the following components:

- **Storage Account**: It provides the namespace and manages access and authorization to all associated queues.
- **Queue**: It stores an unlimited number of messages. Cannot exceed the 500 TB size limit.
- **Message**: It contains a payload of UTF-8 encoded text data. A message can only persist up to seven days in a queue and has a maximum size of 64 KB.

We have seen in the previous chapter how entities are partitioned in tables, by the partition key, to meet the traffic needs of tables. However, messages in a single queue are grouped into a single partition determined by the queue name and are always served by a single server. Thus, for load balancing, you might consider using multiple queues for highly demanding needs.

Messages are usually stored in JSON format and should always be XML-safe or Base64 encoded. You can use the blob storage service to store large messages while keeping a reference in the queue message.

For storing files in a message, you can save the binary file in a blob container, then you can insert its URL in the message.

**Addressing**

Queues are addressable using the following URL pattern:

http://<account-name>.queue.core.windows.net/<queue-name>

The `<account-name>` placeholder stands for your storage account name and the `<queue-name>` placeholder stands on the name of your queue.

For development using the storage emulator, the URL pattern differs and becomes:

http://127.0.0.1:10003/devstorageaccount1/<queue-name>

The `127.0.0.1` IP is the local loopback address for your development machine, and the `devstorageaccount1` is a hardcoded name set by the emulator as the default storage account name.
Naming
When creating queues, there are some naming rules to follow:

- **Queue**: A queue name must be unique within its storage account, and range from 3 to 63 characters long. It can only contain alphanumeric characters or non-consecutive dash (-) characters and cannot begin or end with a dash, nor start with a number. All letters in a queue name must be lowercase.

- **Metadata**: A queue can have its own metadata stored with it as name-value properties. Property names are case-insensitive and must adhere to the C# identifiers naming rules; they must also be unique per queue.

Service Bus Queues
The Azure platform currently offers two different queuing services: Azure Queues and Service Bus Queues. Service Bus is a broader type of messaging service that provides a number of additional capabilities over the storage queues.

In summary, Service Bus provides four different communication mechanisms that are designed to integrate applications or hybrid application components. These entities are Queues, Topics, Relays, and Event Hubs. Service Bus Queues are brokers that store messages; Topics are queues that offer a pub/sub mechanism where multiple applications subscribe to a queue and filter their messages based on certain topics; Relays provide bidirectional direct communication between two separate applications that, for example, might be both residing on premises; Event Hub is an event processing service for large scale data intake.

Choosing between Service Bus Queues and Azure Queues depends on multiple factors that should be considered based on the individual requirements of your application. Service Bus Queues offers more features but with a higher overhead on developers. In this chapter, we will be only focusing on Azure Queues since it is part of the Azure Storage infrastructure.

For a full comparison between these two queue services, check the MSDN article *Azure Queues and Service Bus Queues - Compared and Contrasted*:
Using the queue storage from .NET

The Azure Storage Client Library for .NET has all the classes and methods needed to work with queues programmatically. After creating a storage account, and getting the appropriate access keys (both discussed in earlier chapters), we now have everything it takes to utilize the storage service using the .NET framework.

For the simplicity of our code samples, we will be using a C# Console Application template. To create an empty console application open Visual Studio and navigate to New Project | Templates | Visual C# | Console Application.

By default, the application will not include the storage client library, so we need to add it using the package manager by typing Install-Package WindowsAzure.Storage. This will download and install the library along with all dependencies required.

You can navigate through the library by expanding References in Solution Explorer in Visual Studio and right-click Microsoft.WindowsAzure.Storage and selecting View in Object Browser. This will show you the whole library classes and methods.

After getting the storage library, we need to create the connection string by providing the storage account name and access key. The connection string will look like this:

```xml
<appSettings>
    <add key="StorageConnectionString"
         value="DefaultEndpointsProtocol=https;
                 AccountName=account name;AccountKey=access key"/>
</appSettings>
```

In program.cs we need to reference the following libraries:

```csharp
using System.Configuration; // reference to System.Configuration should be added to the project
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Queue;
```

Alternatively, you can use the Azure Cloud Service project template to create services that run on Azure. A cloud service can contain web roles and worker roles. Typically, Azure Queues is filled by web roles (for example, ASP.NET application) and then processed by worker roles (Windows Service), which is always querying the queue for new messages to handle.
Creating our first queue

Using the connection string defined in the App.config file, we will proceed to create a CloudStorageAccount object. This object represents the storage account where the queue will be created along with all the security details and the endpoint definitions. After that we need to initialize a CloudQueueClient object, which represents a frontage for dealing with queues. Using that CloudQueueClient object we can get a queue reference and then call the queue.CreateIfNotExists() method that will actually call the table REST API under the hood to create a queue and return a True Boolean value. If there exists a queue with the same name or the queue name doesn't follow queue naming rules, the method will return false.

The following is the code to do that:

```csharp
string connectionString = ConfigurationManager.AppSettings["StorageConnectionString"];  
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(connectionString);  
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();  
CloudQueue queue = queueClient.GetQueueReference("firstqueue");  
bool b = queue.CreateIfNotExists();
```

Inserting a message in the queue

After creating a new queue, we will now insert our first message into it. In order to do that, we need to use the CloudQueueMessage class to create a message wrapper object then pass it to CloudQueue: AddMessage method:

```csharp
CloudQueueMessage message = new CloudQueueMessage("First Task !");  
queue.AddMessage(message);
```

The CloudQueueMessage class has three constructors that you can use in order to initialize it: the first one takes one parameter that should be a byte array; the second one accepts also a single parameter but of type string, like the one we used above; and the third one takes two string parameters that should specify a given message ID and pop receipt (we will talk about these in later sections).
The table below shows two additional parameters that can be passed to the `AddMessage` method while saving a message to the queue:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeToLive</td>
<td>TimeSpan</td>
<td>Specifies the amount of time the message will persist in the queue before it gets automatically purged from the server. The value cannot exceed the default and maximum message life (seven days).</td>
</tr>
<tr>
<td>initialVisibilityDelay</td>
<td>TimeSpan</td>
<td>Specifies the interval of time from insertion time during which the message will be invisible. If null then the message will be visible immediately.</td>
</tr>
</tbody>
</table>

Now to demonstrate these extra parameters, the code sample below will set the message's `timeToLive` to four days and `initialVisibilityDelay` to 30 seconds:

```
TimeSpan timeToLive = TimeSpan.FromDays(4); // cannot exceed 7 days
TimeSpan initialVisibilityDelay = TimeSpan.FromSeconds(30);
queue.AddMessage(message, timeToLive, initialVisibilityDelay);
```

**Handling large messages**

As we mentioned before, the maximum size of a queue message cannot exceed 64 KB. For instance, let's say your application has the scenario that we have seen before where a user uploads an image to your app that is following a queue centric pattern and you wish to create different thumbnails for it. At first glance, the messages (images in this case) appear to have sizes that exceed the maximum limit for messages. We can get around this problem by uploading the images to a blob container and only saving a reference for them in the queue messages. The background agent that is responsible for creating these thumbnails will then retrieve the message from queue, reference the blobs and then get the image.
The following is some sample code:

```csharp
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
    CloudBlobContainer blobContainer = blobClient.GetContainerReference("uploads");
    blobContainer.CreateIfNotExists();

    using (var file = new FileStream(@"D:\uploads\myphoto.jpg", FileMode.Open))
    {
        string blobId = Guid.NewGuid().ToString();
        CloudBlockBlob blob = blobContainer.GetBlockBlobReference(blobId);
        blob.UploadFromStream(file);
    }

    queue.AddMessage(new CloudQueueMessage(blobId));
```

The code we used to create the blob and blob container was fully covered in Chapter 3, Working with Blobs.

**Retrieving the next message**

To consume the stored messages, each time your app needs to de-queue the first next messages from the queue; to do this we will use the `queue.GetMessage()` method. Once a message is retrieved, it will be invisible for any other services or apps trying to get messages from the same queue. By default, you will only have 30 seconds to process the message and delete it, during which it will be invisible to others. This is known as visibility timeout.

After successfully processing the message, you will have to delete it from the queue using the `DeleteMessage` method. Following is an example of how to do that:

```csharp
CloudQueueMessage frontMessage = queue.GetMessage();

    // Do work here: process the message in less than 30 seconds, and then delete the message
    queue.DeleteMessage(frontMessage);
```
To control the visibility timeout, you can manually set the `visibilityTimeout` parameter while getting a message just like the following:

```csharp
CloudQueueMessage frontMessage = queue.GetMessage(TimeSpan.FromSeconds(90));
// This will keep the message invisible in the queue, giving you more time to process the request.
```

The queue service returns additional data along with each message that is filled inside the returned `CloudQueueMessage` object as properties. The table below lists the most important of these:

<table>
<thead>
<tr>
<th>Property</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertionTime</td>
<td>DateTimeOffset</td>
<td>The time that the message was added to the queue.</td>
</tr>
<tr>
<td>ExpirationTime</td>
<td>DateTimeOffset</td>
<td>The time that the message expires.</td>
</tr>
<tr>
<td>DequeueCount</td>
<td>Integer</td>
<td>The number of time the message has been retrieved. If the dequeuer count is high, this might be an indicator that something is wrong with the message and that it might be a poison message. (To be discussed later in this chapter).</td>
</tr>
<tr>
<td>TimeNextVisible</td>
<td>DateTimeOffset</td>
<td>The time the message will be visible in the queue again. This is calculated by adding the <code>visibilityTimeout</code> to time the message was retrieved.</td>
</tr>
<tr>
<td>PopReceipt</td>
<td>String</td>
<td>A token calculated by the service and given to each consumer retrieving the message. The owner of the last <code>PopReceipt</code> issued is the only one who is able to edit or delete the message.</td>
</tr>
</tbody>
</table>

The `PopReceipt` is used to prove the current ownership of the message. It will become invalid if the owner failed to process the message within the allowed time and other consumer de-queued the same message. The holder of the latest receipt is the only one eligible to edit or delete the message.
Getting more messages

For many reasons, you might prefer to sometimes retrieve a batch of messages in a single GET request. By using the GetMessages method you are able to retrieve a variable number of queue messages, up to 32. The return type of the method will be an IEnumerable<CloudQueueMessage> that can be iterated using a foreach loop:

```csharp
foreach (CloudQueueMessage mess in queue.GetMessages(10, TimeSpan.FromMinutes(2)))
{
    // Process the message
    string message = mess.AsString;
    queue.DeleteMessage(mess);
}
```

The dequeueCount and poison messages

From time to time, a message may not be processed successfully by your consuming service which will make it reappear after a certain period. This might be due to several reasons, such as an error in the stored data or maybe a programming defect in your consuming job. The dequeueCount property retrieved with every message can help us discover poison messages. It is good practice to check this number with every GET message request. If the dequeueCount is greater than three, you might want to flag it as a poisoned message and log it somewhere else such as a separate queue for later handling. The code below demonstrates this:

```csharp
CloudQueueMessage message = queue.GetMessage();
if (message.DequeueCount >= 3)
{
    // handle the poison the message
    // move it to a special queue for later management
}
```

Peeking messages

Queues provide a method to read the next message without de-queuing it. This might be useful if you want to sneak peek on the next message during development or debugging time. This can be done using the PeekMessage method:

```csharp
CloudQueueMessage peekedMessage = queue.PeekMessage();
```
Editing queue messages

Though it is not commonly used, it is possible to retrieve a message and update its content and then push it back to the queue. This might be useful in the case of a multi-step workflow, where you are saving the current step. Or you might need to change the visibility timeout for a message:

```csharp
message.SetMessageContent("Step:1");
queue.UpdateMessage(message,
    TimeSpan.FromSeconds(120),
    MessageUpdateFields.Content | MessageUpdateFields.Visibility);
```

The code above shows how to use the `SetMessageContent` to change the content of a retrieved message then uses the `UpdateMessage` to push the content changes and the visibility timeout update, specifying that both have been updated.

Setting metadata for the queue

Adding additional properties to your queue is possible by setting metadata name/value properties for it. You might, for instance, wish to add the date the queue was created, or the name of the developer or user that created this queue. The following code shows you how to do this:

```csharp
queue.Metadata.Add("CreationDate", DateTime.Now.ToString());
queue.Metadata.Add("Creator", "John Doe");
queue.SetMetadata();
```

To get all the `Metadata` properties, you need to explicitly call the `FetchAttributes` method in order for the `Metadata` property to be populated from the server. The data will be filled in a `Dictionary` object:

```csharp
queue.FetchAttributes();
IDictionary<string, string> metadata = queue.Metadata;
```

Getting the queue length and metadata

You can query a queue for the number of messages it contains. This is useful for monitoring your queues and the performance of the background services consuming it.
To do that you need to fetch the queue's attributes by calling the `FetchAttributes` method on the queue, which will fill the `ApproximateMessageCount`:

```csharp
queue.FetchAttributes();
int? messagecount = queue.ApproximateMessageCount;
```

### Deleting queues

You can delete a created queue by calling the `Delete` method of the queue. Needless to say, this will delete all the messages contained in it. The following is the simple command to do it:

```csharp
queue.Delete();
```

### Summary

In this chapter we covered the basics for the Azure Queue service, we learned how to create a queue and how to enqueue and de-queue it with messages as well as how to handle large messages by using a hybrid solution between the blob service and queue service. In the next chapter, we will dig into file storage, which offers shared storage for applications using the common standard **Server Message Block (SMB)** protocol.
Where to buy this book
You can buy Microsoft Azure Storage Essentials from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.