Foreword by Jeffrey Richter, Wintellect

Microsoft

Developing Cloud Applications with Windows Azure Storage

Professional

Paul Mehner

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2012948864
ISBN: 978-0-7356-6798-3

Printed and bound in the United States of America.

First Printing

Acquisitions and Developmental Editor: Devon Musgrave
Project Editor: Rosemary Caperton
Editorial Production: Online Training Solutions, Inc. (OTSI)
Copyeditor: Victoria Thulman
Indexer: Jan Bednarczuk
Cover: Twist Creative • Seattle

To my wife Shelley, who tolerated my many absences from family life while I worked on this book and trained and consulted on other projects. I love you! Thanks to my son Austin and my daughter Taryn for being the truly awesome kids that they are! I love you both, too! Thanks to our pet Rotsky (Rottweiler/Husky) Bella, our deceased black lab Codey (RIP), and our Bearded Dragon Uri, who made their own loving and memorable contributions.

Thanks to my friend and mentor Jeffrey Richter for the many hours of assistance that he gave me with this book, and for allowing me to use the source code from Wintellect's Windows Azure class as the basis for most of the code samples provided.

—Paul Mehner

Contents at a glance

Contents

Foreword

To my fellow Data Lover,

The most important part of any application is its data. Data is used for user accounts, orders, game scores, news items, status updates, documents, photos, music, images—the list goes on and on. It used to be that data was all about bytes stored on a hard disk and how quickly our applications could access these bytes. But for today's modern cloud-based applications, topics related to data now include:

- The geo-location of the data center storing the data, which impacts latency and geopolitical boundaries.
- Security and confidentiality of the data.
- Redundancy and high availability of the data.
- Performance and scalability when accessing the data.
- Optimistic concurrency patterns, transactions, and atomicity.
- Historical copies, or versioning of the data.
- Pricing (of course) related to all of the above.

Yes, data and everything associated with it has become a complex web of topics and issues that many applications must manage today. Fortunately, Microsoft has built a world-class, cloud-based data storage service that can easily be incorporated into many existing applications. This system addresses all of the issues I just mentioned and more. This book helps you understand what this service can do and how to use the service effectively. The book offers guidance, design patterns, and tips and tricks along the way.

And, dear reader, you are quite lucky to have Paul Mehner as the author of this book. I met Paul at a .NET user group meeting many years ago and was immediately impressed with him. I watched in admiration as he presented complex topics to the audience in such a way that they immediately grasped what he was saying. In fact, I was so impressed with Paul that I asked him to be a part of my company, Wintellect, and he has been working with us for many years now.

Furthermore, Paul has been working with Windows Azure long before it officially shipped. Through Wintellect, he teaches various Windows Azure topics (including storage) to Microsoft's own employees. And, he has also worked on many consulting engagements related to Windows Azure. This book is filled with insight from Paul's real-world experiences.

Today, just about everyone is interested in learning the best ways possible to manage their data, and this book is the best place to start on your journey.

— Jeffrey Richter (http://Wintellect.com/)

Introduction

Windows Azure storage provides independent data management services to your application, that is, data storage for any application, on any platform capable of making an HTTP request, written in any programming language, and deployed to the cloud or hosted in your own data center. Windows Azure storage provides a rich set of features that your applications can take advantage of to achieve operating characteristics that might otherwise be unobtainable because these characteristics were too complex, cumbersome, time-consuming, or expensive to implement. Not every application will require this set of features offered by the data management service; however, most will benefit from at least a few of them. It's hard to imagine an application that would not benefit from improved data reliability.

Developing Cloud Applications with Windows Azure Storage provides detailed information about the Windows Azure data management services platform. The book approaches the subject from the perspective of an open and RESTful data storage platform that can be used independent of any other Microsoft technology. The book focuses on the RESTful API of Windows Azure data management services to provide you with a much deeper understanding of how the storage platform works. Each REST example is also supplemented with an example that uses the Windows Azure client library (also referred to as Windows Azure storage library), which is available on many platforms, including the Microsoft .NET Framework. The Windows Azure client library eases some of the mundane and repetitive tasks such as attaching security and other custom HTTP headers to your storage requests. This gives you a much more complete, top-to-bottom understanding of the technology.

Who should read this book

This book will help existing Microsoft Visual Basic and Microsoft Visual C# developers understand the core concepts of Windows Azure data management services and related technologies. It is especially useful for programmers looking to manage database-hosted information in their new or existing .NET applications. Although most readers will have no prior experience with Windows Azure data management services, the book is also useful for those familiar with building applications against a relational database such as Microsoft SQL Server.

Assumptions

This book is written with the assumption that you have a minimal understanding of the HTTP protocol in addition to .NET development and object-oriented programming concepts for the Windows Azure client library code samples. Although the Windows Azure client library is available for many platforms and languages, this book includes examples in C# only. You should also have a basic understanding of database concepts, and perhaps some experience with relational database systems such as SQL Server.

Who should not read this book

Because the focus of this book is on software development on the Windows Azure platform, it is not intended for the information technology (IT) professional. It is also not intended for novice developers, because you will need intermediate software development experience.

Organization of this book

This book is divided into three parts:

- **Part I: Architecture and use** This part covers the architecture and use of Windows Azure data management services and how they are accessed.
- **Part II: Blobs, tables, and queues** This part covers specifics about blob, table, and queue storage and the scenarios that lend themselves to their use.
- **Part III: Analytics** This part focusses on how to collect and analyze Windows Azure data management service consumption logs and metrics for blobs, tables, and queues.

Conventions and features in this book

This book presents information using conventions designed to make the information readable and easy to follow:

- Boxed elements with labels such as “Note” provide additional information or alternative methods for completing a step successfully.

- A plus sign (+) between two key names means that you must press those keys at the same time. For example, “Press Alt+Tab” means that you hold down the Alt key while you press the Tab key.

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- One of the following: Windows 7, Windows 8, Windows Server 2003 with Service Pack 2, Windows Server 2003 R2, Windows Server 2008 with Service Pack 2, or Windows Server 2008 R2
- Microsoft Visual Studio 2012, any edition (multiple downloads may be required if using Express Edition products)
- SQL Server 2008 R2 Express Edition or later, with SQL Server Management Studio 2008 Express or later (included with Visual Studio; Express Editions require separate download)
- Windows Azure client library appropriate for your client application platform (the book assumes you are using the .NET Framework)
- Microsoft Internet Information Services (IIS) or IIS Express version 7.5 or newer.
- Computer that has a 1.6 gigahertz (GHz) or faster processor (2 GHz recommended)
- 1 gigabyte (GB), 32-bit; or 2 GB (64-bit) RAM (add 512 MB if running in a virtual machine or SQL Server Express Editions, more for advanced SQL Server editions)
- 3.5 GB of available hard disk space
- 5400 RPM hard disk drive
- DirectX 9 capable video card running at 1024 x 768 or higher resolution display
- DVD-ROM drive (if installing Visual Studio from DVD)
- Internet connection to download software or chapter examples

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2010 and SQL Server 2008 R2 products.

Code samples

Most of the chapters in this book include examples that let you interactively try out new material learned in the main text. All sample projects can be downloaded from the following page:

http://aka.ms/DevCloudApps/files

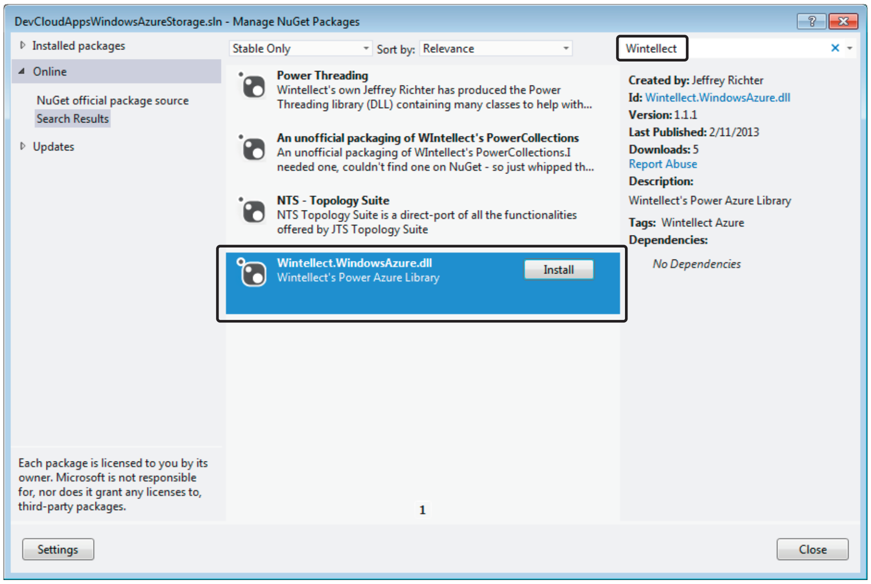
Follow the instructions to download the DevCloudApps_667983_CompanionContent.zip file.

Note In addition to the code samples, your system should have Visual Studio 2010 and SQL Server 2008 R2 (any edition) installed to support the Windows Azure storage emulator. Alternatively, you can run all examples against Windows Azure data management services without using the storage emulator.

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. Unzip the DevCloudApps_667983_CompanionContent.zip file that you downloaded from the book's website.
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.
3. Install the Wintellect Windows Azure Power Library NuGet package. You can do this from the NuGet Package Manager by searching for **Wintellect**, clicking Wintellect.WindowsAzure.dll (Wintellect Power Azure Library), and then clicking Install, as shown in the following figure.


DevCloudAppsWindowsAzureStorage.sln - Manage NuGet Packages

Installed packages
Online
NuGet official package source
Search Results
Updates

Stable Only | Sort by: Relevance | Wintellect

Power Threading
Wintellect's own Jeffrey Richter has produced the Power Threading library (DLL) containing many classes to help with...

An unofficial packaging of WIntellect's PowerCollections
An unofficial packaging of WIntellect's PowerCollections.I needed one, couldn't find one on NuGet - so just whipped th...

NTS - Topology Suite
NTS Topology Suite is a direct-port of all the functionalities offered by JTS Topology Suite

Wintellect.WindowsAzure.dll
Wintellect's Power Azure Library
Install

Created by: Jeffrey Richter
Id: Wintellect.WindowsAzure.dll
Version: 1.1.1
Last Published: 2/11/2013
Downloads: 5
Report Abuse
Description:
Wintellect's Power Azure Library
Tags: Wintellect Azure
Dependencies:
No Dependencies

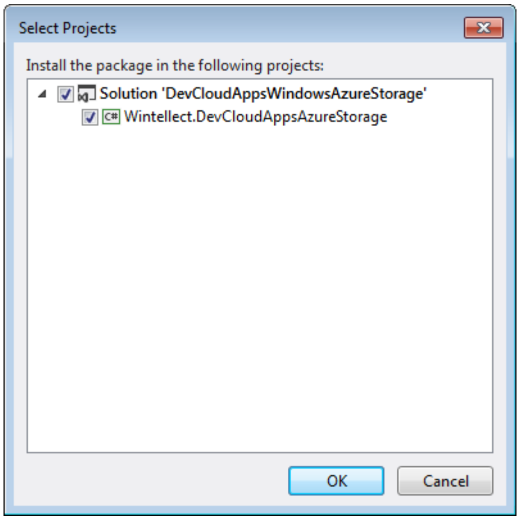
Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

1

Settings | Close

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the DevCloudApps_667983_CompanionContent.zip file.

4. You'll be prompted for the project that you want to add the Wintellect Power Azure Library to. Click the Wintellect.DevCloudAppsAzureStorage project and click OK, as shown in the following figure.


Select Projects
Install the package in the following projects:
Solution 'DevCloudAppsWindowsAzureStorage'
Wintellect.DevCloudAppsAzureStorage
OK
Cancel

5. After the library is installed, you may need to close the package manager by clicking Close.

Note You can also download the Wintellect Windows Azure Power Library package directly from the following URL: *https://nuget.org/packages/Wintellect.WindowsAzure.dll.*

Using the code samples

Locate the AzureSecrets.txt file in the root directory of the folder you expanded the zip file into. Modify this file with your own Windows Azure subscription, management certificate thumbprint, the name of your storage account, and its key.

```
# The projects in this solution require the use of your personal Azure account
# information which you supply below.
# You do not need to enter all the values; you only need to enter values for the
# projects that require the specific information.
# If you run a project that requires a value that you did not supply, an exception
# will be thrown and this file will open automatically in Notepad so that you can
# add the missing value. After adding the value, you must re-run the
# project so that it picks-up the new value.

ManagementSubscriptionId=
ManagementCertificateThumbprint=

# http://msdn.microsoft.com/en-us/library/windowsazure/gg432983.aspx
```

```
StorageAccountName=devstoreaccount1
StorageAccountKey=
Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tqeksoGMGw==
```

The unzipped contents will contain a Visual Studio 2012 solution file named Dev-CloudAppsWindowsAzureStorage.sln and a project folder. Load the DevCloudApps-WindowsAzureStorage.sln solution into Visual Studio 2012, and go to the Storage-Patterns.cs. file located in the Wintellect.DevCloudAppsAzureStorage project folder. All examples shown in this book are located in this file. The *Main()* routine in the following code calls five categories of examples: the code examples beginning with *AzureManagement* are for Chapter 3, "Windows Azure data storage"; *BlobPatterns* are for Chapter 5, "Blobs"; *TablePatterns* are for Chapter 6, "Tables"; *QueuePatterns* are for Chapter 7, "Queues"; and *AnalyticPatterns* are for Chapter 8, "Analytics, logging, and transaction metrics."

```
public static class AzureBookStoragePatterns {
   private static readonly Boolean c_SpawnStorageKiller = false;
   private static CloudStorageAccount m_account = null;
   private static CloudStorageAccount Account {
      [DebuggerStepThrough]
      get {
         if (m_account != null) return m_account;
         // Chose the default account to execute these demos against:
         m_account =
       //GetStorageAccount(FiddlerPrompt(StorageAccountType.DevStorage, true));
       //GetStorageAccount(FiddlerPrompt(StorageAccountType.DevStorageWithFiddler,
true));
       GetStorageAccount(FiddlerPrompt(StorageAccountType.AzureStorage, true));
         return m_account;
      }
   }

   private const String c_tableName = "Demo";

   static string s_storageAccountLabel1 =

"StagingAAAStagingAAAStagingAAAStagingAAAStagingAAAStagingAAAStagingZZZABCDE";
   static string s_storageAccountDesc1 = "Wintellect Demo Staging";
   static string s_storageAccountLocation1 = "West US";

   static string s_storageAccountName1 = "contosocohovinyard";
   static string s_storageAccountName2 = "contosotailspintoys";

   static string s_subscriptionId = AzureSecrets.ManagementSubscriptionId;
   static string s_certThumbprint = AzureSecrets.ManagementCertificateThumbprint;
   static string s_MsVersion = "2011-06-01";

   public static void Main() {
      Management.Rest();
   StorageAccountsEndpoints();

   AzureManagement.CreateAccount(s_storageAccountName1, s_storageAccountLabel1,
        s_storageAccountDesc1, null, s_storageAccountLocation1);
   AzureManagement.GetAccountProperties(s_storageAccountName1);
   AzureManagement.GetAccountKeys(s_storageAccountName1);
   AzureManagement.RegenerateAccountKeys(s_storageAccountName1, "Primary");
   AzureManagement.UpdateAccount(s_storageAccountName1, Convert.ToBase64String(
        Encoding.UTF8.GetBytes(s_storageAccountName2)), "Label Changed");
   AzureManagement.GetAccountProperties(s_storageAccountName1);
   AzureManagement.DeleteAccount(s_storageAccountName1);

   BlobPatterns.Basics(Account);
   BlobPatterns.RootContainer(Account);
   BlobPatterns.Attributes(Account);
   BlobPatterns.ConditionalOperations(Account);
   BlobPatterns.SignedAccessSignatures(Account);
   BlobPatterns.BlockBlobs(Account);
   BlobPatterns.PageBlobs(Account);
   BlobPatterns.Snapshots(Account);
   BlobPatterns.Leases(Account);
   BlobPatterns.DirectoryHierarchies(Account);
   BlobPatterns.Segmented(Account);

   TablePatterns.Basics(Account);
   TablePatterns.OptimisticConcurrency(Account);
   TablePatterns.LastUpdateWins(Account);
   TablePatterns.QueryFilterStrings(Account);
   TablePatterns.Segmented(Account);
   TablePatterns.MultipleKinds(Account);

   QueuePatterns.Basics(Account);
   QueuePatterns.Segmented(Account);

   AnalyticPatterns.AnalyticsLogs(Account);
   AnalyticPatterns.AnalyticsBlobMetrics(Account);
   AnalyticPatterns.AnalyticsCapacityBlob(Account);


   Console.WriteLine("===== finished =====");
   Console.ReadLine();
}

[DebuggerStepThrough]
private static StorageAccountType FiddlerPrompt(StorageAccountType accountType,
    Boolean prompt = true) {
   const MessageBoxOptions MB_TOPMOST = (MessageBoxOptions)0x00040000;
   if (prompt && (accountType == StorageAccountType.DevStorage)) {
      if (MessageBox.Show(
       "Drag Fiddler's Process Filter cursor on this window.\n" +
       "Are you using Fiddler?",
       "Wintellect's Windows Azure Data Storage Demo",
            MessageBoxButtons.YesNo, MessageBoxIcon.Information,
            MessageBoxDefaultButton.Button1, MB_TOPMOST) == DialogResult.Yes)
              accountType = StorageAccountType.DevStorageWithFiddler;
        }
        return accountType;
     }

     public enum StorageAccountType {
        AzureStorage,
        DevStorage,
        DevStorageWithFiddler
     }

     [DebuggerStepThrough]
     private static CloudStorageAccount GetStorageAccount(StorageAccountType
  accountType) {
         switch (accountType) {
             default:
             case StorageAccountType.DevStorage:
                 return CloudStorageAccount.DevelopmentStorageAccount;
             case StorageAccountType.DevStorageWithFiddler:
                 return CloudStorageAccount.Parse(
      "UseDevelopmentStorage=true;DevelopmentStorageProxyUri=http://ipv4.fiddler");
             case StorageAccountType.AzureStorage:
                 String accountName = AzureSecrets.StorageAccountName;
                 String accountKey = AzureSecrets.StorageAccountKey;
                 return new CloudStorageAccount(new StorageCredentials(accountName,
                      accountKey), false);
         }
     }

     private static void StorageAccountsEndpoints() {
         Console.Clear();

         Console.WriteLine("Azure storage endpoints:");
         String accountName = AzureSecrets.StorageAccountName;
         String accountKey = AzureSecrets.StorageAccountKey;
         CloudStorageAccount account = new CloudStorageAccount(
              new StorageCredentials(accountName, accountKey), true);
         Console.WriteLine("   BlobEndpoint:  " + account.BlobEndpoint);
         Console.WriteLine("   TableEndpoint: " + account.TableEndpoint);
         Console.WriteLine("   QueueEndpoint: " + account.QueueEndpoint);
         Console.WriteLine();

         Console.WriteLine("Storage emulator endpoints:");
         account = CloudStorageAccount.DevelopmentStorageAccount;
         Console.WriteLine("   BlobEndpoint:  " + account.BlobEndpoint);
         Console.WriteLine("   TableEndpoint: " + account.TableEndpoint);
         Console.WriteLine("   QueueEndpoint: " + account.QueueEndpoint);
         Console.WriteLine();
     }
```

To use the samples, set a breakpoint on the line AzureManagement *CreateAccount* method and run the sample with your debugger attached. When the debugger stops on the breakpoint, you can set the next line to execute by clicking Set Next Statement from the context menu (or the hotkey sequence Ctrl+Shift+F10) in Visual Studio to set the next statement to execute to the example you want to see, and then pressing F11 to step into the code.

Acknowledgments

I'd like to thank Jeffrey Richter for the sample code included in this book, and the Wintellect Windows Azure Power Library, which the code uses heavily. The code and the library are part of Wintellect's training class materials for Windows Azure. Jeffrey also reviewed several chapters and provided a lot of useful suggestions.

I'd also like to thank Victoria Thulman for her editorial review and suggestions, and Marc Young for his technical review. I could not have completed this endeavor without their care and dedication to this project. Scott Seely, Julie Lerman, and Sharyn Mehner also reviewed various chapters, and I would like to thank them as well.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://aka.ms/DevCloudApps/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the preceding addresses.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 2

Windows Azure data storage overview

In this chapter:

The objective of this chapter is to provide you with an infrastructural understanding of Windows Azure data storage to facilitate strategic and architectural decisions regarding its use in your applications. A primary byproduct of achieving this objective is establishment of the versatility of Windows Azure storage for use as an autonomous data management service independent of your application's platform or your deployment choices.

Feature-rich data storage for almost any application

Windows Azure storage provides independent data management services to your application, that is, data storage for *any* application, on *any* platform capable of making an HTTP request, written in *any* programming language, and deployed to the cloud or hosted in your own data center. Windows Azure storage provides a rich set of features that your applications can take advantage of to achieve operating characteristics that might otherwise be unobtainable because these characteristics were too complex, cumbersome, time-consuming, or expensive to implement. Not every application will require this set of features offered by the data management service; however, most will benefit from at least a few of them. It's hard to imagine an application that would not benefit from improved data reliability.

Before exploring what Windows Azure storage is, it is useful to clearly identify what it is not. Windows Azure storage is *not* a relational database. Relational databases eliminate (or significantly reduce) redundant data through a process called *normalization*. A single entity is stored in the database one time, where it is indexed by a unique identifier (called a *primary key*), which is used to make reference to that single entity wherever duplication of that information would occur in other entities. This is a powerful feature because data needs to be updated in only one place. Data stored in relational databases is also strongly typed and constrained to ensure that only strictly conformant data can be inserted into entities, reducing the collection of bogus data and the numerous bugs encountered when processing unanticipated data. If you need the features of a relational database, you likely need Windows Azure SQL Database. In Chapter 6, "Tables," you tackle the problem of how to structure your data for transactional storage in Windows Azure table storage. If you're a developer who has worked with Microsoft SQL Server (or other relational databases) most of your career, it's enough to simply state that Windows Azure table storage is significantly different from what you're likely familiar with.

The following list describes a few of the high-level Windows Azure storage features that your application might use and that you delve into in this chapter:

- **Independence and interoperability** Windows Azure storage is a service designed to be used independent of the applications that might utilize it. Because access to data management services is done with a simple REST API, Windows Azure storage can be used independent of your application's platform, programming language, or deployment model.
- **Geo-location** An application might use the geo-location feature of Windows Azure storage to store and retrieve data local to where it is being most used. Sometimes there are legal requirements for data storage as well. For example, some types of data must be stored in data centers that are physically located in a particular country.
- **Replication** Windows Azure storage maintains three replicas of your data at all times, providing a high degree of confidence in the safety of your data. Replication also improves the scalability of reading data, because data being read can be served from all replicas simultaneously.
- **Geo-replication** Windows Azure data is replicated from the data center that you select as your primary data to a secondary data center within the same geographical region but hundreds of miles away, further increasing your confidence in the safety of your data in the event of data center disaster.
- **The illusion of infinite storage** A sudden increase in storage capacity for your application or service can be accommodated without costly infrastructural changes to your own data center. Data storage capacity is allocated on demand in units of logical servers called *storage nodes*.
- **Automatic and on-demand scalability** A sudden increase in demand for your data will be automatically accommodated by Windows Azure storage without costly infrastructural changes to your own data center. The scalability capabilities of Windows Azure are truly astounding. Windows Azure will dedicate an entire storage node to a single partition of data if that's what is required to meet Microsoft's scalability objectives, which are introduced later in this chapter.

- **Data consistency** Windows Azure storage maintains a fully consistent data model. When a write operation completes, all subsequent read operations will return the updated value. This is different from other eventually consistent data models such as Amazon's S3, where data is eventually consistent. In the *eventually consistent* model, read operations that happen after a write operation may or may not return the updated value. All read operations will eventually return the updated value.
- **ISO 27001 certification** Windows Azure storage received ISO 27001 certification, which may be useful in helping your own application meet certification requirements for sensitive data. The Windows Azure core services of Cloud Services, data management services, Virtual Network, and Virtual Machines are all covered by this certification, which was conducted in 2011 by BSI Americas. A copy of this certification can be found online at the following URL:

  http://www.bsigroup.com/en/Assessment-and-certification-services/Client-directory/CertificateClient-Directory-Search-Results/?searchkey=companyXeqXmicrosoft
- **Pay as you go and pay only for what you need** No upfront expenditures to purchase equipment, space, and bandwidth are necessary to meet peak demand in your data center. You pay as you go and only for what you consume today. As your storage capacity and scalability needs change, so does your bill. This helps to keep your fixed costs in alignment with your revenues.

That's a very compelling list of features. The following sections explore what you need to know to use and have confidence in these capabilities.

Data storage abstractions

To provide context for some of the features of Windows Azure storage, you must first have an understanding of the types and characteristics of the data you will be storing. Windows Azure storage provides three major storage types. In Chapter 5, "Blobs," Chapter 6, and Chapter 7, "Queues," you will delve into these three types and learn when to use each one as well as the value that each brings to your applications. For the purposes of this chapter, a short and high-level description of the three major data storage types is adequate.

The abstractions of Windows Azure storage are considered forms of NoSQL data storage. The characteristics of NoSQL are that the database does not use the Structured Query Language (SQL), and the ACID-style transaction guarantees (*ACID* stands for atomicity, consistency, isolation, and durability) are either not supported or are supported only in a limited way. They may employ techniques such as *eventual consistency*, which is the notion that the data will eventually arrive at a consistent state across all storage nodes, but that brief periods of data inconsistency are anticipated and tolerable. NoSQL database management systems are useful when working with extremely large data stores because they improve performance and high availability of data. The emphasis in NoSQL database systems is on the ability to store and retrieve large quantities of data quickly rather than on the relationships between elements of data.

Blobs

The Windows Azure Blob service provides a simple RESTful API for storing and retrieving unstructured data (for example, documents, pictures, videos, and music), including a small amount of metadata about the contents. Blobs are organized into containers (which are very similar to directories), and you can store several hundred gigabytes (GB) in a single blob. Blobs and blob containers are accessible as unique URLs, and they can be created, read, updated, and deleted using simple HTTP verbs against their URLs. Blob containers may be set to allow public accessibility, allowing anonymous read-only access to the blob's contents. Blobs are ideal for the storage of web content. The primary purpose of publicly accessible read-only containers is to allow direct consumption of such content by unauthenticated web browsers. This is a convenient way of augmenting your on-premise web servers with content delivered from Windows Azure storage. This not only saves you the cost of making additional upfront capital investments, but it also allows you some flexibility in not having to purchase server and network capacity for planned peaks in advance of demand. In the real world, the demand may never materialize; or potentially even worse, the demand might materialize but your resources are inadequate to meet the demand and you miss opportunities. You'll learn more about the Blob service in Chapter 5.

Tables

The Windows Azure Table service provides a simple RESTful API for storing massive amounts of semi-structured data. Windows Azure tables are not at all like relational database tables, and you should avoid making such a correlation. Rows, which are called *entities*, consist of properties, which are like columns.

Different rows stored in the same table can often have different sets of properties. You can perform queries on individual tables just as you might do with a relational database, but queries cannot span tables. Of course, relationships are what a relational database is all about, and queries that span multiple tables are the bread and butter of what relational databases do, and how normalized data becomes denormalized for consumption by your applications and reports. You should therefore avoid making any kind of direct correlation between relational database tables and Windows Azure tables. You'll learn more about the Table service in Chapter 6.

Queues

Unlike blobs and tables, queues do not provide permanent storage of the messages they handle. In fact, messages are quietly deleted from queues if they are not processed within seven days. Windows Azure queues are a means of transmitting messages between different server roles and instances when your application runs on Windows Azure Cloud services. Like all of Windows Azure storage, queues are exposed through a RESTful API that could potentially be used by on-premise and non-Microsoft platforms. The queued message data management service provides durable fire-and-forget message storage and reliable delivery to your applications. You'll learn more about the Windows Azure Queue service in Chapter 7.

Windows Azure data centers

Windows Azure data centers are located in three major geographic regions: the United States, Europe, and Asia. Each geographic region is further subdivided into subregions. As depicted in the world map in Figure 2-1, the United States is divided into South Central, North Central, East, and West. Europe is divided into North and West, and Asia is divided into Southeast and East. You select the primary subregion where you want to store your data.

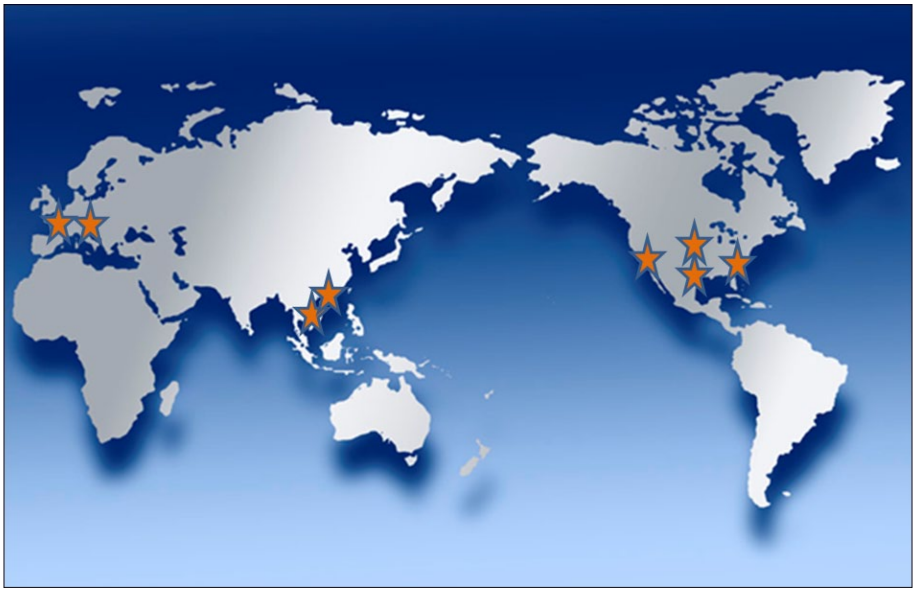


FIGURE 2-1 This world map shows where Windows Azure data centers are located.

Inside each data center are shipping containers packed full of servers with appropriate cooling and electrical supplies. You can take an interesting tour of one of the Microsoft data centers by visiting the following YouTube link, which was created by the Microsoft Cloud Infrastructure Team (MSGFST):

http://www.youtube.com/watch?v=hOxA1l1pQIw

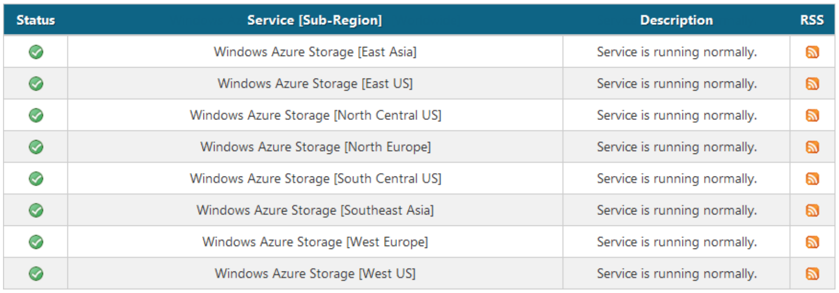
You can then continue the tour to facilitate a more detailed view of the assembly of the individual shipping containers that house the servers by going to this MSGFST-prepared YouTube link:

http://www.youtube.com/watch?v=nIliMskAHro

Your data in one data center will be replicated again to another data center within the same geographical region. For example, if your data is stored in the North Central US data center, that data will be replicated to the South Central US data center. Having three replicas of your data stored in two distinct data centers means that there will be at least six copies of your data, making the specter of cataclysmic data loss because of a natural disaster extremely unlikely.

Microsoft publishes a dashboard (shown in Figure 2-2) to allow convenient monitoring of the health of the data centers worldwide at the following URL:

http://www.windowsazure.com/en-us/support/service-dashboard/



Status	Service [Sub-Region]	Description	RSS
✓	Windows Azure Storage [East Asia]	Service is running normally.	▨
✓	Windows Azure Storage [East US]	Service is running normally.	▨
✓	Windows Azure Storage [North Central US]	Service is running normally.	▨
✓	Windows Azure Storage [North Europe]	Service is running normally.	▨
✓	Windows Azure Storage [South Central US]	Service is running normally.	▨
✓	Windows Azure Storage [Southeast Asia]	Service is running normally.	▨
✓	Windows Azure Storage [West Europe]	Service is running normally.	▨
✓	Windows Azure Storage [West US]	Service is running normally.	▨

FIGURE 2-2 Windows Azure Service dashboards enable monitoring of data management service health.

Each Windows Azure service has a convenient RSS feed for each geographical subregion. This allows you an opportunity to provide custom notification implementations or create automated adjustments to your application based on the health of any particular part of Windows Azure.

Storage topology

As depicted in Figure 2-3, the Windows Azure storage architecture consists of a front-end Virtual IP and three basic layers: a front-end (FE) layer, a partition layer, and a Distributed File System (DFS) layer. In this chapter, you're going to drill down into these three layers to gain a clear understanding of and confidence in exactly how this architecture is used. Doing so will help you achieve the 99.9 percent level of reliability in your applications that Microsoft guarantees in its Service Level Agreement (SLA).

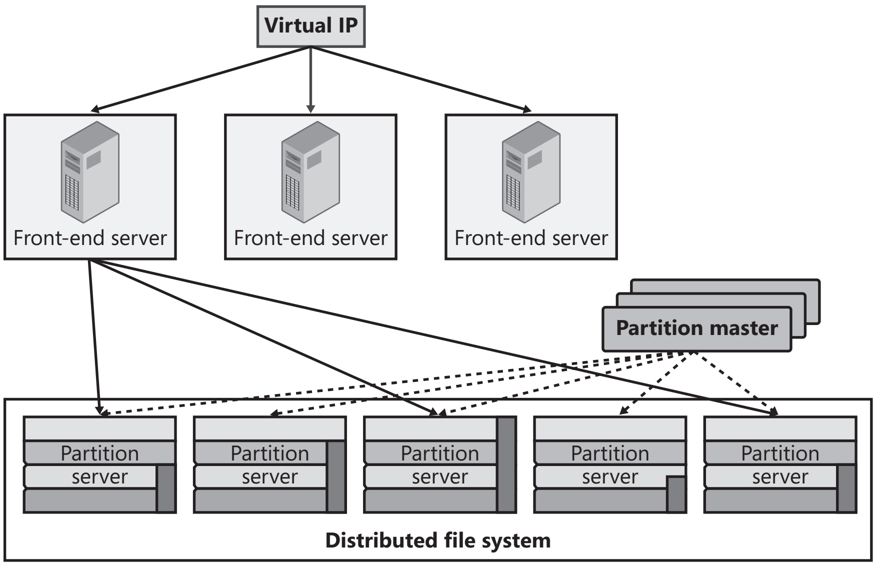


FIGURE 2-3 Windows Azure storage architecture consists of a front-end Virtual IP and three basic layers.

The following list describes the Windows Azure architecture:

- **Virtual IP** Requests for data storage operations enter through a Virtual IP (VIP) address, where they are routed to an available server in the front-end layer. The VIP balances the load of incoming requests by evenly distributing these requests to the front-end servers.
- **Front-end layer** The front-end (FE) layer accepts incoming requests and routes them to an appropriate partition server in the partition layer based on a partition map. It maintains this map to keep track of which servers are servicing which partitions.
- **Partition layer** The partition layer manages the partitioning of blob, table, and queue data. A data object belongs to a single partition identified by a partition key, and each partition is served by only one partition server. The partition layer manages what partition is served on what partition server and provides automatic load balancing of partitions across servers to meet the traffic requirements. A single partition server can serve the data for many partitions at one time.
- **Distributed File System layer** The DFS layer stores the data on disk and distributes and replicates the data across many servers. Data is stored by the DFS layer, but all data stored in servers managed by the DFS layer are accessible from any of the partition servers in the partition layer.

  The DFS layer provides Windows Azure storage with redundant durability because all data is replicated multiple times. The DFS layer spreads your data out over potentially hundreds of storage nodes. All of the replicas of your data are accessible from all the partition servers as well as from other DFS servers.

Failure management and durability

Hardware failure management and recovery is handled by the storage system according to the layer it occurs on. The following sections describe how failures are mitigated at each of the three layers.

Front-end layer failure mitigation

The load balancer monitors the responsiveness of each of the front-end servers. If one of the servers becomes unresponsive, the load balancer removes it from the available server pool so that incoming requests are no longer dispatched to it. This ensures that requests arriving at the VIP get sent only to healthy front-end servers.

Partition layer failure mitigation

If one of the partition servers is unavailable, the storage system immediately reassigns any partitions it was serving to other available partition servers and updates the front-end server partition maps to reflect this change. This allows the front-end servers to continue to correctly locate the partitioned data. When this reassignment is made, no data is moved on disk because all of the partition data is stored in the DFS layer and is accessible from any partition server. The storage system ensures that data partitions are always available to be served.

Distributed File System layer failure mitigation

If the storage system determines that one of the DFS servers is unavailable, the partition layer will direct the request instead to one of the other available DFS servers containing replicas of the data. The partition layer will resume usage of the DFS server when it is available again, but if a DFS server remains unavailable too long, the storage system generates an additional replica of the data to ensure an adequate number of durable replicas are maintained.

Data is stored in the DFS in basic units of storage called *extents*, which range in size from 100 MB to 1 GB in size. Each extent is spread randomly and replicated multiple times over multiple DFS servers. Data in a blob, entities in a table, or messages in a queue are all stored in one or more of these extents. A 10-GB blob may be stored across 10 1-GB extents, with three replicas for each extent, which potentially means that storage of this single blob is spread out over 30 DFS servers. The spreading and duplication of the data over multiple extents is what gives the DFS such resiliency against failure, and its inherent parallelism significantly increases the number of I/O operations that can be performed.

Each extent has a primary server and multiple DFS secondary servers. All writes to the extent are routed through the extent's primary DFS server. The writes are then forwarded to the secondary servers and success is returned back from the primary DFS server to the requestor once the data has been written to at least three DFS servers. If one of the DFS servers is unreachable, the DFS layer will select different DFS servers to write to until the data for the extent has been written at least three times. Once the third write has occurred, success is returned from the primary DFS server to the requestor. The other replicas will be updated asynchronously resulting in them being *eventually consistent*. When a subsequent read occurs on the same extent, the DFS layer will serve that data from any up-to-date extent replica.

To ensure high availability, no two replicas for an extent are ever placed on the same fault domain or upgrade domain (defined in the next section). If one fault domain goes down, or when an upgrade is occurring, there will always be healthy replicas from which the data can be accessed from. The data storage system automatically keeps the number of available replicas at a healthy level by replacing unavailable extent replicas. It does this by re-replicating to healthy servers when necessary.

Dynamic replication reduces the mean-time-to-recovery of healthy data extents when failures occur. If a DFS server fails for any reason, all extents that had a replica on that server are re-replicated to another server as quickly as possible; thus, a healthy number of replicas of every extent are always available. While re-replication is taking place, the other healthy replicas are used to service data requests and are used as data sources for re-replication. Because the extents are distributed randomly, network availability is also spread out to prevent hotspots.

The DFS layer also provides detection and repair of random errors (so-called *bit-rot*). It does this by computing and storing a checksum with the extent data. When data is read, this checksum is recomputed and verified against the stored checksum. In the rare event that the checksums do not match (indicating bit rot has occurred), the DFS discards the replica and re-replicates to another DFS server to bring the extent back to a healthy level of replication.

Fault and upgrade domains

As mentioned earlier in the chapter, the Microsoft Service Level Agreement guarantees that Windows Azure storage will successfully process add, update, read, and delete requests 99.9 percent of the time. Imagine the planning and expense necessary to achieve this level of reliability for your application within your own data center. To achieve your 99.9 percent objective, you would need to keep your downtime for all hardware, network access, electricity, HVAC, and software to fewer than nine hours per year! That's a pretty daunting task when you consider that even a single hardware failure, network outage, or software defect could exceed your entire SLA for the year. Achieving a 99.9 percent level of reliability for your application would require considerable investment in infrastructural software and redundant hardware, and so is yet another compelling argument for how cloud computing benefits your application.

To achieve the level of reliability promised in its SLA, Microsoft configures their hardware and software assets into two arrangements called *fault domains* and *upgrade domains*. These configurations boost the reliability to the promised level in ways that you will learn about in more detail later in this chapter, but first you need to learn exactly what these configurations are.

Fault domains

The term *fault domain* describes a unit of hardware components that share a single point of failure. One technique for maintaining high availability of your data during hardware failures is to spread your data out across multiple fault domains, thereby limiting the impact of a single hardware component failure. Windows Azure applies fault domain strategies to three layers of hardware in each data center: the server rack, the network switch, and the power supply. The Windows Azure SLA guarantees that at least two fault domains will be utilized at all three of these layers, meaning that if one rack, network switch, or power supply fails, at least one of these components will still be operating. The data stored on each storage node is replicated onto two other storage nodes across two or more fault domains at all hardware layers.

Upgrade domains

Upgrade domains represent another form of potential outage because storage nodes may not be available during application upgrades or operating system patches. To minimize the impact of upgrades, servers for each of the three layers (for example, the front-end layer, partition layer, and DFS layer) are spread evenly across upgrade domains in a similar fashion to the way data is spread over fault domains. Upgrades are performed through a *rolling upgrade* process, where only a small percentage of available servers are taken offline for upgrades to the data management service. Once upgraded, storage nodes are brought back up and then checked for health before being rolled back online.

Replication, geo-replication, and reliability

There are three replicas within the same data center in three separate racks, giving you robust resilience against hardware failures within a single data center. To add an even higher margin of safety against data loss, your data in your primary data center is replicated to another data center within the same geographical location. This provides you with one original and five current backups of your data in six separate server racks located in two separate geographical locations.

At the time of this writing, the geo-location displayed in the Windows Azure Management Portal, which you will see more of in the next chapter, is your primary geo-location. According to the September 15, 2011, post on the Windows Azure Storage Team Blog (*http://blogs.msdn.com/b/windowsazurestorage*), the secondary location will eventually be shown in a future version of the Management Portal (described a bit later in this book).

Geo-replication is included at no additional charge and is on by default. You can turn the feature on or off from the storage account's configuration pane of the Windows Azure portal. There is no cost savings in turning off the geo-replication. Replication to your secondary data center is handled asynchronously so that there is no load or performance impact on your applications.

When you perform data-modifying updates or deletes on your data, these changes are fully replicated on three separate and distinct storage nodes across three fault domains and upgrade domains within that data center.

After the transaction has been committed, a successful status is returned to the caller and the changes are asynchronously replicated to the secondary data center. The transaction is made durable in that data center by replicating itself across three storage nodes in different fault and upgrade domains. Because the updates are asynchronously geo-replicated, there is no impact on performance.

Microsoft's goal is to keep the data durable at both the primary and secondary locations. They accomplish this goal by keeping enough replicas in both locations to ensure that each location is capable of recovering itself from common failures such as a hard drive failure, storage node failure, rack failure, Top of the Rack (TOR) switch failure, and so on, without having to talk to the other location. The two locations talk to each other only to recover data in the event of common failures. If you had to failover to a secondary storage account then all the data that had been committed to the secondary location would already be there.

At the time of this writing, there was no SLA for how long it takes to asynchronously geo-replicate the data, but transactions are typically geo-replicated within a matter of a few minutes after the primary location has been updated.

Dynamic scalability

A significant factor to Microsoft in meeting the SLA obligations for data storage is scalability. The data service must scale so that 99.9 percent of the data operations can be executed successfully. In addition, Microsoft has established scalability targets per storage accounts. At the time of this writing, the following targets are in place:

- **Capacity** Up to 100 terabytes
- **Transactions** Up to 5,000 entities, messages, or blobs per second
- **Bandwidth** Up to 3 GB per second

At the time of this writing, all objects stored in Windows Azure storage have a partition key, which the data management service uses in allocating resources. One or more storage nodes will be allocated to your data dynamically. The partition key is your way of indicating your preferences for how this allocation is performed. A single partition key can have an entire storage node devoted to it if the data management service determines that is what is necessary to meet these scalability targets.

Every storage object (blobs, table entities, and queue messages) has a partition key that is used to locate the object in the data management service. The partition key is also used to load balance and dynamically partition the objects across storage nodes to meet storage request traffic in accordance with the scalability objectives. The partition keys used by storage type are given in Table 2-1.

TABLE 2-1 Storage types and partition keys

Storage type	Full partition key
Blobs	*Container name + blob name*
Table entities	*Table name + partition key*
Queue messages	*Queue name*

As you might infer from the differences in full partition keys shown in Table 2-1, the scalability characteristic of the three data storage types is significantly different. Blobs use the name of the blob within a container, but the partition level for queues is at the queue name level (not at the individual message level). Table entities are special in the sense that the data for the partition key is part of the data being stored, where blobs and queues are at the level of name (regardless of the data contained in the blob or message).

A blob always lives in one partition; two blobs could be on separate partitions. A queue also lives on one partition; two queues may live on separate partitions. A set of entities from a table could live on one partition; different sets from the same table could be on different partitions. Of course, different tables could be on separate partitions.

RESTful APIs

The data management services are exposed through an open and RESTful API, which can be used from any platform (including many non-Microsoft platforms). The RESTful API is covered throughout the book. These APIs are accessible from anywhere on the public Internet, allowing you to use the data management services from any kind of application, even applications not written using Microsoft technologies. Your applications might be running on-premises but storing and retrieving data from Windows Azure storage in order to take advantage of its multiple geographical locations, reliability, redundancy, and dynamic scalability characteristics. You might even have your application deployed to a competitive cloud platform but use Windows Azure storage instead of the competitor's equivalent. The reasons for hybrid configurations are abundant, but certainly the features offered by storage platforms such as performance, location, reliability, and their pricing structure are almost always going to be factors.

Software development kits

The RESTful API of Windows Azure storage requires the use of HTTP verbs against the Windows Azure storage endpoints. This includes repetitively populating the required HTTP headers and providing security credentials. Of course, this pattern quickly emerges to any application developer making use of data management services. Most application developers tend to work at a higher layer of abstraction than the HTTP transport layer by using object oriented programming paradigms. To formalize these patterns, Microsoft has provided several client library software development kits (SDKs) for Microsoft .NET languages, Node.js, Java, and PHP. There is also an oddly named "other" SDK which contains the storage and compute emulators as well as package and deployment tools for developers running on a Windows machine.

Storage library ports are available on other platforms as well, including Python, Ruby, Perl, and JavaScript. Steve Marx published a blog on many of these libraries that you can find at this URL:

http://blog.smarx.com/posts/windows-azure-storage-libraries-in-many-languages

There is also source code available on the Windows Azure site, which might prove useful in developing ports to other platforms:

https://www.windowsazure.com/en-us/develop/downloads/

In the chapters that follow, you will see many transcripts of HTTP traffic that were gathered using a diagnostic tool called Fiddler. *Fiddler* is a free tool for inspecting, diagnosing, and replaying HTTP web traffic. It is installed as an HTTP proxy that runs on port 8888 of your development workstation. WinINET-based applications such as Windows Internet Explorer will automatically use Fiddler as an HTTP Proxy when the Capture Traffic check box is selected on Fiddler's File menu. You can debug the traffic of any application that is capable of being configured to use an HTTP Proxy. It is highly recommended that you download and install a copy of Fiddler on your own development machine, because this tool will likely save you hours of diagnostic time by allowing you to directly monitor and debug your own application's requests against the Windows Azure data management service. You can read more about Fiddler and download a free copy from *http://www.fiddler2.com.*

Pricing

In the world of cloud computing and storage, it is impossible to not take pricing into account when making architectural decisions. Unfortunately, specific prices for Windows Azure are under constant review and adjustment based on a wide variety of factors. Variable prices that can change rapidly are a result of the commodification of computing and storage resources. The latest pricing information is available online at:

http://www.microsoft.com/windowsazure/pricing/

Although subject to change, at the time of publication, the following statements were true:

- Data storage utilized (including metadata) is 12.5 cents per gigabyte per month.
- Data transfer into a data center is free, but outbound data egress is billable at 12.5 cents per gigabyte per month. Transfer within the same subregion is free.
- Transactions are billable based on the number of I/O transactions completed at 1 cent per 10,000 per month.

If you are using a client library to assist you with development, you should be aware that some client library methods make multiple I/O requests to the data center to complete a single logical call.

Analytics and metrics

Windows Azure Storage Analytics provides metrics and logging of your storage account activity. Logs provide tracing of requests, and metrics provide capacity and request statistics. This information can be very useful in analyzing usage trends and in diagnosing storage account issues. Logging and metrics are controlled independently of one another and target each type of storage: blobs, tables, and queues. You will be introduced to the specifics of each in Chapters 5, 6, and 7, respectively.

Conclusion

The purpose of this chapter was to acquaint you with the Windows Azure storage platform. In particular, the chapter familiarized you with the features of the platform, which add valuable data availability, reliability, and protection assurances to your applications, as well as the tools for gathering analysis and metrics.

Although Windows Azure is built on Windows servers, it should be apparent that this data management service is sold piecemeal and may be used in any heterogeneous application that requires NoSQL type storage. In Chapter 4, "Accessing Windows Azure data storage," you will explore the Windows Azure SDKs for leveraging Windows Azure storage from Microsoft and non-Microsoft platforms.

CHAPTER 5

Blobs

In this chapter:

In this chapter, you learn about Windows Azure blob storage. First you examine the characteristics of this kind of data storage, including the kinds of real-world data and storage scenarios that lend themselves well to blob storage. You then learn about the organizational structure of this storage type, including the naming conventions and other rules that must be followed. This chapter discusses how to perform common create, read, update, and delete (CRUD) operations on blobs and their containers. To deepen your understanding, you tackle the advanced and valuable but often overlooked features of blobs, such as metadata, snapshots, and granular security access, which allow CRUD operations to be performed only by authorized parties. Finally, you learn how to write applications for robustness and resiliency in the cloud.

Blob basics

BLOB is an acronym for *Binary Large Object*, but the uppercase convention is generally ignored in favor of the more colloquial lowercase *blob*, which I use throughout the book. A *blob* holds arbitrarily structured data, which the blob has no knowledge of. To the blob, the data it contains is just a bunch of random bytes that may be read or written to either sequentially or in randomly accessed chunks (called *blocks*, or *pages*). Although the data contained in a blob may have a structure and may even adhere to a schema, the blob itself, as just mentioned, has no knowledge of what this structure might be. Blobs are often used to store documents such as Microsoft Word, Microsoft Excel, and XML documents; pictures; audio clips; videos; and backups of file systems. Files that might be stored on your computer's hard drive, or content that you might publish on a website, can alternatively be stored in blob storage.

In addition to the data contained within a blob, a blob also stores its own name, a small amount of metadata—8 kilobytes (KB) at the time of this writing—and an MD5 hash that can be used to validate a blob's integrity.

The cloud fabric manages the dynamic scaling of your data to meet demand. If a particular set of blobs are receiving a high volume of traffic, the cloud fabric will move those blobs to their own storage node. In a more extreme circumstance, an individual blob could potentially be on its own storage node. An individual blob cannot float around on its own anywhere it pleases, however; it must be stored in a structure called a *blob container*, which you will learn about later in this chapter. Windows Azure storage provides two distinct types of blob: the block blob and the page blob. You'll examine the block blob first.

Block blobs

Block blobs are useful in sequential access scenarios when storage and consumption of the data can begin at the first byte and end at the last. These blobs can be uploaded in equal-sized chunks referred to as *blocks*. This characteristic makes them well suited for applications requiring recovery from transmission failures, because transmission can be simply resumed from the last successfully transmitted block. Blocks in a blob may also be uploaded in parallel to increase throughput. An individual block blob can be any size up to 200 gigabytes (GB). When on-demand access to arbitrary locations within a blob is required, a better option may be the page blob, which is covered next.

Page blobs

Page blobs are useful when storage and consumption of the data may occur in any order. When on-demand access to arbitrary locations within a blob is required, the page blob is often the best option. An individual page blob can be any size up to 1 terabyte. Page blobs may also be sparsely populated, which is useful when implementing certain kinds of data structures and algorithms. Microsoft uses the sparsely populated page blob as the basis for *drive storage*, which is a virtual VHD—and for those paying careful attention, that would be a *Virtually Virtual Hard Drive*! Microsoft charges only for the pages that are occupied, so if you had a 1-terabyte blob with only 2 GB of population, you would pay only for the 2 GB of actual storage space used. This cost does not include fees for egress out-of-data-center and transaction fees, which are not impacted by a page blob's ability to be sparsely populated.

Blob containers

The structures used to store blobs are called *blob containers*. Blob containers provide a unit of organization and also of privacy sharing. By default, all blobs stored in a container share the same level of sharing, either private or public. Private containers require credentials to perform operations, whereas public containers allow anonymous read-only access to all blobs stored in the container. Creation, deletion, and update of the blobs stored in a container always require an authenticated request, irrespective of the privacy settings you assigned to the container.

An individual blob container can hold anywhere from zero to an infinite number of individual blobs. There is a limit, of course, on the total amount of storage capacity available with your account (not to mention the likely constraints you have on the money available to pay for your storage), but the limit is placed on your storage capacity, not on the number of blobs that can be placed in a single blob container. Because the capacity restrictions on an account are so large, in most situations, this number is virtually limitless.

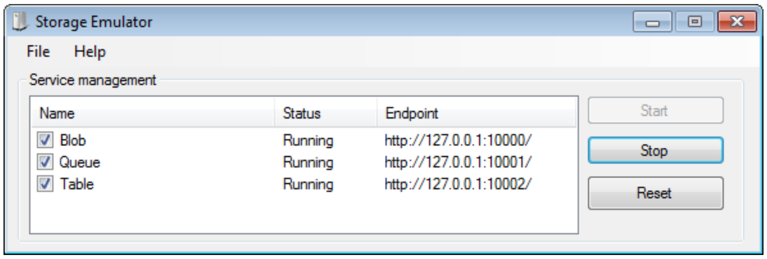
No limit is placed on the number of containers that you can have in a single Windows Azure storage account, but just like individual blobs, the actual numerical limit is determined indirectly by the storage capacity of your Windows Azure storage account.

Blob containers allow access policies to be applied, which control access and operations performed against the individual blobs that the containers encapsulate. You'll learn more about access policies later in this chapter.

Blob addressing

Blob resources are located in data storage via URLs that match this pattern: *http://*<account>*.blob.core.windows.net/*<container>*/*<blobname>. The <*account*> placeholder is the Windows Azure account name, <*container*> is the blob's container name, and <*blobname*> is the name of the blob (for example, *http://wintellect.blob.core.windows.net/pictures/Employee.jpg*).

When using the local development storage emulator, the URL pattern is slightly different. The hostname becomes the IP address of the loopback adapter (that is, 127.0.0.1), to which the port number *10000* and the hardcoded literal account name *devstorageaccount1* are appended to form the complete base address, as depicted in the Storage Emulator window shown in Figure 5-1. The container name and blob name are appended to this base address to form the full URL of a resource (for example, *http://127.0.0.1:10000/devstorageaccount1/pictures/Employee.jpg*).



Name	Status	Endpoint
✓ Blob	Running	http://127.0.0.1:10000/
✓ Queue	Running	http://127.0.0.1:10001/
✓ Table	Running	http://127.0.0.1:10002/

FIGURE 5-1 The IP address is shown in the Windows Azure Storage Emulator window.

Business use cases

Much of the nontextual content displayed by web browsers is blob data. This kind of content tends to be significantly larger in size than the markup that references it, making it more demanding on servers and networks to deliver. Images, documents, audio, and video files are all good examples of this kind of bulky data. Because of size or demand (or both), some of this data will inevitably require greater server and network capacity to deliver, and this creates challenges for redistributing the data to meet demand.

Of course, blob data is not generally sent to the browser with the HTML markup of a website application; instead, URLs to the resources are embedded in the HTML tags that the browser receives and then uses to retrieve the referenced resources and render them locally on the user's machine. Because each resource is referenced by a URL, it makes no difference to the client's browser whether the resource is located on the same server that the HTML was retrieved from, or in another location in an entirely different domain. It is therefore quite easy to take advantage of the massive and dynamic scalability of Windows Azure blob storage to supplement on-premise and cloud-deployed applications by storing this content there. The same thing is true for other kinds of free-standing content accessed by a URL, such as Word, Excel, and PDF documents.

Blob storage structure

Figure 5-2 shows the hierarchical relationships between storage accounts, blob containers, blobs, and pages. A Windows Azure storage account encapsulates zero or more blob containers, and each blob container can in turn encapsulate a set of zero or more blobs. The final column of Figure 5-2 shows the encapsulation of the individual blocks or pages of a blob.

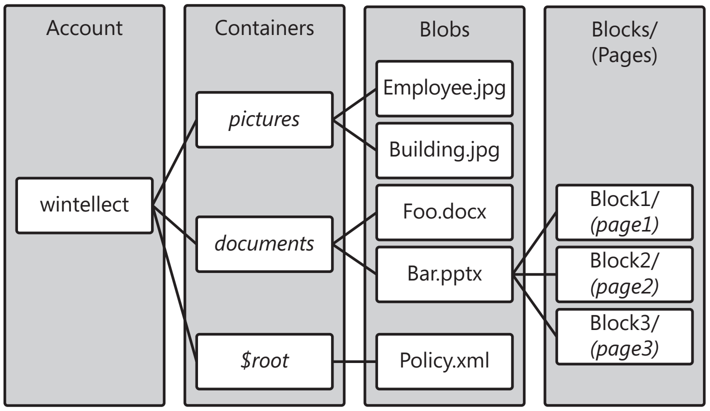


FIGURE 5-2 The blob storage structure is a hierarchical relationship between storage accounts, blob containers, blobs, and pages.

A storage account can be visualized as being similar to the root directory of your computer's hard drive, where the blob containers are like directory folders. Individual blobs can be thought of like files placed in a directory. Furthering this analogy, blobs are frequently named with common suffixes matching their content type (just as files are named with extensions that reflect their types, such as the .jpg or .png file extensions for image files, and the .docx extension for a Word file). Unlike the directory structure on your computer's hard drive, which can contain nested subfolders, blob containers cannot contain subcontainers. The way that subfolder-like behavior can be simulated is discussed a little later in this chapter.

For security and architectural reasons, there may be requirements for a blob to be physically located in the base address of a URL. For example, a cross-domain policy file is an XML document that adheres to a specification published by Adobe. This kind of file is used to grant web clients such as Microsoft Silverlight, Adobe Flash Player, and Adobe Reader permission to handle data across multiple domain boundaries. When a client running from one domain makes a request for a resource located in a secondary domain, the secondary domain must have a cross-domain policy file granting access to the requested resources in order for the web client to continue with the request. The specification requires that the file be named *policy.xml* and that it be located in the root directory of the secondary domain.

Because blobs must be stored in a blob container, a special hidden blob container named *$root* was created. The *$root* container is aliased to the base address of the domain. Any blob placed in the *$root* container will be accessible both by its physical URL (including its *$root* container name) and by its alias URL off the base address of the domain. The following two URLs are equivalent:

```
http://www.wintellect.com/$root/Policy.xml
http://www.wintellect.com/Policy.xml
```

Navigating blob container hierarchies

As suggested earlier, a rough analogy of blob storage is your file system. Actually, when developing your cloud-deployable software, your file system may be used in some circumstances as an adequate on-premise substitute for blob storage (without a few of the advanced features such as snapshots and shared access signatures). You may even consider implementing a provider model in your software to facilitate this kind of convenient on-premise abstraction.

In your file system, files are placed within directories, and those directories are stored within other directories to create an extensive organizational hierarchy. All directories can be traversed back to a single root directory that houses the entire tree structure. Blob containers are like directories that live within the root directory of blob storage, but the analogy begins to weaken at this point because blob containers may not be embedded within other blob containers. If that were the end of the story, you would be left with a very flat file system. Fortunately, this is not the case. The Windows Azure client library provides support for accessing blobs by using a simulation of a nested file system, thus allowing directory-style navigation over delimiters used in your blob names, such as the slash character.

To see how to navigate flat blob storage as if it were hierarchical, you'll first create a set of blobs in a container that uses a path delimiter. In this case, you will use the default delimiter of a slash (/).

The following code creates a container called *demo* and then populates this container with eight blobs named *FileA*, *FileB*, *Dir1/FileC*, *Dir1/FileD*, *Dir1/Dir2/FileE*, *Dir3/FileF*, *Dir3/FileG*, and *Dir4/FileH* by uploading an empty string as the content of each blob. The *UseFlatBlobListing* property of an instance of the *BlobRequestOptions* class is used as a parameter to control whether the container is navigated. You set this property to true when you want each blob in the container to be navigated without regard to the delimiter, and to false when you want navigation to behave as if the container were a file system style directory.

```
public static void DirectoryHierarchies(CloudStorageAccount account) {
    Console.Clear();
    CloudBlobClient client = account.CreateCloudBlobClient();
    Console.WriteLine("Default delimiter={0}", client.DefaultDelimiter /* settable */);
    Console.WriteLine();

    // Create the virtual directory
    const String virtualDirName = "demo";
    CloudBlobContainer virtualDir =
    client.GetContainerReference(virtualDirName).EnsureExists(true);

    // Create some file entries under the virtual directory
    String[] virtualFiles = new String[] {
                        "FileA", "FileB", // Avoid  $&+,/:=?@ in blob names
                        "Dir1/FileC", "Dir1/FileD", "Dir1/Dir2/FileE",
                        "Dir3/FileF", "Dir3/FileG",
                        "Dir4/FileH"
                };
    foreach (String file in virtualFiles) {
        virtualDir.GetBlockBlobReference("Root/" + file).UploadText(String.Empty);
    }
```

```
      // Show the blobs in the virtual directory container
      ShowContainerBlobs(virtualDir);   // Same as UseFlatBlobListing = false
      Console.WriteLine();
      ShowContainerBlobs(virtualDir, true);

      // CloudBlobDirectory (derived from IListBlobItem) is for traversing
      // and accessing blobs with names structured in a directory hierarchy.
      CloudBlobDirectory root = virtualDir.GetDirectoryReference("Root");
      WalkBlobDirHierarchy(root, 0);

      // Show just the blobs under Dir1
      Console.WriteLine();
      String subdir = virtualDir.Name + "/Root/Dir1/";
      foreach (var file in client.ListBlobs(subdir))
         Console.WriteLine(file.Uri);
   }

   private static void ShowContainerBlobs(CloudBlobContainer container,
        Boolean useFlatBlobListing = false, BlobListingDetails details = BlobListingDetails.None,
             BlobRequestOptions options = null, OperationContext operationContext = null) {
      Console.WriteLine("Container: " + container.Name);
      for (BlobResultSegment brs = null; brs.HasMore(); ) {
         brs = container.ListBlobsSegmented(null, useFlatBlobListing, details, 1000,
               brs.SafeContinuationToken(), options, operationContext);
         foreach (var blob in brs.Results) Console.WriteLine("   " + blob.Uri);
      }
   }

   private static void WalkBlobDirHierarchy(CloudBlobDirectory dir, Int32 indent) {
      // Get all the entries in the root directory
      IListBlobItem[] entries = dir.ListBlobs().ToArray();
      String spaces = new String(' ', indent * 3);

      Console.WriteLine(spaces + dir.Prefix + " entries:");
      foreach (var entry in entries.OfType<ICloudBlob>())
         Console.WriteLine(spaces + "   " + entry.Name);

       foreach (var entry in entries.OfType<CloudBlobDirectory>()) {
         String[] segments = entry.Uri.Segments;
         CloudBlobDirectory subdir = dir.GetSubdirectoryReference(segments[segments.Length - 1]);
         WalkBlobDirHierarchy(subdir, indent + 1); // Recursive call
      }
   }

   private static void ShowContainer(CloudBlobContainer container, Boolean showBlobs) {
      Console.WriteLine("Blob container={0}", container);

      BlobContainerPermissions permissions = container.GetPermissions();
      String[] meanings = new String[] {
                              "no public access",
                              "anonymous clients can read container & blob data",
                              "anonymous readers can read blob data only"
                     };
      Console.WriteLine("Container's public access={0} ({1})",
         permissions.PublicAccess, meanings[(Int32)permissions.PublicAccess]);
```

```
    // Show collection of access policies; each consists of name & SharedAccesssPolicy
    // A SharedAccesssBlobPolicy contains:
    //    SharedAccessPermissions enum (None, Read, Write, Delete, List) &
    //    SharedAccessStartTime/SharedAccessExpireTime
    Console.WriteLine("   Shared access policies:");
    foreach (var policy in permissions.SharedAccessPolicies) {
        Console.WriteLine("   {0}={1}", policy.Key, policy.Value);
    }

    container.FetchAttributes();
    Console.WriteLine("   Attributes: Name={0}, Uri={1}", container.Name, container.Uri);
    Console.WriteLine("   Properties: LastModified={0}, ETag={1},",
         container.Properties.LastModified, container.Properties.ETag);
    ShowMetadata(container.Metadata);

    if (showBlobs)
        foreach (ICloudBlob blob in container.ListBlobs())
            ShowBlob(blob);
}

private static void ShowBlob(ICloudBlob blob) {
    // A blob has attributes: Uri, Snapshot DateTime?, Properties & Metadata
    // The CloudBlob Uri/SnapshotTime/Properties/Metadata properties return these
    // You can set the properties & metadata; not the Uri or snapshot time
    Console.WriteLine("Blob Uri={0}, Snapshot time={1}", blob.Uri, blob.SnapshotTime);
    BlobProperties bp = blob.Properties;
    Console.WriteLine("BlobType={0}, CacheControl={1}, Encoding={2}, Language={3},
         MD5={4}, ContentType={5}, LastModified={6}, Length={7}, ETag={8}",
       bp.BlobType, bp.CacheControl, bp.ContentEncoding, bp.ContentLanguage,
            bp.ContentMD5, bp.ContentType, bp.LastModified, bp.Length, bp.ETag);
    ShowMetadata(blob.Metadata);
}

private static void ShowMetadata(IDictionary<String, String> metadata) {
    foreach (var kvp in metadata)
        Console.WriteLine("{0}={1}", kvp.Key, kvp.Value);
}
```

Executing this code produces the following results.

```
Default delimiter=/

Container: demo, UseFlatBlobListing: False
   http://azureinsiders.blob.core.windows.net/demo/Dir1/
   http://azureinsiders.blob.core.windows.net/demo/Dir3/
   http://azureinsiders.blob.core.windows.net/demo/Dir4/
   http://azureinsiders.blob.core.windows.net/demo/FileA
   http://azureinsiders.blob.core.windows.net/demo/FileB

Container: demo, UseFlatBlobListing: True
   http://azureinsiders.blob.core.windows.net/demo/Dir1/Dir2/FileE
   http://azureinsiders.blob.core.windows.net/demo/Dir1/FileC
   http://azureinsiders.blob.core.windows.net/demo/Dir1/FileD
   http://azureinsiders.blob.core.windows.net/demo/Dir3/FileF
   http://azureinsiders.blob.core.windows.net/demo/Dir3/FileG
   http://azureinsiders.blob.core.windows.net/demo/Dir4/FileH
   http://azureinsiders.blob.core.windows.net/demo/FileA
   http://azureinsiders.blob.core.windows.net/demo/FileB

demo entries:
   FileA
   FileB
   Dir1 entries:
      FileC
      FileD
      Dir2 entries:
         FileE
   Dir3 entries:
      FileF
      FileG
   Dir4 entries:
      FileH

http://azureinsiders.blob.core.windows.net/demo/Dir1/Dir2/FileE
http://azureinsiders.blob.core.windows.net/demo/Dir1/FileC
http://azureinsiders.blob.core.windows.net/demo/Dir1/FileD
```

After printing the delimiter being used, the blob container named *demo* is iterated by using the *UseFlatBlobListing* property of an instance of *BlobRequestOptions* set to *false*. This option suppresses the iterator's descent into the blob names beyond the first occurrence of the delimiter character, providing you with a high-level listing of all of the simulated directories in the root of the container. The next section of code performs the same operation, with the *UseFlatBlobListing* property of an instance of *BlobRequestOptions* set to true. You'll see more on this class later in this chapter. With this option set, the container's *ListBlobsSegmented* method recursively returns the subdirectories in each directory (using the segmented technique described in Chapter 4, "Accessing Windows Azure data storage"), providing a flattened view of the blobs in the container.

Occasionally, because of business requirements, you may have to traverse all of the blobs in a container as if they were files in a file system tree. The next section of code calls the *WalkBlobDir-Hierarchy* routine, which recursively calls itself to list the contents of each segment of the delimited blob names. The *CloudBlobDirectory* class (which derives from *CloudBlob*) provides the abstraction of a blob directory. You traverse the entire tree by calling the *GetSubdirectory* method on each directory to retrieve a list of subdirectories and then use that list to recursively call back into the *WalkBlobDir-Hierarchy* routine.

In some situations, it may be desirable to locate all blobs that are contained in a single simulated directory structure. This can be accomplished using the *ListBlobsWithPrefix* method of your instance of *CloudBlobClient*, as shown in the preceding section of the code.

Storage Client library blob types

The Storage Client library provides abstractions for blobs and containers, making them easy to work with in the Microsoft .NET Framework code. The following alphabetized list explains the most important types, methods, and properties used in the topics covered later in this chapter:

- *CloudBlob* provides a convenient object-oriented abstraction for working with an individual blob.
- *CloudBlobContainer* provides a convenient object-oriented abstraction for working with a blob storage container.
- *CopyFromBlob* copies an existing blob's contents, properties, and metadata to a new blob.
- *Create[IfNotExist]* creates a blob container or optionally creates the container only if the container does not already exist.
- *Delete* deletes a blob container and its contents.
- *FetchAttributes* returns the container's attributes, including its system properties and any user-defined metadata.
- *Get/SetPermissions* gets or sets the permission settings for the container.
- *GetBlobReference* returns a reference to a blob in the container.
- *GetSharedAccessSignature* returns a shared access signature for the container.
- *ListBlobs[Segmented]* returns an enumerable collection of the blobs in the container, or a segmented enumerable collection of the blobs in the container.
- *Metadata* returns the user-defined metadata for the blob or blob container.
- *Name* returns the name of the blob or blob container.
- *OpenWrite/Read* opens a stream for reading or writing the blob's contents.
- *Properties* returns the blob's system properties.
- *SnapshotTime* returns the *DateTime* value that uniquely identifies the snapshot (only when the blob is a snapshot).
- *Upload(ByteArray/File/FromStream/Text)* uploads data from a byte array, file, stream, or string to a blob.
- *Uri* returns the blob or container's address.

Container and blob naming rules

You should be aware of several naming rules for blobs and their containers. A blob container name must be between 3 and 63 characters in length; start with a letter or number; and contain only letters, numbers, and the hyphen. All letters used in blob container names must be lowercase. Lowercase is required because using mixed-case letters in container names may be problematic. Locating trouble in a failing application related to the incorrect use of mixed-case letters might result in a lot of wasted time and endless amounts of frustration and confusion.

To make matters a bit confusing, blob names *can* use mixed-case letters. In fact, a blob name can contain any combination of characters as long as the reserved URL characters are properly escaped. The length of a blob name can range from as short as 1 character to as long as 1024 characters.

If you inadvertently violate any of these naming rules, you receive an HTTP 400 (Bad Request) error code from the data storage service, resulting in a *StorageClientException* being thrown if you are accessing blob storage using the Windows Azure software development kit (SDK).

You are not prohibited from using mixed casing in code, though, but some irregularities may adversely impact you when you do use it. For example, if you create a container properly in lowercase, but then later attempt to use that container in mixed-cased requests, your requests will all succeed because the mixed case container name is silently matched with the lowercase container name. This silent but menacing casing coercion can lead you to really scratch your head during debugging, so I strongly urge you to commit to memory the rule that blob container names must not contain uppercase letters.

Performing create, read, update, and delete blob operations

Blobs contain many operations for saving and retrieving data to and from storage. You'll begin with the simple operation of creating a new blob container and populating it with your first blob.

Blob container security

It is useful for you to organize your blobs into storage containers by grouping data with the same security requirements into the same containers (or sets of identically secured containers, as may be appropriate). This strategy should include grouping blobs that your application requires anonymous (public) read-only access to (which is our next topic). Because each blob can be referenced directly from the Internet using its URI, delivery of anonymous public read-only content to web browsers is one of the most useful purposes of blob storage. If blobs in the same container have differing security requirements, you probably want to re-factor your design until they don't. Blob containers are full-access when the request is made with the Windows Azure account key or public read-only (where anyone with the URL to the blob or blob container can read its contents and its metadata), or they might be more granular when the request is made with a Shared Access Signature. Each of these security models is covered in this chapter.

The Windows Azure account key should generally be kept secret, because it's really the key to the entire data fiefdom controlled by a single Windows Azure data storage account. Your application using the account key is similar to Microsoft SQL Server using an account with database owner authority. This trusted application model is generally adequate for many on-premise applications and services. However, you may want to give some attention and analysis to the security ramifications of using the trusted application model in your cloud architectures. The risks go up considerably when you're no longer operating behind the safety and protection of your corporate firewall, where identities are managed and under the careful control and scrutiny of your corporate personnel department, IT staff, and infrastructure team. You may also want to give some thought to using different storage accounts for different applications (or sets of applications) in order to compartmentalize your data so that the leak of one application's credentials is not a threat to the data of other applications.

Anonymous (public) read-only blob access

In the business use-case section of this chapter, I suggested that blob web content could be placed in Windows Azure blob storage, which the markup code could simply reference. To enable this scenario, the content must be publicly accessible via an unauthenticated web request. Most content on the web is public read-only data, but by default, blob containers do not allow public access, so to enable this business use-case, you have to set your permissions on your blob containers to grant the desired level of access to anonymous users. Blob storage is the only type of data storage in Windows Azure that allows public read-only access. (Unauthenticated public access is not available for Windows Azure table or queue storage.)

By default, no public access is granted to a blob container or the blobs it encapsulates. You will learn later in this chapter how you can change this setting to Blob to allow public access to individual blobs stored in the container or to Container, which grants public read-only access to the blob container and all the blobs contained therein. It is not possible to set public read-only access on an individual blob—only on its container.

Creating the blob container

You can create a new blob container named *demo* by sending an HTTP *PUT* request to the URI of the blob container location. The following request creates a new container called *demo* in the *azureinsiders* storage account.

```
PUT http://azureinsiders.blob.core.windows.net/demo?restype=container&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Mon, 17 Dec 2012 05:32:31 GMT
Authorization: SharedKey azureinsiders:+TRYhpqkDgZ6WlgG37lOqa+d/5tfvZXyYqpEKjaDs9w=
Host: azureinsiders.blob.core.windows.net
Content-Length: 0
```

The preceding code results in an HTTP status code 201 (Created) upon its successful completion.

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Last-Modified: Mon, 17 Dec 2012 05:32:31 GMT
ETag: "0x8CFAA2F0B3FF8C8"
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 79eea64c-7f01-4193-a6a5-1d918868dac8
x-ms-version: 2012-02-12
Date: Mon, 17 Dec 2012 05:32:31 GMT
0
```

In the Wintellect.DevCloudAppsAzureStorage project in the sample code, locate the *Blob-Patterns.Basics* method in *StoragePatterns.cs*. The *Basics* method, part of which is shown in the following code, accepts a *CloudStorageAccount*, which is a container around security credentials and storage endpoint addresses. You create a container named *demo* if one doesn't already exist.

```
// Use an OperationContext for debugging and to estimate billing
OperationContext oc = new OperationContext();
oc.SendingRequest += (Object s, RequestEventArgs e) => {
     HttpWebRequest request = e.Request;
};
oc.ResponseReceived += (Object s, RequestEventArgs e) => {
    HttpWebRequest request = e.Request;
    HttpWebResponse response = e.Response;
    RequestResult rr = e.RequestInformation;
};

CloudBlobClient client = account.CreateCloudBlobClient();

// Create a container:
CloudBlobContainer container = client.GetContainerReference("demo");
Boolean created = container.CreateIfNotExists(null, oc);
```

Listing storage account containers

After the blob container demo has been created in the storage account, you should be able to see it in storage. The following HTTP request against the *azureinsiders* storage account augments the URI with the query string parameter *?comp=list,* which in turn causes Windows Azure storage service to return a list of all blob containers for the storage account specified by the URI.

```
GET http://azureinsiders.blob.core.windows.net/?comp=list&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Mon, 17 Dec 2012 05:55:22 GMT
Authorization: SharedKey azureinsiders:7QR/PWux3s6anFSQR2gSV5UPUvPInEuh+jeO8R3sflk=
Host: azureinsiders.blob.core.windows.net
```

The preceding code returns a response containing an enumeration of blob containers for the storage account.

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/xml
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 7e1a5018-0080-472a-b817-33df3bb82f09
x-ms-version: 2012-02-12
Date: Mon, 17 Dec 2012 05:55:21 GMT

2F5
<?xml version="1.0" encoding="utf-8"?>
  <EnumerationResults AccountName="http://azureinsiders.blob.core.windows.net/">
    <Containers>
      <Container>
        <Name>demo</Name>
        <Url>http://azureinsiders.blob.core.windows.net/demo</Url>
        <Properties>
          <Last-Modified>Mon, 17 Dec 2012 05:32:31 GMT</Last-Modified>
          <Etag>"0x8CFAA2F0B3FF8C8"</Etag>
          <LeaseStatus>unlocked</LeaseStatus>
          <LeaseState>available</LeaseState>
        </Properties>
      </Container>
      <Container>
        <Name>manyblobs</Name>
        <Url>http://azureinsiders.blob.core.windows.net/manyblobs</Url>
        <Properties>
          <Last-Modified>Tue, 05 Jun 2012 00:14:02 GMT</Last-Modified>
          <Etag>"0x8CF10C73F0E2A12"</Etag>
          <LeaseStatus>unlocked</LeaseStatus>
          <LeaseState>available</LeaseState>
        </Properties>
      </Container>
    </Containers>
  <NextMarker />
</EnumerationResults>
0
```

If you're using the Windows Azure storage client, the *ListContainers* method of the storage client will return an *IEnumerable<CloudBlobContainer>*.

```
// Show this account's containers:
foreach (var c in client.ListContainers(null, ContainerListingDetails.None,
operationContext: oc))
    Console.WriteLine(c.Uri);
```

At this point, you have an empty blob container. The following code creates and populates two empty blobs in that blob container.

```
// Create 2 blobs in the container:
CloudBlockBlob blob = container.GetBlockBlobReference("SomeData.txt");
using (var stream = new MemoryStream(("Some data created at " + DateTime.Now).Encode())) {
    blob.UploadFromStream(stream, operationContext: oc);
}
```

```
using (var stream = new MemoryStream()) {
    blob.DownloadToStream(stream, operationContext: oc);
    stream.Seek(0, SeekOrigin.Begin);
    Console.WriteLine(new StreamReader(stream).ReadToEnd());   // Read the blob data back
}
```

With the blob container and blobs created, you are ready to explore permissions settings.

Setting blob container permissions

Container permissions control public read-only access (public access) to a blob container and the blobs it contains. It is not possible to set public access permission on an individual blob. This permission is applicable only to blob containers. An individual blob inherits its public access characteristic by virtue of the container's permission. To control public access to the blob container and its contents, perform an HTTP *PUT* operation against the URI of the blob container, setting the x-ms-blob-public-access header to one of three values listed in Table 5-1. (Note that the header is all lowercase letters, whereas the object model of the API depicted in Table 5-1 is Pascal-cased.) A value of *container* grants anonymous public read-only access to a blob container and its contents, a value of *blob* grants the same access but only to the blobs in the container, and a value of *off* prohibits any anonymous access.

In the following example, you are indicating that public access is being granted to the container and all of the blobs that it may contain.

```
PUT http://azureinsiders.blob.core.windows.net/demo?
    restype=container&comp=acl&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-blob-public-access: container
x-ms-date: Mon, 17 Dec 2012 06:54:11 GMT
Authorization: SharedKey azureinsiders:NYvUlXRWqZCFXhtPQu/o8OFiKe8aKOlOSXAbHeyEOUY=
Host: azureinsiders.blob.core.windows.net
Content-Length: 62

<?xml version="1.0" encoding="utf-8"?><SignedIdentifiers />
```

Successful execution of the preceding HTTP *PUT* request will result in an HTTP status code 200 (OK).

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Last-Modified: Mon, 17 Dec 2012 06:54:11 GMT
ETag: "0x8CFAA3A745859B4"
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 54cea7b5-26b7-444c-b3a5-10a251e779ad
x-ms-version: 2012-02-12
Date: Mon, 17 Dec 2012 06:54:10 GMT
```

Public access may be granted through the Windows Azure client library, too. A *BlobContainer-Permissions* object is used to control public access to a blob container and the blobs it contains. An instance of *BlobContainerPermissions* has a *PublicAccess* property, which can be set to one of three values, as shown in Table 5-1.

TABLE 5-1 Blob container public access permission settings

Setting	Public read-only access
Off	Grant no public access to the container or the blobs stored in the container.
Blob	Grant public read-only access to all of the blobs stored in the container, but not to the container itself.
Container	Grant public read-only access to the container and all of the blobs stored in the container.

To grant public access permissions to the blob container and its blob contents, you create an instance of *BlobContainerPermissions* and set its *PublicAccess* property to *BlobContainerPublicAccess-Type.Container*. You then call the container's *SetPermissions* method, passing in the permission object. There are three values to the *BlobContainerPublicAccessType*: *Off* (the default) prohibits public read-only access to the container and its blobs; *Blob* grants access to the blobs in the container (but not the container itself); and *Container* grants access to read the container and the blobs it encapsulates. The next bit of code illustrates this.

```
// Change container's security to allow read access to its blobs:
BlobContainerPermissions permissions = new BlobContainerPermissions {
    PublicAccess = BlobContainerPublicAccessType.Container
};
container.SetPermissions(permissions, operationContext: oc);

// Attempt to access a blob from browser & in code (succeeds):
Process.Start("IExplore", container.Uri.ToString()).WaitForExit();
using (var stream = new MemoryStream()) {
    anonymous.GetContainerReference("demo").GetBlockBlobReference("SomeData.txt")
       .DownloadToStream(stream, operationContext: oc);
    Console.WriteLine("Download result: " + stream.GetBuffer().Decode());
    Console.WriteLine();
}

// Show the container's blobs via REST:
Process.Start("IExplore", container.Uri + "?comp=list").WaitForExit();
```

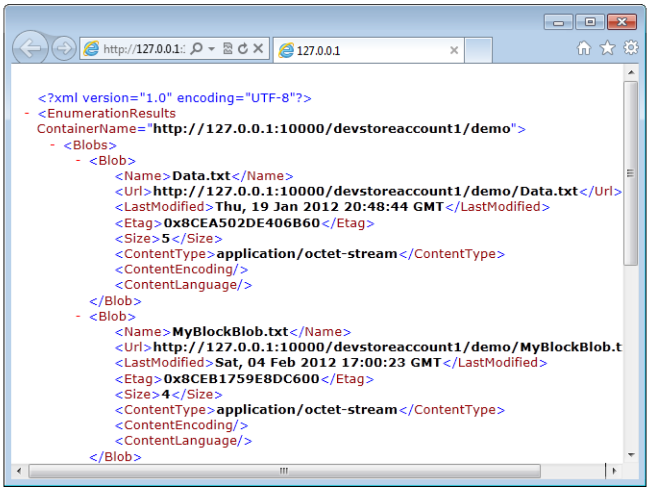
The blob container *demo* has the default public read-only permission of *off*, meaning that any attempt to read the blobs located in this container without credentials will fail. To test this assertion and prove this point, you launch a web browser to the URL of the blob you uploaded. You then attempt to access one of the blobs using code to demonstrate that this also fails.

```
// Change container's security to allow read access to its blobs:
BlobContainerPermissions permissions = new BlobContainerPermissions {
    PublicAccess = BlobContainerPublicAccessType.Container
};
container.SetPermissions(permissions, operationContext: oc);

// Attempt to access a blob from browser & in code (succeeds):
Process.Start("IExplore", container.Uri.ToString()).WaitForExit();
using (var stream = new MemoryStream()) {
    anonymous.GetContainerReference("demo").GetBlockBlobReference("SomeData.txt")
       .DownloadToStream(stream, operationContext: oc);
    Console.WriteLine("Download result: " + stream.GetBuffer().Decode());
    Console.WriteLine();
}
```

```
// Show the container's blobs via REST:
Process.Start("IExplore", container.Uri + "?comp=list").WaitForExit();
```

Now when you launch Windows Internet Explorer on the blob's URL, the contents of the blob are displayed. Because the container is now set to allow public read access, you can launch a browser directly against the blob container's URL, passing the filtration criteria (*?comp=list*) in the query string to list the contents of the blob container, as shown in Figure 5-3.



```
<?xml version="1.0" encoding="UTF-8"?>
- <EnumerationResults
  ContainerName="http://127.0.0.1:10000/devstoreaccount1/demo">
  - <Blobs>
    - <Blob>
        <Name>Data.txt</Name>
        <Url>http://127.0.0.1:10000/devstoreaccount1/demo/Data.txt</Url>
        <LastModified>Thu, 19 Jan 2012 20:48:44 GMT</LastModified>
        <Etag>0x8CEA502DE406B60</Etag>
        <Size>5</Size>
        <ContentType>application/octet-stream</ContentType>
        <ContentEncoding/>
        <ContentLanguage/>
      </Blob>
    - <Blob>
        <Name>MyBlockBlob.txt</Name>
        <Url>http://127.0.0.1:10000/devstoreaccount1/demo/MyBlockBlob.t
        <LastModified>Sat, 04 Feb 2012 17:00:23 GMT</LastModified>
        <Etag>0x8CEB1759E8DC600</Etag>
        <Size>4</Size>
        <ContentType>application/octet-stream</ContentType>
        <ContentEncoding/>
        <ContentLanguage/>
      </Blob>
```

FIGURE 5-3 A list of the blob container contents is shown here.

You're going to set the *PublicAccess* property of the blob container to *BlobContainerPublicAccess-Type.Off* in the code that follows. This step may seem superfluous because *Off* is the default setting, but you want to ensure clarity of the demonstration, and you also want to ensure that the resulting access you observe is verifiably a result of the Shared Access Signature (SAS) and not a side-effect of anonymous access being granted. (You learn more about SAS in the next sections.) You use the *SetPermissions* method of the blob container to apply the *BlobContainerPermission* object to remove public access to the blob container and its contents. As mentioned earlier, this is equivalent to executing an HTTP *PUT* operation against the URI of the blob container with the *x-ms-blob-public-access* header set to a value of *off*. Attempting to show the contents of the container or blob in a web browser after applying the permissions results in an HTTP 403 (Forbidden) error code, which proves that no anonymous access is allowed to your blob container. The following code demonstrates this.

```
container.SetPermissions(new BlobContainerPermissions {
   PublicAccess = BlobContainerPublicAccessType.Off });
CloudBlob blob = container.GetBlobReference("test.txt");
blob.UploadText("Data accessed!");
// This fails
Process.Start("IExplore", blob.Uri.ToString()).WaitForExit();
```

Shared Access Signatures and shared access policies

So what do you do when you want to grant to another party more granular control over blobs in a container without providing your private account key? This is where Shared Access Signatures are useful. A *Shared Access Signature* is a bit of cryptographic data that is appended to the query string of the URL that is used by Windows Azure storage to grant more granular control to blobs or their containers for a short period of time. After issued, an SAS is good for 60 minutes by default. This time can be extended by applying a Shared Access Policy to the Shared Access Signature, which will be introduced shortly; however, it is important to keep in mind that the shorter the duration the signature is valid, and the more minimal the authorization granted by the SAS, the stronger the effectiveness of the security the policy provides.

When you execute the following code to create a blob called *test.txt* in the blob container *demo*, and then attempt to access the contents of that blob using its address in your web browser, the attempt fails because the blob container is not public by default.

```
        // Create a container (with no access) & upload a blob

// Create a container (with no access) & upload a blob
CloudBlobContainer container = account.CreateCloudBlobClient()
    .GetContainerReference("demo").EnsureExists();
container.SetPermissions(new BlobContainerPermissions {
     PublicAccess = BlobContainerPublicAccessType.Off
});
CloudBlockBlob blob = container.GetBlockBlobReference("test.txt");
blob.UploadText("Data accessed!");
Process.Start("IExplore", blob.Uri.ToString()).WaitForExit(); // This fails
```

In the following sections, you will grant granular permission to the blob for a specified period of time.

Shared Access Signature

An SAS can be thought of as a security permission filter of your blob or blob container's URI, with many segments providing the data necessary to establish the validity and duration of the signature provided in the *sig* field. The following is an example of what an SAS looks like.

```
http://account.blob.core.windows.net/container/blob
    ? st=2011-01-04T00:06:22Z
    & se=2011-01-04T01:06:22Z
    & sr=b
    & sp=r
    & si=Managers
    & sig=KKW…ldw=
```

The meaning of each segment of the SAS is described in Table 5-2.

TABLE 5-2 Shared Access Signature query string segments

Segment	Query string segment purpose
st	Signed start time (optional) provided in UTC format (for example, 2011-01-04T00:06:22Z).
se	Expiry time provided in UTC format (for example, 2011-01-04T00:06:22Z).
sr	Resource (container's blob or blobs).
sp	Permissions (*r*)ead/(*w*)rite/(*d*)elete/(*l*)ist. Permissions are supplied as single letters, and must be supplied in this order: *rwdl*. Any combination of these permissions is acceptable as long as they are supplied in the proper order (for example, *rw*, *rd*, *rl*, *wd*, *wl*, and *dl*). Specifying a permission designation more than once is not allowed.
si	Signature identifier (optional) relates an SAS to a container's shared access policies. The signature identifier must be 64 characters or fewer.
sig	Shared Access Signature. The URL parameters are signed (HMACSHA256) with the account's key.

The permissions that may be applied to a blob container using the *sp* segment are provided in Table 5-3.

TABLE 5-3 Allowable shared access signature permissions

Permission	Description
Read	Read content, properties, metadata, block list for blob (any blob in the container).
Write	Write content, properties, metadata, and block list for blob (any blob in container); copy blob is not supported.
Delete	Delete blob (any blob in the container).
List	Lists blobs in container. (This permission can be granted only to blob containers.)

It is important to note that the generation of a Shared Access Signature is a client-side cryptographic activity that makes use of the storage account key. There are no network requests made of the storage service in order to create one. To create an SAS, you simply take all of the signed query string parameter values delimited by newline characters and then hash this value using the HMACSHA256 algorithm to create the *sig* parameter. Any request received by the storage service bearing an appropriately formatted and cryptographically intact signature will be granted the access defined in the *sp* (permissions) parameter.

Because the security signature and related parameters are provided in the URL, it's very important to pay careful attention to the fact that they are subject to snooping and replay attacks if they are leaked to unintended parties, hijacked by someone sniffing traffic on the wire, or emailed to or from an employee who is ignorant of the security ramifications. You should use an SAS only in a production environment over HTTPS. You should also not use an SAS to expose blobs in a web browser, because the unexpired SAS will be cached in the browser history and hard drive of the client's machine and could be used by an unauthorized party to perform data operations. Similar precautions should be taken to keep applications from storing the SAS in a persistent data store, or from exposing the SAS directly or inadvertently in its clear text form. If the SAS must be stored, you should consider encrypting it. If a user can see the SAS, it's probably not secure enough and you may want to rethink your approach.

If you are using the Windows Azure SDK client libraries for the .NET Framework, the *GetShared-AccessSignature* method of a blob instance will ease your pain in manually constructing an SAS, saving you from having to fool around with a lot of messy URL string manipulations. The following code demonstrates how to create an SAS that is valid for a period of one hour beginning immediately, which will grant its bearer read, write, and delete permissions. You will see more on the *SharedAccess-Policy* and *SharedAccessPermission* classes shortly.

Creating a shared access policy

In the preceding code, you used an instance of a *SharedAccessPolicy* to create an SAS. The policy controls the usage characteristics of the SAS, for example, when it takes effect and how long it is valid. The policy also encapsulates, to the resource being protected, the permission set that you want to grant to the bearer of the SAS.

SharedAccessPermissions is an enumeration of bit flags that can be OR'd together to create the permission set necessary to meet your blob's permission requirements. In the following code, you select a permission set comprising the desired combination of read, write, and delete permissions.

```
Permissions perms = SharedAccessPermissions.Read |
    SharedAccessPermissions.Write |
    SharedAccessPermissions.Delete
```

There is also a *List* permission for use with blob containers.

Applying a shared access policy to a blob container

To apply a shared access policy to a blob, you first must create an instance of *SharedAccessPolicy* and then set its permissions and the effective start and end time properties (given in coordinated universal time, or UTC), as shown in the next bit of code. You can then call the blob's *GetSharedAccess-Signature* method, passing the shared access policy object as an argument to retrieve a signed shared access URL that can be used by callers to perform subsequent CRUD operations granted on the blob (as specified in the shared access policy).

```
// Create an SAS for the blob
var now = DateTime.UtcNow;
SharedAccessPolicy sap = new SharedAccessPolicy {
   // Max=1 hr after start
   SharedAccessStartTime = now,
   SharedAccessExpiryTime = now.AddHours(1),
   Permissions = SharedAccessPermissions.Read |
                 SharedAccessPermissions.Write |
                 SharedAccessPermissions.Delete
};
String sas = blob.GetSharedAccessSignature(sap);
String sasUri = blob.Uri + sas;
// This succeeds (in Internet Explorer, modify URL and show failure)
Process.Start("IExplore", sasUri).WaitForExit();
```

Alternatively, you can protect the blob and other blobs in the container by adding the Shared Access Policy to the container's collection of shared access policies.

Storing access policies

What if you have more stringent data access requirements than the SAS provides? For example, what if you require additional constraints on the starting or ending times that the SAS will be valid, or you require more granular control over the set of permissions being granted to a data storage item by an SAS? What if you require a means of revoking an SAS after it has been issued?

All of these tighter data access requirements can be met by augmenting a *SharedAccessSignature* with a stored access policy (SAP). Instead of being supplied as part of the query string parameters of the URL, the values that you select for your policy are stored with the data on the storage service. This decouples the SAS from the policy, thus allowing you to modify the parameters without having to re-issue another SAS. You can also revoke access granted to an SAS. The SAP is referenced by the *signedidentifier* field of the URL provided in Table 5-2. A signed identifier is a string containing 64 characters or fewer. You'll see these classes shortly, but first it's a good idea to review how to create a shared access policy through the RESTful API. In order to add new policies to the blob container, you must first retrieve the policies that are already present, as shown in the following HTTP *GET* request.

```
GET http://azureinsiders.blob.core.windows.net/demo
     ?restype=container&comp=acl&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Tue, 18 Dec 2012 06:13:06 GMT
Authorization: SharedKey azureinsiders:vB79RzKYOVkhdJ9NgMonq7OU4fI9DE4OHOpxipiVQOQ=
Host: azureinsiders.blob.core.windows.net

Authorization: SharedKey azureinsiders:N6SZp4XX6NaC3ZOXHqVC94jTSnoUBQrgDV/By2+OHRU=
Host: azureinsiders.blob.core.windows.net
```

This code returns the collection of policies in the body of the response. In this case, the *Signed-Identifiers* element is empty, showing that you have no stored access policies currently assigned to this blob storage container.

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/xml
Last-Modified: Tue, 18 Dec 2012 06:09:29 GMT
ETag: "0x8CFAAFD5FEF12F7"
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: a34d3e7d-29a5-4742-8efa-7acab29aff4c
x-ms-version: 2012-02-12
Date: Tue, 18 Dec 2012 06:13:05 GMT

3E
<?xml version="1.0" encoding="utf-8"?>
<SignedIdentifiers />
0
```

To add a policy, you execute an HTTP *PUT* request against the blob container's URI with the *comp=acl* query string parameter set. The body of the request contains a payload of signed identifiers. These signed identifiers represent the policies to be applied on the Windows Azure storage service side of the network for the blob container that is specified as the target of the request, as shown in the next code. Notice the *SignedIdentifier* ID is *Managers* and the *Permission* element has a value of *rw*.

```
PUT http://azureinsiders.blob.core.windows.net/demo
    ?restype=container&comp=acl&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Tue, 18 Dec 2012 06:28:02 GMT
Authorization: SharedKey azureinsiders:Oya7S8vBgOqBTBaKU3AMSL8ljnpiE9XAJrLq7kD1HYA=
Host: azureinsiders.blob.core.windows.net
Content-Length: 232

<?xml version="1.0" encoding="utf-8"?>
<SignedIdentifiers>
  <SignedIdentifier>
    <Id>Revokable-12/18/2012 6:28:00 AM</Id>
    <AccessPolicy>
    <Start />
    <Expiry />
    <Permission>rw</Permission>
    </AccessPolicy>
  </SignedIdentifier>
</SignedIdentifiers>
```

Like the preceding example, the following code snippet creates a shared access policy—with the Windows Azure client library—that grants read and write permissions on a blob container that has a signature identifier of *Managers*. The signature identifier is allowed to be any string up to 64 characters in length.

```
// Alternatively, we can add the SAP policies to the container with a name:
String signatureIdentifier = "Revokable-" + DateTime.UtcNow;
var permissions = container.GetPermissions();
// NOTE: A container can have up to 5 SAP policies
permissions.SharedAccessPolicies.Add(signatureIdentifier,
   new SharedAccessBlobPolicy {
       Permissions = SharedAccessBlobPermissions.Read |
                     SharedAccessBlobPermissions.Write
   });
container.SetPermissions(permissions);
```

Note A blob container can have a maximum of five policies assigned to it at one time. If you attempt to create more than five access policies, the sixth will result in the service returning status code 400 (Bad Request).

The *SharedAccessPolicy* cannot specify what is already present in the signature identifier.

```
// This SharedAccessPolicy CAN'T specify what is already present in the Signature Identifier
sas = blob.GetSharedAccessSignature(new SharedAccessBlobPolicy {
    SharedAccessStartTime = start,
    SharedAccessExpiryTime = start.AddYears(10)
}, signatureIdentifier);
sasUri = blob.Uri + sas;
Process.Start("IExplore", sasUri);
```

Revoking SAS permissions

After the SAS has served its useful purpose, it may be necessary or desirable (as a precautionary measure) to revoke the granted permissions. This is accomplished by simply removing the *Signature-Identifier* from the collection and performing another HTTP *PUT* operation against the blob container's URI. There is really no difference between adding or deleting a stored SAS policy because they are both accomplished in an identical fashion: by simply providing a complete list of the permissions.

```
PUT http://azureinsiders.blob.core.windows.net/demo
     ?restype=container&comp=acl&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Tue, 18 Dec 2012 06:28:06 GMT
Authorization: SharedKey azureinsiders:jwtmE3xzpCu7xo/TTqJVWCZVJeO19MnXwNoPQMYdbCI=
Host: azureinsiders.blob.core.windows.net
Content-Length: 62

<?xml version="1.0" encoding="utf-8"?>
<SignedIdentifiers />
```

With the Windows Azure client library, the container's signature identifier can be used to revoke the SAS permission as follows.

```
// We can now revoke access on the container:
permissions.SharedAccessPolicies.Remove(signatureIdentifier);
container.SetPermissions(permissions);
Process.Start("IExplore", sasUri); // This fails now
```

Blob attributes and metadata

Containers and blobs support two types of ancillary data: system properties and user-defined metadata. System properties exist on every blob and blob container. Some properties are read-only, whereas others can be set. A few of them correspond to specific standard HTTP headers, which the Windows Azure SDK will maintain for you. User-defined metadata allows you to define supplementary name-value dictionary information about the blob container or the blob. As its name implies, metadata should be used to store data about your data (not the data itself). This can sometimes be a matter of perspective.

Blob containers have attributes that describe both the containers' URI and Name. Each container has two read-only properties: *LastModifiedUtc*, which is the UTC time that the blob container was last updated; and *ETag*, which is a version number used for optimistic concurrency. Blob containers also provide 8 KB of customizable metadata in the form of a name-value pair dictionary, which you can use for your own purposes. Metadata should be used to store information about the blob or the blob's container. For example, you might store the name of the person who last read the contents of a blob in its metadata, and possibly the date and time the access was made.

The following HTTP *PUT* request demonstrates how to set metadata values. The query string *comp=metadata* sets up the operation. The values for the metadata are transmitted via HTTP *x-ms-meta-*<name> headers, where <*name*> is the name you are giving your metadata, and the value of the HTTP header is the value of your named metadata. The following example shows this.

```
PUT http://azureinsiders.blob.core.windows.net/demo/ReadMe.txt?comp=metadata&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-meta-CreatedBy: Paul
x-ms-meta-SourceMachine: DILITHIUM
x-ms-date: Tue, 18 Dec 2012 07:05:23 GMT
Authorization: SharedKey azureinsiders:02TWst4Dx5Qgr0zq31w7AvENcc+0ez06+HobJ4qZMlY=
Host: azureinsiders.blob.core.windows.net
Content-Length: 0
```

The following code demonstrates how to use the Windows Azure client library to store metadata containing the person's name and the machine that person was using when they created the blob.

```
container.SetPermissions(new BlobContainerPermissions() {
PublicAccess = BlobContainerPublicAccessType.Container });
CloudBlob blob = container.GetBlobReference("ReadMe.txt");
blob.UploadText("This is some text");
blob.Attributes.Metadata["CreatedBy"] = "Paul";
blob.Metadata["SourceMachine"] = Environment.MachineName;
blob.SetMetadata();
// NOTE: SetMetadata & SetProperties update the blob's ETag & LastModifiedUtc
```

When you launch Internet Explorer on the blob's URL, the contents of the blob are displayed. Similar to what you did previously, you can launch a browser directly against the blob container's URL, passing the filtration criteria *?restype=container&comp=list&include=metadata* in the query string. The following code will launch Internet Explorer to list the contents of the blob container, including its properties and any metadata.

```
// Get blobs in container showing each blob's properties & metadata
// See http://msdn.microsoft.com/en-us/library/dd135734.aspx for more options
Process.Start("IExplore", container.Uri +
          "?restype=container&comp=list&include=metadata").WaitForExit();
```

You can also retrieve a list of blobs, including properties and metadata, programmatically using the Windows Azure client library. First, pass an instance of *BlobRequestOptions* with its *BlobListing-Details* property set to *BlobListingDetails.Metadata*. Then call the *FetchAttributes* method on the blob proxy. The *FetchAttributes* method will include properties and metadata.

```
container.ListBlobs(new BlobRequestOptions {
    BlobListingDetails = BlobListingDetails.Metadata });
blob.FetchAttributes();
```

The following code then loops through the collection of the blob's properties and metadata and displays the corresponding value for each.

```
// Read the blob's attributes (which include properties & metadata)
blob.FetchAttributes();
BlobProperties p = blob.Properties;
Console.WriteLine("Blob's metadata (LastModifiedUtc={0}, ETag={1})",
         p.LastModifiedUtc, p.ETag);

Console.WriteLine("   Content Type={0}, Encoding={1}, Language={2}, MD5={3}",
   p.ContentType, p.ContentEncoding, p.ContentLanguage, p.ContentMD5);
Console.WriteLine();
foreach (String keyName in blob.Metadata.Keys)
   Console.WriteLine("   {0} = {1}", keyName, blob.Metadata[keyName]);
```

Conditional operations

It is often desirable to perform an operation on data only when particular conditions can be satisfied. It's better still when such operations can be conditionally performed by the data storage service rather than burden the client application, because doing so reduces the time and costs associated with transporting data to the application. Filtration operations limit the consumable data to a subset of the complete set of data available, so transporting all of the data across the wire simply to discard portions of that data upon evaluation wastes time, bandwidth, and ultimately money.

Although the evaluation can often be done by the application on the client side of the wire, multiple requests may be necessary to retrieve and evaluate data. In addition to the cost and time of transmitting large volumes of unnecessary data, the elapsed time also increases the probability of a data collision when one application attempts to perform an operation on an entity, but before that operation can take place, another application performs a successful operation on it, which renders the first application's copy of the entity as stale. Most blob operations can be performed conditionally based on their date of modification, or their *ETag*.

Conditional operations using REST

Conditional operations are implemented in blob storage by including one of four optional HTTP headers in the request, including a corresponding date and time or *ETag* value, as shown in Table 5-4.

TABLE 5-4 Conditional operation HTTP headers

HTTP header	Specified value
If-Modified-Since	*DateTime*
If-Unmodified-Since	*DateTime*
If-Match	*ETag* or wildcard (*)
If-None-Match	*ETag* or wildcard (*)

Reading data conditionally using the If-Modified-Since header can save unnecessary network bandwidth and data processing time (as well as associated costs for data transmission) by only transmitting the data when it's modified. When the condition cannot be met, an HTTP status code is returned that indicates the reason the condition was not met. Table 5-5 lists these HTTP status codes.

TABLE 5-5 HTTP response codes returned for unsatisfied conditions

Conditional header	HTTP response codes when condition is not met	
If-Modified-Since	304	Not Modified
If-Unmodified-Since	412	Precondition Failed
If-Match	412	Precondition Failed
If-None-Match	304	Not Modified

Conditional operations using the Windows Azure client library

The Windows Azure client library provides a convenient programming grammar for performing conditional operations. This grammar abstracts the setting of the underlying HTTP header to give a more comfortable and intuitive programming model to the developer. To explore conditional operations, you first need a blob in storage. Given a *CloudStorageAccount* credential object, the following code snippet sets up a proxy to blob storage, creates a blob named *Data.txt*, and uploads some data (the string *"Data"*) into that blob. It also sets up a retry policy, which you will see later in this chapter, and establishes timeouts.

```
// Create a blob and attach some metadata to it:
CloudBlobClient client = account.CreateCloudBlobClient();

// No retry for 306 (Unused), 4xx, 501 (Not Implemented), 505 (HTTP Version Not Supported)
client.RetryPolicy = new ExponentialRetry();

// Time server can process a request (default = 90 secs)
client.ServerTimeout = TimeSpan.FromSeconds(90);
```

```
// Time client can wait for response across all retries (default = disabled)
client.MaximumExecutionTime = TimeSpan.FromSeconds(5);

CloudBlobContainer container = client.GetContainerReference("demo").EnsureExists();
CloudBlockBlob blob = container.GetBlockBlobReference("Data.txt");
using (var stream = new MemoryStream("Data".Encode())) {
    blob.UploadFromStream(stream);
}
```

Conditional reads

Now that your test blob has been uploaded to storage, you can create an instance of *BlobRequest-Options* and set its *AccessCondition* property to an appropriate value to try various conditional means of retrieving it. The values of this property correspond directly with the HTTP headers shown in Table 5-4. Let's say that you want to retrieve the contents of the blob but only if that content has been updated. You may have a copy of the blob you've cached, and you don't want to waste valuable resources continuously re-fetching the same data you already have. You want to expend re-sources only when there is something new to retrieve. You can accomplish this using the *IfModified-Since* method. First, you want to simulate what happens when the blob has not been updated by another party, so you pass the *LastModifiedUtc* property of the blob to the *IfModifiedSince* method, knowing that this condition could never be met and that you will deliberately fail, as depicted in the following code.

```
// Download blob content if newer than what we have:
try {
    blob.DownloadText(
        AccessCondition.GenerateIfModifiedSinceCondition(
            blob.Properties.LastModified.Value)); // Fails
}
catch (StorageException ex) {
    Console.WriteLine(String.Format("Failure: Status={0}({0:D}), Msg={1}",
       (HttpStatusCode)ex.RequestInformation.HttpStatusCode,
        ex.RequestInformation.HttpStatusMessage));
}
```

You can do the inverse of the previous example by reading a blob only if its contents have not been modified since a specified date using the *IfNotModifiedSince* static method of the *AccessCondition* class. You might do this as part of a process for archiving an older date. Here is the code for this.

```
// Download blob content if more than 1 day old:
try {
    blob.DownloadText(
        AccessCondition.GenerateIfNotModifiedSinceCondition(
            DateTimeOffset.Now.AddDays(-1))); // Fails
}
catch (StorageException ex) {
    Console.WriteLine(String.Format("Failure: Status={0}({0:D}), Msg={1}",
       (HttpStatusCode)ex.RequestInformation.HttpStatusCode,
        ex.RequestInformation.HttpStatusMessage));
}
```

Conditional updates

You can perform updates conditionally, too. For example, many applications require optimistic concurrency when updating data. You want to replace an existing blob's contents, but only if someone hasn't updated the blob since you last retrieved it. If the blob was updated by another party, consider your copy of the blob to be stale and handle it according to your application's business logic for a concurrency collision. You accomplish this by using the static *IfMatch* method of the *AccessCondition* class to conditionally perform an action only if the properties match. If no updates are made to the target blob, the *ETag* properties of two blobs are identical and the update succeeds. If an update has occurred to the targeted blob (for example, the *ETag* properties do not match), a *StorageClient-Exception* exception is thrown.

```
// Upload new content if the blob wasn't changed behind your back:
try {
    blob.UploadText("Succeeds",
        AccessCondition.GenerateIfMatchCondition(blob.Properties.ETag)); // Succeeds
}
catch (StorageException ex) {
    Console.WriteLine(String.Format("Failure: Status={0}({0:D}), Msg={1}",
         (HttpStatusCode)ex.RequestInformation.HttpStatusCode,
          ex.RequestInformation.HttpStatusMessage));
}
```

When contention is encountered in an optimistic concurrency scenario, the usual countermeasure is to catch the exception, notify the requesting user or application of the contention, and then offer the option of fetching a fresh copy of the data. Generally, this means that the user has lost his revisions and must reapply his edits to the fresh copy before re-attempting to save his changes.

Another common application requirement is to create a blob in storage, but only if the blob doesn't already exist. You can use the asterisk wildcard character to match on any value. In the following code, when no properties match anything (for example, the blob does not already exist), you proceed with uploading. If the blob already exists, a *StorageClientException* is thrown. Generally, in production code, you should catch this exception and handle the situation according to the specific requirements of your application.

```
// Upload your content if it doesn't already exist:
try {
    // Fails
    blob.UploadText("Fails", AccessCondition.GenerateIfNoneMatchCondition("*"));
}
catch (StorageException ex) {
    Console.WriteLine(String.Format("Failure: Status={0}({0:D}), Msg={1}",
        (HttpStatusCode)ex.RequestInformation.HttpStatusCode,
         ex.RequestInformation.HttpStatusMessage));
}
```

Blob leases

Windows Azure storage provides a locking mechanism called a *lease* for preventing multiple parties from attempting to write to the same blob. A *blob lease* provides exclusive write access to the blob. After a lease is acquired, a client must include the active lease ID with the write request. The client has a one-minute window from the time the lease is acquired to complete the write; however, the lease can be continuously renewed to extend this time indefinitely to meet your application's needs.

A lease request may be performed in one of four modes:

- **Acquire** Request a new lease.
- **Renew** Renew an existing lease.
- **Release** Release the lease, which allows another client to immediately acquire a lease on the blob.
- **Break** End the lease, but prevent other clients from acquiring a new lease until the current lease period expires.

Taking a lease on a blob can also be used as a very convenient and inexpensive locking semantic for other cloud operations. For example, one Windows Azure Cloud Services instance might take a lease on a blob as means of signaling other instances that the resource is busy. Other instances would then be required to check for the existence of a lease before proceeding with a competing operation. If a lock is present, the competing operation can be held in a loop until the blob lock is released. When used in this manner, the blob lease acts as a traffic light. The competing instances stop when the traffic light is red (for example, the lock is present) and proceed when it turns green (for example, the lock was removed).

The following HTTP *PUT* request demonstrates how to acquire a lease on a blob. The query string *comp=lease* sets up the operation. The action to be taken on the lease is transmitted via the x-ms-lease-action HTTP header, as the following example demonstrates.

```
PUT http://azureinsiders.blob.core.windows.net/demo/test.txt
    ?comp=lease&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-lease-action: acquire
x-ms-lease-duration: 30
x-ms-date: Wed, 26 Dec 2012 08:36:22 GMT
Authorization: SharedKey azureinsiders:YsBInVJ6NhkAIgkuUk9647JbpEthVnZR1WAOcYRM1Hc=
Host: azureinsiders.blob.core.windows.net
Content-Length: 0
```

In the following Windows Azure client library code, you call the *AcquireLease* method on a blob and then show the blob's contents in Internet Explorer to prove you can perform reads against blobs that have leases. Immediately after, you attempt to acquire a second lease on the same blob. Always wrap your attempts to acquire a lease inside of a try/catch block, and always confirm that the *Web-Exception*'s status code is a Conflict status code, because the *WebException* could have been caused by any one of a number of possible conditions.

```
String leaseId = blob.AcquireLease(TimeSpan.FromSeconds(30), null);

// Succeeds: reads are OK while a lease is obtained
Process.Start("IExplore", blob.Uri.ToString()).WaitForExit();

// Try to acquire another lease:
try {
    leaseId = blob.AcquireLease(TimeSpan.FromSeconds(30), null);
}
catch (StorageException ex) {
    if ((HttpStatusCode)ex.RequestInformation.HttpStatusCode !=
          HttpStatusCode.Conflict) throw;
    Console.WriteLine(ex.RequestInformation.HttpStatusMessage);
}
```

You next demonstrate that the lease prevents writing to the blob until you supply a valid lease ID.

```
// Fails: Writes are not OK while a lease is held:
try {
    blob.UploadText("Can't upload while lease held without lease ID");
}
catch (StorageException ex) {
    if ((HttpStatusCode)ex.RequestInformation.HttpStatusCode !=
          HttpStatusCode.Conflict) throw;
    Console.WriteLine(ex.RequestInformation.HttpStatusMessage);
}
// Succeeds: Writes are OK if we specify a lease Id:
try {
    leaseId = blob.AcquireLease(TimeSpan.FromSeconds(30), null);
}
catch { } // Ensure we have the lease before doing the PUT
```

To release the lease on a blob via the RESTful API, simply execute another HTTP *PUT* against the blob's URI by using a query string parameter *comp=lease*, and set *x-ms-lease-action* to *Release*, as shown here.

```
PUT http://azureinsiders.blob.core.windows.net/demo/test.txt
    ?comp=lease&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-lease-id: a9053786-c96e-4bb9-8711-477fbd7fcb03
x-ms-lease-action: release
x-ms-date: Sun, 30 Dec 2012 02:29:22 GMT
Authorization: SharedKey azureinsiders:LJEakr84hNV6nStrTAgBzZdo3k50GlHy3f2syVuRrEM=
Host: azureinsiders.blob.core.windows.net
Content-Length: 0
```

To release a blob using the Windows Azure client library, call the *ReleaseLease* method and pass in the lease ID, as shown in this code snippet.

```
blob.ReleaseLease(AccessCondition.GenerateLeaseCondition(leaseId));
Process.Start("IExplore", blob.Uri.ToString()).WaitForExit();
```

Finally, after you have released your lease on the blob, you can update its contents to demonstrate how the lease is no longer preventing writing to the blob. You bring the contents up in the browser to visually verify that the contents were updated.

```
          blob.UploadText("Data uploaded while lease NOT held");
          Process.Start("IExplore", blob.Uri.ToString()).WaitForExit();
```

Using block blobs

As you learned earlier in this chapter, block blobs segment your data into chunks, or blocks. The size of one of these chunks is 4 MB or smaller.

When you upload data using the block semantics, you must provide a block ID, which is stored with your data; the stream that you are uploading your data from; and an MD5 hash of your data that is used to verify the successful transfer but is not stored. Uploaded blocks are stored in an uncommitted state. After uploading all of the blocks and calling the *Commit* method, the uncommitted blobs become committed in an atomic transaction. There is a restriction that the final blob be no greater than 200 GB after it is committed. An exception is thrown if this value is exceeded. If you don't commit an uploaded blob within seven days, Windows Azure storage deletes them.

The block ID is an array of 64 bytes (or fewer) that is base64-encoded for transport over the HTTP protocol.

Another useful characteristic of block blobs is that they can be uploaded in parallel to increase throughput, providing you have unused CPU power and available network bandwidth.

In the following code, you create an array of three strings (*A*, *B*, and *C*) that represents three distinct single-character blocks of data that you want to place in blob storage. You encode this array of strings into a memory stream using UTF8 encoding and then, for every block, you call the *PutBlock* method, passing your block ID, the stream containing your data, and an MD5 hash of the data being put into blob storage.

Hashes must be calculated and blocks must be apportioned before they can be stored, so you will start with the Windows Azure client library code for this example, and then look at the RESTful representation of this code immediately thereafter.

```
// Put 3 blocks to the blob verifying data integrity
String[] words = new[] { "A ", "B ", "C " };
var MD5 = new MD5Cng();
for (Int32 word = 0; word < words.Length; word++) {
    Byte[] wordBytes = words[word].Encode(Encoding.UTF8);
    // Azure verifies data integrity during transport; failure=400 (Bad Request)
    String md5Hash = MD5.ComputeHash(wordBytes).ToBase64String();
    blockBlob.PutBlock(word.ToBlockId(), new MemoryStream(wordBytes), md5Hash);
}
```

Execution of the preceding code causes three HTTP *PUT* requests to be made against data storage—one for each of the three blocks containing the data *A*, *B* and *C*. The *comp=block* parameter controls the kind of blob you are updating, and the *blockid*=<blockid> (where the *<blockid>* represents a unique block identifier).

```
PUT http://azureinsiders.blob.core.windows.net/demo/MyBlockBlob.txt
    ?comp=block&blockid=MDAwMDA%3D&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
Content-MD5: ZOO5/MV88bFp+072x6lV0g==
x-ms-date: Sun, 30 Dec 2012 03:02:12 GMT
Authorization: SharedKey azureinsiders:tM/FIZZnnzdb1fFIjgD+hb/wiHHOFyFvGN1JPx82TPo=
Host: azureinsiders.blob.core.windows.net
Content-Length: 2
Connection: Keep-Alive

A

HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-MD5: ZOO5/MV88bFp+072x6lV0g==
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 34e2df15-088f-46e1-af4e-6c10ef390940
x-ms-version: 2012-02-12
Date: Sun, 30 Dec 2012 03:02:15 GMT

0

PUT http://azureinsiders.blob.core.windows.net/demo/MyBlockBlob.txt
    ?comp=block&blockid=MDAwMDE%3D&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
Content-MD5: CUf4UWGwWRnZaUDz3hSFLg==
x-ms-date: Sun, 30 Dec 2012 03:02:14 GMT
Authorization: SharedKey azureinsiders:6ednOFFuqGuoe3qt9cmMtYD6OkLChpFFddBm3BEtx9k=
Host: azureinsiders.blob.core.windows.net
Content-Length: 2

B

HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-MD5: CUf4UWGwWRnZaUDz3hSFLg==
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: ca5790b6-d889-43d5-ab24-a21a21617fbf
x-ms-version: 2012-02-12
Date: Sun, 30 Dec 2012 03:02:16 GMT

0
```

```
PUT http://azureinsiders.blob.core.windows.net/demo/MyBlockBlob.txt
    ?comp=block&blockid=MDAwMDI%3D&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
Content-MD5: Q600VNgdC/J0zJ/wCXMiQw==
x-ms-date: Sun, 30 Dec 2012 03:02:15 GMT
Authorization: SharedKey azureinsiders:uc4Ttlza/fqoi5lYtXpYKqbhTLQoeojDOnvAGFf5OC8=
Host: azureinsiders.blob.core.windows.net
Content-Length: 2

C

HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-MD5: Q600VNgdC/J0zJ/wCXMiQw==
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 90adf2d6-7ea9-4b58-90a9-41d23e47de36
x-ms-version: 2012-02-12
Date: Sun, 30 Dec 2012 03:02:18 GMT

0
```

All three uploaded blocks of your block blob remain in an uncommitted state until you call *Commit* on them. You can verify this by downloading a list of uncommitted block IDs using the *DownLoad-BlockList* method and passing in a filter of *Uncommitted*. Two other values in the *BlockListingFilter* enumeration—*All* and *Committed*—allow you to download a list of all blocks or only those blocks that have been committed, respectively.

```
Console.WriteLine("Blob's uncommitted blocks:");
foreach (ListBlockItem lbi in blockBlob.DownloadBlockList(BlockListingFilter.Uncommitted))
    Console.WriteLine("   Name={0}, Length={1}, Committed={2}",
       lbi.Name.FromBlockId(), lbi.Length, lbi.Committed);
// Fails
try {
    blockBlob.DownloadText();
}
catch (StorageException ex) {
    Console.WriteLine(String.Format("Failure: Status={0}({0:D}), Msg={1}",
        (HttpStatusCode)ex.RequestInformation.HttpStatusCode,
         ex.RequestInformation.HttpStatusMessage));
}
```

Executing the preceding code demonstrates that the three uploaded blobs all have an uncommitted status and any attempt to download the uncommitted blob will result in failure.

```
Blob's uncommitted blocks:
   Name=0, Size=2, Committed=False
   Name=1, Size=2, Committed=False
   Name=2, Size=2, Committed=False
Failure: Status=NotFound, Msg=The specified blob does not exist.
```

You can request a list of all uncommitted blobs directly by sending an HTTP *GET* request to the blob's URI, the query string parameter *comp=blocklist*, and *blocklisttype=Uncommitted*, as shown here.

```
GET http://azureinsiders.blob.core.windows.net/demo/MyBlockBlob.txt
    ?comp=blocklist&blocklisttype=Uncommitted&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Sun, 30 Dec 2012 04:28:08 GMT
Authorization: SharedKey azureinsiders:zKdrwpdnKaQX8UKeq3COblqvMc3BDBhmmmDdWSuS4Wo=
Host: azureinsiders.blob.core.windows.net
```

You are not limited to placing blocks in storage one at a time, or even in the same order that you have them arranged. In fact, in some applications, it may even be desirable to upload the same block multiple times at different positions within the blob, or to completely change the order in which the blocks exist. Imagine scenarios in which blocks of data are reorganized based on sorting some element of their content.

```
// Commit the blocks in order (and multiple times):
blockBlob.PutBlockList(new[] { 0.ToBlockId(), 0.ToBlockId(), 1.ToBlockId(), 2.ToBlockId() });
// Succeeds
try {
     blockBlob.DownloadText();
}
catch (StorageException ex) {
    Console.WriteLine(String.Format("Failure: Status={0}({0:D}), Msg={1}",
         (HttpStatusCode)ex.RequestInformation.HttpStatusCode,
         ex.RequestInformation.HttpStatusMessage));
}

Console.WriteLine("Blob's committed blocks:");
foreach (ListBlockItem lbi in blockBlob.DownloadBlockList())
    Console.WriteLine("   Name={0}, Length={1}, Committed={2}",
       lbi.Name.FromBlockId(), lbi.Length, lbi.Committed);
```

Executing the preceding code commits the changes and produces the following results, confirming that the blobs are now all committed and that the *A* blob was committed twice.

```
Blob's committed blocks:
   Name=0, Size=2, Committed=True
   Name=0, Size=2, Committed=True
   Name=1, Size=2, Committed=True
   Name=2, Size=2, Committed=True
A A B C
```

As you might anticipate from the pattern that is emerging, you can request a list of all committed blobs directly by sending an HTTP *GET* request to the blob's URI and the query string parameter *comp=blocklist*, and *blocklisttype=Committed* as follows.

```
GET http://azureinsiders.blob.core.windows.net/demo/MyBlockBlob.txt
    ?comp=blocklist&blocklisttype=Committed&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Sun, 30 Dec 2012 21:52:11 GMT
Authorization: SharedKey azureinsiders:AMEqUfAuSg6oub4/zz+aE5LB3S6qbXxSNLip0oCNOxs=
Host: azureinsiders.blob.core.windows.net
```

Blocks might represent discrete segments of data that are organized like scenes in a movie, where you want to delete some scenes and change the order of others. The block blob API supports this kind of functionality. You can delete blocks just by excluding their *BlockIDs* from the *BlockList* body of your request, and you can reorder your blocks by changing their order in the list. This is shown in the following HTTP request, which deletes block *0* and saves block *2* before block *1*. It may be a little hard to see this directly, because the *BlockIDs* are base64-encoded in the BlockList, but you can easily modify the sample code shown a little later to see how the body of the message is changed by the block order.

```
PUT http://azureinsiders.blob.core.windows.net/demo/MyBlockBlob.txt
    ?comp=blocklist&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-blob-content-type: application/octet-stream
Content-MD5: uO2OSdbs3agLOJthlv1b4w==
x-ms-date: Sun, 30 Dec 2012 22:03:32 GMT
Authorization: SharedKey azureinsiders:IcJMoA2vWMPVfWK0f2NzzpqwR5hi1JOuPB8poQef8D4=
Host: azureinsiders.blob.core.windows.net
Content-Length: 114
Connection: Keep-Alive

<?xml version="1.0" encoding="utf-8"?>
<BlockList>
    <Latest>MDAwMDI=</Latest>
    <Latest>MDAwMDE=</Latest>
</BlockList>
```

The following code snippet shows how you can delete blocks by excluding their *BlockIDs* when you call *PutBlockList using the Windows Azure client library*. In the following code snippet, you delete block *0* and your duplicate block *1*, and save block *2* before block *1*.

```
// You can change the block order & remove a block:
blockBlob.PutBlockList(new[] { 2.ToBlockId(), 1.ToBlockId() });
// Succeeds
try {
    blockBlob.DownloadText();
}
catch (StorageException ex) {
    Console.WriteLine(String.Format("Failure: Status={0}({0:D}), Msg={1}",
        (HttpStatusCode)ex.RequestInformation.HttpStatusCode,
         ex.RequestInformation.HttpStatusMessage));
}
```

After executing the preceding code, you can verify that block A was deleted and blob C appears before block B by executing an HTTP *GET* against the blob's URI, as shown here.

```
GET http://azureinsiders.blob.core.windows.net/demo/MyBlockBlob.txt?timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Mon, 31 Dec 2012 07:04:54 GMT
Authorization: SharedKey azureinsiders:NhZ/aPp3HEtB6tMyT1NMj4BD4LRvySi8YV5/1BfQAwk=
Host: azureinsiders.blob.core.windows.net
```

The full blob as committed is returned as the response to this request. You can see the *C B* content in the body of the message response.

```
HTTP/1.1 200 OK
Content-Length: 4
Content-Type: application/octet-stream
Last-Modified: Mon, 31 Dec 2012 07:04:53 GMT
Accept-Ranges: bytes
ETag: "0x8CFB53C446E71CD"
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 3523b2a0-1069-4e01-8f7a-8210f4e60612
x-ms-version: 2012-02-12
x-ms-lease-status: unlocked
x-ms-lease-state: available
x-ms-blob-type: BlockBlob
Date: Mon, 31 Dec 2012 07:04:54 GMT

C B
```

When you upload a blob that is greater than 32 MB, the *Upload*Xxx operations automatically break your upload up into 4-MB blocks, upload each block with *PutBlock*, and then commit all blocks with the *PutBlockList* method. The block size can be changed by modifying the *WriteBlockSizeInBytes* property of your client proxy (for example, your instance of *CloudBlobClient*), as shown in the following commented code block.

```
     // The client library can automatically upload large blobs in blocks:
     // 1. Define size of "large blob" (range=1MB-64MB, default=32MB)
     client.SingleBlobUploadThresholdInBytes = 32 * 1024 * 1024;

     // 2. Set individual block size (range=1MB-4MB, default=4MB)
     client.WriteBlockSizeInBytes = 4 * 1024 * 1024;

         // 3. Set # of blocks to simultaneously upload (range=1-64, default=# CPUs)
         client.ParallelOperationThreadCount = Environment.ProcessorCount;
```

Using page blobs

Page blobs add the features of random access and sparse population to the blob storage story, and they quintuple the maximum size from 200 MB to 1 terabyte. Page blobs were added to Windows Azure storage to enable development of the VHD virtual drive abstraction. They are also useful in fixed-length logging scenarios, where rolling overwrite of the oldest data in the log may be desired.

The sparse feature is nice because it allows you to allocate a storage amount up to 1 terabyte, but you are charged only for the pages that you place in the blob, no matter how much storage space you allocate. When reading and writing page blobs, you're required to read and write your data in page-sized chunks that begin on a page boundary.

In the following sample code, you create an array of 5 bytes (integers 1 through 5), and you write that data to page 0. Next, you create an array of 5 bytes (descending integers 5 through 1), and you write that data to page 2. There is no significance to the integers I selected for this example beyond demonstrating that you can read and write data in sparsely populated pages that begin on page boundaries.

```
const Int32 c_BlobPageSize = 512;
Console.Clear();
CloudBlobClient client = account.CreateCloudBlobClient();
CloudBlobContainer container = client.GetContainerReference("demo").EnsureExists();
CloudPageBlob pageBlob = container.GetPageBlobReference("MyPageBlob.txt");
pageBlob.DeleteIfExists();

// You must create a page blob specifying its size:
pageBlob.Create(10 * c_BlobPageSize);

Byte[] data = new Byte[1 * c_BlobPageSize];  // Must be multiple of page size

// Write some data to Page 0 (offset 0):
Array.Copy(new Byte[] { 1, 2, 3, 4, 5 }, data, 5);
pageBlob.WritePages(new MemoryStream(data), 0 * c_BlobPageSize); // Offset 0
// Write some data to Page 2 (offset 1024):
Array.Copy(new Byte[] { 5, 4, 3, 2, 1 }, data, 5);
pageBlob.WritePages(new MemoryStream(data), 2 * c_BlobPageSize); // Offset 1024

// Show committed pages:
foreach (PageRange pr in pageBlob.GetPageRanges())
    Console.WriteLine("Start={0,6:N0}, End={1,6:N0}", pr.StartOffset, pr.EndOffset);

// Read the whole blob (with lots of 0's):
using (var stream = new MemoryStream()) {
    pageBlob.DownloadToStream(stream);
    data = stream.GetBuffer();
}
Console.WriteLine("Downloaded length={0:N0}", data.Length);
Console.WriteLine(
    "  Page 0 data: " + BitConverter.ToString(data, 0 * c_BlobPageSize, 10));
Console.WriteLine(
    "  Page 1 data: " + BitConverter.ToString(data, 1 * c_BlobPageSize, 10));
Console.WriteLine(
    "  Page 2 data: " + BitConverter.ToString(data, 2 * c_BlobPageSize, 10));
```

At this point of execution, you have only two 512-byte pages in a 10-page, sparsely populated blob (pages 4–9 would look identical to pages 1 and 3). The first page blob starts at byte 0 and ends at byte 511, and the second one starts at byte 1024 and ends at byte 1535. When you loop through the pages of the blob and display the bytes that are in each page, you can see that pages 0 and 2 contain the bytes you uploaded, whereas pages 1, 3, and 4–10 all return zeros. You are being charged only for the two pages you stored, but the blob behaves as if all 10 pages were populated.

From this output, you might be tempted to think that uploading a page of zeros would be treated as a nonexistent page. Unfortunately, it would be incorrect to make such an assumption. A page must be cleared from the collection of pages in a blob in order for it to return to a nonexistent sparse state and for you to avoid being billed. You use the *ClearPages* method of the page blob in the code that follows to do this. Pages of zeros are considered part of the blob's official population, and you will be billed for their storage.

Over the network, the data is transmitted as HTTP *PUT* requests with the query string parameter *comp=page*, the HTTP header *x-ms-range* indicating the byte range, and *x-ms-page-write* indicating the type of operation being performed (*Update* in this example).

```
PUT http://azureinsiders.blob.core.windows.net/demo/MyPageBlob.txt
?comp=page&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-range: bytes=0-511
x-ms-page-write: Update
x-ms-date: Mon, 31 Dec 2012 07:45:32 GMT
Authorization: SharedKey azureinsiders:MLh8U/RxECpksvMuHdgzn2KDOQ6CSUHlku4NJPb7MJI=
Host: azureinsiders.blob.core.windows.net
Content-Length: 512<unprintable data>
```

After writing the pages to blob storage, as shown in the preceding code, there are two 512-byte pages in a sparsely populated 5,120-byte page blob.

```
Start=0, End=511
Start=1024, End=1535
Downloaded length=5,120
  Page 0 data: 01-02-03-04-05-00-00-00-00-00
  Page 1 data: 00-00-00-00-00-00-00-00-00-00
  Page 2 data: 05-04-03-02-01-00-00-00-00-00
  Page 3 data: 00-00-00-00-00-00-00-00-00-00
```

You can continue modifying blobs by page. The following code shows reading a specific page range, clearing a set of pages, and then committing those changes.

```
// Read a specific range from the blob (offset 1024, 10 bytes):
using (var stream = new MemoryStream()) {
    pageBlob.DownloadRangeToStream(stream, 2 * c_BlobPageSize, 10);
    stream.Seek(0, SeekOrigin.Begin);
    data = new BinaryReader(stream).ReadBytes((Int32)stream.Length);
    Console.WriteLine("  Page 2 data: " + BitConverter.ToString(data, 0, 10));
}

// Clear a range of pages (offset 0, 512 bytes):
pageBlob.ClearPages(0, 512);

// Show committed pages:
foreach (PageRange pr in pageBlob.GetPageRanges())
    Console.WriteLine("Start={0,6:N0}, End={1,6:N0}", pr.StartOffset, pr.EndOffset);
```

After clearing the bytes in page 0 with the *ClearPages* method, only page 2 remains committed in this page blob. You use the *GetPageRanges* method to return a list of pages and then iterate over the result to prove that there is only one remaining with a starting position of 1,024 and an ending position of 1,535.

Blob snapshots

Blob storage supports a very powerful feature called snapshots. Like the name might imply, *snapshots* operate in a manner similar to photographs. You can take as many snapshots of a family member over time as you want. The resulting set of photographic snapshots form a chronology of the changes to your subject over time. Likewise, a snapshot in blob storage is a record of a blob's state at a particular point in time. To maintain efficient use of storage, Windows Azure storage stores only the delta between the previous version and the current version of the blob. Although snapshots may be deleted, they are immutable and provide an audit trail of the changes as well as a convenient mechanism for rolling back changes.

This powerful feature provides a rich and cost-effective versioning mechanism for your documents, images, and other artifacts. Consider the number of times that business documents such as sales orders, purchase orders, or contracts might be revised before being agreed to. Consider also the desire that business people may have to keep snapshots of those documents for use in mitigating disputes.

In the RESTful API of Windows Azure storage, snapshots are identified by using an opaque value supplied as a query string parameter to the blob's URI. Although the documentation states that this is an opaque value and can therefore be changed without notice, it sure looks a lot like a time stamp! The following shows an example.

```
http://.../cotnr/blob.dat?snapshot=2011-05-14T18:25:53.6230000Z
```

Creating the original blob

To create the blob in storage for this example, issue an HTTP *PUT* request against blob storage as follows.

```
PUT http://azureinsiders.blob.core.windows.net/demo/Original.txt?timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-blob-type: BlockBlob
Content-MD5: vfup6ZfBE05+wonANUivGw==
x-ms-date: Mon, 31 Dec 2012 08:13:20 GMT
Authorization: SharedKey azureinsiders:7NDtxB5cV0SveodezKGFTe8NFWXGgIHgs294Aq6IrdI=
Host: azureinsiders.blob.core.windows.net
Content-Length: 13

Original data
```

Windows Azure storage will respond with an HTTP status code of 201 (Created) to confirm the successful upload of your data.

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-MD5: vfup6ZfBE05+wonANUivGw==
Last-Modified: Mon, 31 Dec 2012 08:13:22 GMT
ETag: "0x8CFB545D5F2B219"
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 579553cf-d37b-46aa-b1ed-8e4841993c02
x-ms-version: 2012-02-12
Date: Mon, 31 Dec 2012 08:13:20 GMT

0
```

You can perform this action using the Windows Azure client library by uploading contents into blob storage after acquiring a reference to the blob, as shown here.

```
         // Create the original page blob (512 1s & 2s):
         CloudBlob origBlob = container.GetBlobReference("Original.txt");
         origBlob.UploadText("Original data");
```

Creating the blob's snapshot

The value of the *SnapshotTime* property contains the date and time the snapshot was taken. The following code retrieves the *SnapshotTime* value, which you can use to identify this specific blob snapshot later on.

```
PUT http://azureinsiders.blob.core.windows.net/demo/Original.txt
    ?comp=snapshot&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Mon, 31 Dec 2012 08:13:20 GMT
Authorization: SharedKey azureinsiders:h164Br8QbritWbEiDG8OBnUjH8B/hxiFCQ1+ZIPqwRQ=
Host: azureinsiders.blob.core.windows.net
Content-Length: 0
```

Windows Azure responds to your snapshot request by returning another HTTP 201 (Created) status code similar to the following.

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Last-Modified: Mon, 31 Dec 2012 08:13:22 GMT
ETag: "0x8CFB545D5F2B219"
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 6e88ae98-a68a-46e5-a734-25a76a55e92e
x-ms-version: 2012-02-12
x-ms-snapshot: 2012-12-31T08:13:23.0482586Z
Date: Mon, 31 Dec 2012 08:13:21 GMT

0
```

You can perform the same action with the client library by calling the blob's *CreateSnapshot* method, which returns a cloud blob reference, and then caching the snapshot's *SnapshotTime* property (which returns the value of the HTTP *ETag* header).

```
         // Create a snapshot of the original blob & save its timestamp:
         CloudBlob snapshotBlob = origBlob.CreateSnapshot();
         DateTime? snapshotTime = snapshotBlob.SnapshotTime;
```

You can prove that the snapshot cannot be updated by attempting to upload new data into it and noting that it fails.

```
         // Try to write to the snapshot blob:
         try { snapshotBlob.UploadText("Fails"); }
         catch (ArgumentException ex) { Console.WriteLine(ex.Message); }
```

A similar attempt to upload data into the original blob, however, succeeds.

```
PUT http://azureinsiders.blob.core.windows.net/demo/Original.txt?timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-blob-type: BlockBlob
Content-MD5: zgYkAjIEXVBppjaEcT9nLg==
x-ms-date: Mon, 31 Dec 2012 08:13:21 GMT
Authorization: SharedKey azureinsiders:sRxSV/eOIdHBsOHSQU6QMmfY1pMZl5LsqNPuQGaQkVs=
Host: azureinsiders.blob.core.windows.net
Content-Length: 8

New data

Resulting in another HTTP result status code of 201 (Created):

HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-MD5: zgYkAjIEXVBppjaEcT9nLg==
Last-Modified: Mon, 31 Dec 2012 08:13:23 GMT
ETag: "0x8CFB545D6138160"
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: f9e86330-6de5-402a-b592-0e65abb75c03
x-ms-version: 2012-02-12
```

```
Date: Mon, 31 Dec 2012 08:13:21 GMT

0
```

Of course, you can perform this same operation using the client library. You can also add code to perform a comparison of the two normal and snapshot blobs for additional verification.

```
// Modify the original blob & show it:
origBlob.UploadText("New data");
Console.WriteLine(origBlob.DownloadText());      // New data
Console.WriteLine();

// Show snapshot blob via original blob URI & snapshot time:
snapshotBlob = container.GetBlockBlobReference(
"Original.txt", snapshotTime);
Console.WriteLine(snapshotBlob.DownloadText());  // Original data


// Show all blobs in the container with their snapshots:
foreach (ICloudBlob b in
    container.ListBlobs(null, true, BlobListingDetails.Snapshots)) {
    Console.WriteLine("Uri={0}, Snapshot={1}", b.Uri, b.SnapshotTime);
}
```

Listing snapshots

If you're using snapshots, you might find it necessary in your application to obtain a list of all of the snapshots that exist for your blobs. To retrieve a list of all blobs in the container, issue an HTTP *GET* command against the resource, passing the query string parameters *restype=container&comp=list&include=snapshots&timeout=90*.

```
GET http://azureinsiders.blob.core.windows.net/demo
    ?restype=container&comp=list&include=snapshots&timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-date: Mon, 31 Dec 2012 08:13:21 GMT
Authorization: SharedKey azureinsiders:xEkszk1RQ+WTGa7nfZfbP9QDQz8JWh7Fh7fPplxIvuE=
Host: azureinsiders.blob.core.windows.net

HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/xml
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 3f3d3919-aef4-4289-97c7-4fad8d655747
x-ms-version: 2012-02-12
Date: Mon, 31 Dec 2012 08:13:21 GMT

4D2
<?xml version="1.0" encoding="utf-8"?>
<EnumerationResults ContainerName="http://azureinsiders.blob.core.windows.net/demo">
  <Blobs>
    <Blob>
      <Name>Original.txt</Name>
      <Snapshot>2012-12-31T08:13:23.0482586Z</Snapshot>
      <Url>http://azureinsiders.blob.core.windows.net/demo/Original.txt
          ?snapshot=2012-12-31T08%3a13%3a23.0482586Z</Url>
      <Properties>
        <Last-Modified>Mon, 31 Dec 2012 08:13:22 GMT</Last-Modified>
        <Etag>0x8CFB545D5F2B219</Etag>
        <Content-Length>13</Content-Length>
        <Content-Type>application/octet-stream</Content-Type>
        <Content-Encoding />
        <Content-Language />
        <Content-MD5>vfup6ZfBE05+wonANUivGw==</Content-MD5>
        <Cache-Control />
        <BlobType>BlockBlob</BlobType>
      </Properties>
    </Blob>
    <Blob>
      <Name>Original.txt</Name>
      <Url>http://azureinsiders.blob.core.windows.net/demo/Original.txt</Url>
      <Properties>
        <Last-Modified>Mon, 31 Dec 2012 08:13:23 GMT</Last-Modified>
        <Etag>0x8CFB545D6138160</Etag>
        <Content-Length>8</Content-Length>
        <Content-Type>application/octet-stream</Content-Type>
        <Content-Encoding />
        <Content-Language />
        <Content-MD5>zgYkAjIEXVBppjaEcT9nLg==</Content-MD5>
        <Cache-Control />
        <BlobType>BlockBlob</BlobType>
        <LeaseStatus>unlocked</LeaseStatus>
        <LeaseState>available</LeaseState>
      </Properties>
    </Blob>
  </Blobs>
<NextMarker />
</EnumerationResults>
0
```

To retrieve a listing of blob snapshots in a container by using the Windows Azure client library, set an instance of the *BlobRequestOptions* class's *BlobListingDetails* property equal to *BlobListing-Details.Snapshots*, which is passed to the *ListBlobs* method of your blob container, as shown in the following code snippet.

```
// Show all blobs in the container with their snapshots:
foreach (ICloudBlob b in container.ListBlobs(null, true, BlobListingDetails.Snapshots))
{
    Console.WriteLine("Uri={0}, Snapshot={1}", b.Uri, b.SnapshotTime);
}
```

Running the preceding code produces the following output.

```
Original data
Uri=http://azureinsiders.blob.core.windows.net/demo/Original.txt,
    Snapshot=12/31/2012 8:13:23 AM +00:00
Uri=http://azureinsiders.blob.core.windows.net/demo/Original.txt, Snapshot=
Original data
```

As stated earlier, snapshots are an immutable record of your data taken at a given point in time. Snapshots, like photographs, may be read, copied, or deleted, but not altered. And just as you might produce an altered copy of an original photograph with a photo editing program, you can produce an altered copy of a blob with blob snapshots. Although you cannot update a blob snapshot, you can clone the snapshot to make another writeable blob that can then be modified. There is an efficiency implemented in the Windows Azure REST API through the use of an HTTP header. This efficiency allows creation of the snapshot clone to be done as a single transactional request; this avoids the requirement of one trip to fetch the source data and another to create the clone. To use this feature, the destination URI establishes the target resource to be updated, and the *x-ms-copy-source* header establishes the source of the data to be copied. The following HTTP *PUT* request shows an example of cloning the *Original.txt* blob to a blob called *Copy.txt*:

```
PUT http://azureinsiders.blob.core.windows.net/demo/Copy.txt?timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-copy-source: http://azureinsiders.blob.core.windows.net/demo/Original.txt
    ?snapshot=2012-12-31T08%3A13%3A23.0482586Z
x-ms-date: Mon, 31 Dec 2012 08:13:21 GMT
Authorization: SharedKey azureinsiders:TEhQVnfZF/kLbbTVYCpzQlCn5UHrF9QRBzSWKBOohz8=
Host: azureinsiders.blob.core.windows.net
Content-Length: 0
```

Upon successful execution of the *PUT* operation, Windows Azure returns an HTTP status code 201 (Created).

```
HTTP/1.1 202 Accepted
Transfer-Encoding: chunked
Last-Modified: Mon, 31 Dec 2012 08:13:23 GMT
ETag: "0x8CFB545D63BA3D7"
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 2b087f91-b3eb-4cc6-844f-b45e4de85aeb
x-ms-version: 2012-02-12
x-ms-copy-id: ebe42e0e-4094-4aff-a244-32a951f00562
x-ms-copy-status: success
Date: Mon, 31 Dec 2012 08:13:21 GMT

0
```

Using the Windows Azure client library, you can clone a snapshot of the blob by creating the target blob (in this case, *Copy.txt*), and then calling the *CopyFromBlob* method, passing an instance of the snapshot blob in as an argument. The following code demonstrates this technique.

```
// Create writable blob from the snapshot:
CloudBlockBlob writableBlob = container.GetBlockBlobReference("Copy.txt");
writableBlob.StartCopyFromBlob(snapshotBlob);
Console.WriteLine(writableBlob.DownloadText());
writableBlob.UploadText("Success");
Console.WriteLine(writableBlob.DownloadText());
```

Deleting snapshots

You can delete a blob and all of its snapshots, or you can delete individual snapshots from a blob; however, you are not allowed to delete a blob without also deleting all of its snapshots.

```
DELETE http://azureinsiders.blob.core.windows.net/demo/Original.txt?timeout=90 HTTP/1.1
x-ms-version: 2012-02-12
User-Agent: WA-Storage/2.0.0
x-ms-delete-snapshots: include
x-ms-date: Mon, 31 Dec 2012 08:13:21 GMT
Authorization: SharedKey azureinsiders:IAOrqcWDrjJUTBkM6jg+YKC6gLOCJvTTBVFai+wC8mQ=
Host: azureinsiders.blob.core.windows.net
```

Upon successful deletion of your blob and all of its snapshots, the Windows Azure storage service returns an HTTP status code 202 (Accepted) response.

```
HTTP/1.1 202 Accepted
Transfer-Encoding: chunked
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: f2c4f4a4-0319-4c27-a16a-e9bb28078fd6
x-ms-version: 2012-02-12
Date: Mon, 31 Dec 2012 08:13:21 GMT

0
```

When using the Windows Azure client library, snapshot deletion behavior is controlled by the *DeleteSnapshotsOption* property of an instance of the *BlobRequestOptions* class, which is passed as an argument to the *Delete* method of the blob to be deleted. An example of deleting a blob and its snapshots follows.

```
// DeleteSnapshotsOption: None (blob only; throws StorageException if snapshots exist),
// IncludeSnapshots, DeleteSnapshotsOnly
origBlob.Delete(DeleteSnapshotsOption.IncludeSnapshots);
```

Continuation tokens and blobs

A pagination mechanism for data that can be presented in a tabular format is often required because humans generally cannot digest more than a page of information at one time. Also, you want to ensure that the data being requested is appropriate for the query and not a mistake, because you are dealing with potentially massive databases in the cloud. It can take a lot of resources to compute and transmit billions of rows of data over the wire. Windows Azure must also take into account its own scalability; a large query can block or significantly impede many smaller queries, because those smaller queries may have to wait or compete for resources. *Continuation tokens,* as introduced in Chapter 4, allow Windows Azure storage to return a smaller subset of your data. A continuation token imposes upon you, however, to ask for subsequent pages of your data by passing you back a continuation token that you must then use to call back to retrieve subsequent pages of your query. Windows Azure storage refers to these pages of data as *segments*.

Blobs are not tabular data, so you do not have to anticipate continuation tokens when retrieving blobs; however, some of the API does return tabular data such as retrieving a list of blobs stored in a container. Because a container can have an unlimited number of entries, you should always anticipate that you may receive a continuation token back from any request that you make by calling an xxx-*Segmented* retrieval method.

```
Console.Clear();
CloudBlobClient client = account.CreateCloudBlobClient();

// ListContainers/BlobsSegmented return up to 5,000 items
const Int32 desiredItems = 5000 + 1000;  // Try to read more than the max

// Create a bunch of blobs (this takes a very long time):
CloudBlobContainer container = client.GetContainerReference("manyblobs");
container.EnsureExists();
for (Int32 i = 0; i < desiredItems; i++)
    container.GetBlockBlobReference(String.Format("{0:00000}", i)).UploadText("");

// Enumerate containers & blobs in segments
for (ContainerResultSegment crs = null; crs.HasMore(); ) {
    crs = client.ListContainersSegmented(
      null, ContainerListingDetails.None, desiredItems, crs.SafeContinuationToken());
    foreach (var c in crs.Results) {
        Console.WriteLine("Container: " + c.Uri);
        for (BlobResultSegment brs = null; brs.HasMore(); ) {
            brs = c.ListBlobsSegmented(null, true, BlobListingDetails.None,
                desiredItems, brs.SafeContinuationToken(), null, null);
            foreach (var b in brs.Results) Console.WriteLine("   Blob: " + b.Uri);
        }
    }
}
```

In the preceding code, the value of the *ContinuationToken* property of the *ResultSegment* will be either a continuation token or null when there are no more results to read. The following extension method, which is included in the Wintellect Azure Power Library, was used to simplify the programming model. It does this by supplying a Boolean indicator that controls exiting the for-loop when no more result segments are available.

```
public static BlobContinuationToken SafeContinuationToken(
          this ContainerResultSegment crs) {
   return (crs == null) ? null : crs.ContinuationToken;
}Blob request options
```

As used in the preceding code examples, instances of *BlobRequestOptions* may be passed as arguments to blob operations to augment the behavior of a request. The *BlobRequestOptions* class is an assortment of loosely related properties that are applicable during different kinds of operations. Only *Timeout* and *RetryPolicy* are used by all data access storage requests. Table 5-6 describes the properties of *BlobRequestOptions*.

TABLE 5-6 *BlobRequestOptions* properties and their impact on requests

***BlobRequestOptions* properties**	**Impact on requests**
Timeout	Used for all blob operations. The timespan allowed for a given operation to complete before a timeout error condition, which is handled according to the *RetryPolicy* property. See Chapter 4 for more information.
RetryPolicy	Used for all blob operations. Controls the retry policy used when accessing data. See Chapter 4 for more information.
AccessCondition	Used only when performing conditional operations. Controls the conditions for selecting data based on an *ETag* value (for example, *If-Match*, *If-Non-Match*, *If-Modified-Since*, and *If-Not-Modified-Since*).
CopySourceAccessCondition	Used only when performing conditional copy operations on blobs. Controls the conditions for selecting data based on an *ETag* value (for example, *If-Match*, *If-Non-Match*, *If-Modified-Since*, and *If-Not-Modified-Since*).
DeleteSnapshotsOption	Used only when performing delete operations on blobs. *IncludeSnapshots* deletes the blob and all of its snapshots; *DeleteSnapshotsOnly* deletes the snapshots only (leaving the blob); *None*.
BlobListingDetails	Used only when performing blob list operations. Controls the data that is included in the list. Options include the following: *All* lists all available committed blobs, uncommitted blobs, and snapshots, and returns all metadata for those blobs. ■ *Metadata* retrieves blob metadata for each blob returned in the listing. ■ *None* lists only committed blobs and does not return blob metadata. ■ *Snapshots* lists committed blobs and blob snapshots. ■ *UncommittedBlobs* lists committed and uncommitted blobs.
UseFlatBlobListing	Used only when performing blob list operations.

Conclusion

The purpose of this chapter was to provide you with a thorough understanding of how to securely perform data operations against the blob storage service. You were first introduced to how primary data operations are accomplished against the two kinds of blobs supported by Windows Azure storage and the organizations container structure that encapsulates them. You learned how to navigate this structure as if it was a hierarchical directory using the storage API and about the metadata that can be used to augment them.

This chapter then explained the special high-value use of public anonymous containers for supplying Internet-addressable content to your applications. Finally, you learned about many advanced features, such as conditional operations, leases, Shared Access Signatures, Shared Access Policies, snapshots, and continuation tokens.

Index

About the author

PAUL MEHNER has been a software developer, architect, project manager, consultant, speaker, mentor, instructor, and entrepreneur for more than three decades. He is cofounder of the South Sound .NET User Group, one of the oldest recorded .NET user groups in the world, and was one of the earliest committee members of the International .NET Association (INETA). He also works for Wintellect as a Senior Consultant and Trainer.

Currently, Paul specializes in cloud computing on the Windows Azure platform, Service Oriented Architectures, Security Token Servers, Windows Communication Foundation, Windows Identity Foundation, and Windows Workflow Foundation. Prior to being reborn as a .NET protagonist in 2000, Paul's experience included more than 20 years supporting many flavors of the UNIX operating system. Paul began his early computing career in 1977 on a homebuilt breadboard computer with 256 bytes of RAM, 12 toggle switches, 9 light-emitting diodes, and an RCA CDP1802 microprocessor.