

Microsoft

DEVELOPING APPLICATIONS FOR THE CLOUD

on the Microsoft Windows®
Azure™ Platform

Dominic Betts
Scott Densmore
Ryan Dunn
Masashi Narumoto
Eugenio Pace
Matias Woloski



patterns & practices

^

DEVELOPING APPLICATIONS FOR THE CLOUD

Developing Applications for the Cloud

on the Microsoft® Windows Azure™ Platform

Authors

Dominic Betts
Scott Densmore
Ryan Dunn
Masashi Narumoto
Eugenio Pace
Matias Woloski

ISBN: 978-0-7356-5606-2

This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it. Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

© 2010 Microsoft. All rights reserved.

Microsoft, Microsoft Dynamics, Active Directory, Active Directory Federation Services, IntelliTrace, MSDN, Sharepoint, SQL Azure, Visual C#, Visual Studio, Windows, Windows Azure, Windows Identity Foundation, Windows Live, and Windows Server are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Contents

FOREWORD

Bharat Shyam xi

PREFACE

Who This Book Is For xiii

Why This Book Is Pertinent Now xiv

How This Book Is Structured xv

What You Need to Use the Code xvi

Who's Who xvii

1 The Tailspin Scenario 1

The Tailspin Company 1

Tailspin's Strategy 1

The Surveys Application 2

Tailspin's Goals and Concerns 3

The Surveys Application Architecture 5

2 Hosting a Multi-Tenant Application on Windows Azure 7

Single-Tenant vs. Multi-Tenant 7

Multi-Tenancy Architecture in Azure 8

Selecting a Single-Tenant or Multi-Tenant Architecture 9

Architectural Considerations 9

Application Stability 10

Making the Application Scalable 10

Service Level Agreements 10

The Legal and Regulatory Environment 11

Handling Authentication and Authorization 11

Application Life Cycle Management Considerations	11
Maintaining the Code Base	11
Handling Application Upgrades	12
Monitoring the Application	12
Using .NET Providers and Third-Party Components	12
Provisioning for Trials and New Customers	13
Customizing the Application	13
URLs to Access the Application	13
Customizing the Application by Tenant	13
Multi-Tenant Data Architecture	14
Protecting Data from Other Tenants	14
Data Architecture Extensibility	15
Data Architecture Scalability	15
Financial Considerations	16
Billing Customers	16
Managing Application Costs	17
3 Accessing the Surveys Application	19
DNS Names, Certificates, and SSL in the Surveys Application	19
Web Roles in the Surveys Application	19
Goals and Requirements	20
Overview of the Solution	20
Inside the Implementation	21
Geo-Location	23
Goals and Requirements	23
Overview of the Solution	24
Authentication and Authorization	25
Goals and Requirements	26
Overview of the Solution	26
Inside the Implementation	29
Protecting Session Tokens in Windows Azure	34
Content Delivery Network	35
The Solution	36
Setting the Access Control for the BLOB Containers	36
Configuring the CDN and Storing the Content	37
Configuring URLs to Access the Content	37
Setting the Caching Policy	39
More Information	39

4	Building a Scalable, Multi-Tenant Application for Windows Azure	41
	Partitioning the Application	41
	The Solution	41
	Inside the Implementation	42
	On-Boarding for Trials and New Customers	45
	Basic Subscription Information	45
	Authentication and Authorization Information	46
	Provisioning a Trust Relationship with the Subscriber's Identity Provider	46
	Provisioning Authentication and Authorization for Basic Subscribers	47
	Provisioning Authentication and Authorization for Individual Subscribers	48
	Geo Location Information	48
	Database Information	48
	Billing Customers	48
	Customizing the User Interface	50
	Scaling Applications by Using Worker Roles	50
	Example Scenarios for Worker Roles	51
	Triggers for Background Tasks	52
	Execution Model	53
	The MapReduce Algorithm	54
	Scaling the Surveys Application	60
	Goals and Requirements	60
	The Solution	61
	Inside the Implementation	63
	Using a Worker Role to Calculate the Summary Statistics	63
	The Worker Role "Plumbing" Code	67
	Testing the Worker Role	72
	References and Resources	73
5	Working with Data in the Surveys Application	75
	A Data Model for a Multi-Tenant Application	75
	Storing Survey Definitions	76
	Storing Tenant Data	77
	Storing Survey Answers	78
	Storing Survey Answer Summaries	79

The Store Classes	80
SurveyStore Class	80
SurveyAnswerStore Class	80
SurveyAnswersSummaryStore Class	80
SurveySqlStore Class	81
SurveyTransferStore Class	81
TenantStore Class	81
Testing and Windows Azure Storage	81
Goals and Requirements	81
The Solution	81
Inside the Implementation	82
Saving Survey Response Data	87
Goals and Requirements	87
The Solution	87
Solution 1: The Delayed Write Pattern	88
Solution 2: Writing Directly to BLOB Storage	89
Comparing the Solutions	91
Inside the Implementation	92
Saving the Survey Response Data to a Temporary Blob	92
Displaying Data	94
Paging through Survey Results	94
Goals and Requirements	94
The Solution	94
Inside the Implementation	96
Session Data Storage	98
Goals and Requirements	98
The Solution	100
Inside the Implementation	102
Displaying Questions	103
Displaying the Summary Statistics	104
Using SQL Azure	105
Goals and Requirements	106
The Solution	106
Inside the Implementation	107
References and Resources	110

APPENDICES	
A	Updating a Windows Azure Service 111
	OPTIONS FOR UPDATING A WINDOWS
	AZURE SERVICE 111
	Limitations When Updating a Windows Azure Service 112
	Updating Only the Configuration of a Service 113
	Redeploying a Service without Incurring Downtime 113
	Design Considerations for Maximizing Upgradability 113
	About Upgrade Domains and Fault Domains 115
	Fault Domains 115
	Upgrade Domains 115
	Using Upgrade Domains for In-Place
	Upgrades 116
	Using the Windows Azure Service
	Management API 117
	Accessing the Windows Azure Service
	Management API 118
	Performing Service Upgrades Using the Service
	Management API 119
	Using the Windows Azure Service Management
	API to Manage Scaling 119
	Dependencies for Azure Applications and Projects 121
B	DEBUGGING AND TROUBLESHOOTING
	WINDOWS AZURE APPLICATIONS 121
	Debugging Azure Applications Locally
	during Development 122
	Common Setup And Deployment Errors
	with Windows Azure 123
	Using IntelliTrace with Applications
	Deployed to Windows Azure 124
	Using Windows Azure Diagnostics 126
	About Windows Azure Diagnostics 126
	Limitations of Windows Azure Diagnostics 129
	Using a Custom Tracing, and Debugging Agent 129
	Tools for Working with Windows Azure
	Diagnostics 130
	Configuring Diagnostics Using Configuration Files 132
	Using the Diagnostics API in a Windows Azure
	Application 133
	Resolving Permission Issues in Partial Trust Mode 140
	More Information 141
	INDEX 143

Foreword

Windows Azure is a highly scalable, robust, and cost-effective solution for the deployment of applications and services—for organizations and enterprises of all sizes, and in all types of markets. My task at Microsoft is to ensure that developers can help these organizations and enterprises to achieve their goals by quickly and easily taking advantage of the power, flexibility, and availability of Windows Azure services.

Developing on the Windows Azure platform is a natural extension for developers who are familiar with the Microsoft platform, and with tools and development environments such as .NET and Visual Studio. However, to be really successful in this exciting new arena requires developers to understand some less familiar concepts such as multi-tenancy, data partitioning, and the robust storage mechanisms used in Windows Azure and SQL Azure.

Within our teams, we've done our best to build a broad set of the tools that developers can use to build Windows Azure applications and services. And now, with this guide, the patterns & practices team is extending the developer experience by providing the information they require to build high performance, real world-ready applications on the Windows Azure platform.

“Developing Applications for the Cloud” is the second in a series of patterns & practices guides that explore the scenarios for building cloud-hosted applications. The first guide showed how developers can move their applications to the cloud. This guide explores the development of new applications for the cloud. It is based on the experiences of a fictitious organization named Tailspin that is building a scalable, multi-tenant application to host surveys; and explores major concepts and implementation details associated with the development of Windows Azure applications.

As with the other guides in this series, it takes a practical hands-on approach to explaining the scenario for the application, the design and development considerations, and how the application

takes advantage of features and capabilities of the Windows Azure platform. The guide focuses on the use of good practice design and code implementation, integrating with the run-time environment, and making full use of storage and background worker tasks. It includes debugging and troubleshooting information, and a comprehensive sample application that developers can download and use to help them understand the implementation, and to apply the knowledge they've acquired when designing and building their own applications.

I'm proud of the work we have done to provide developers with a great experience for working with Windows Azure, and I'm excited by the work that patterns & practices are doing with this series of guides. Together we can help developers to realize their potential, and to be productive and successful on the Windows Azure platform.

Bharat Shyam, General Manager
Windows Azure Developer Platform,
Microsoft Corporation

Preface

How can a company create an application that has truly global reach and that can scale rapidly to meet sudden, massive spikes in demand? Historically, companies had to invest in building an infrastructure capable of supporting such an application themselves and, typically, only large companies would have the available resources to risk such an enterprise. Building and managing this kind of infrastructure is not cheap, especially because you have to plan for peak demand, which often means that much of the capacity sits idle for much of the time. The cloud has changed the rules of the game: by making the infrastructure available on a “pay as you go” basis, creating a massively scalable, global application is within the reach of both large and small companies.

The cloud platform provides you with access to capacity on demand, fault tolerance, distributed computing, data centers located around the globe, and the capability to integrate with other platforms. Someone else is responsible for managing and maintaining the entire infrastructure, and you only pay for the resources that you use in each billing period. You can focus on using your core domain expertise to build and then deploy your application to the data center or data centers closest to the people who use it. You can then monitor your applications, and scale up or scale back as and when the capacity is required.

Yes, by moving applications to the cloud, you’re giving up some control and autonomy, but you’re also going to benefit from reduced costs, increased flexibility, and scalable computation and storage. *Developing Applications for the Cloud on the Microsoft Windows Azure Platform* shows you how to do this.

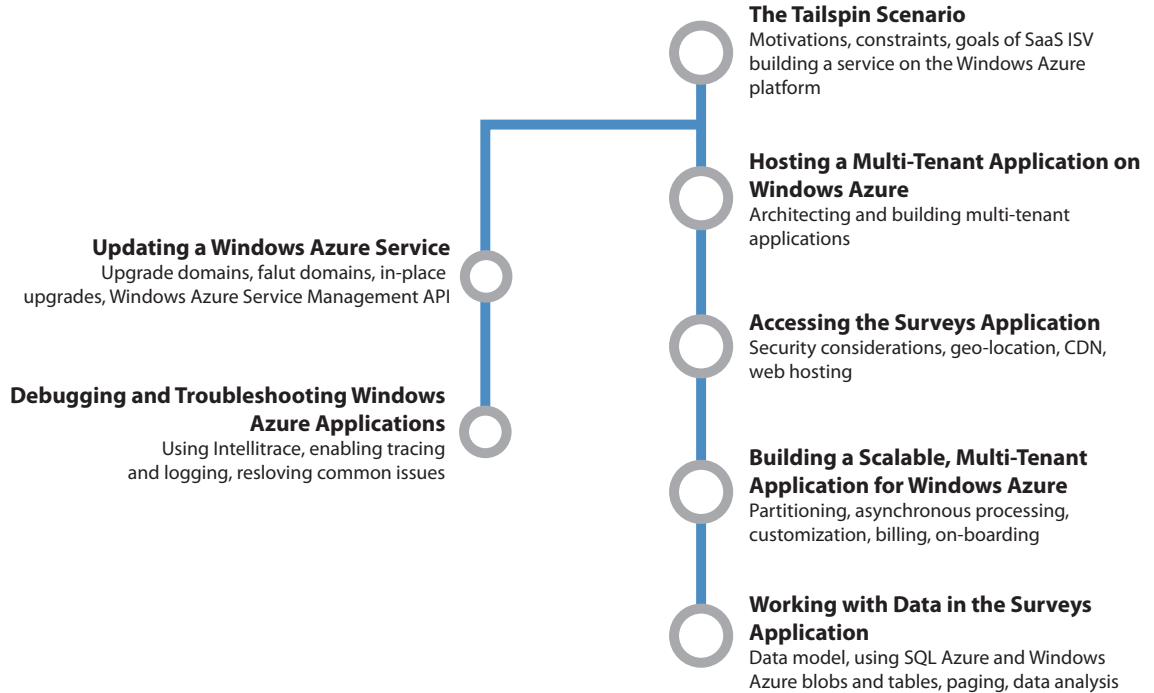
Who This Book Is For

This book is the second volume in a planned series about Windows Azure™ technology platform. Volume 1, *Moving Applications to the Cloud on the Windows Azure Platform*, provides an introduction to Windows Azure, discusses the cost model and application life cycle management for cloud-based applications, and describes how to migrate an existing ASP.NET application to the cloud. This book demonstrates how you can create from scratch a multi-tenant, Software as a Service (SaaS) application to run in the cloud by using the latest versions of the Windows Azure tools and the latest features of the Windows Azure platform. The book is intended for any architect, developer, or information technology (IT) professional who designs, builds, or operates applications and services that run on or interact with the cloud. Although applications do not need to be based on the Microsoft® Windows® operating system to work in Windows Azure, this book is written for people who work with Windows-based systems. You should be familiar with the Microsoft .NET Framework, Microsoft Visual Studio® development system, ASP.NET MVC, and Microsoft Visual C#® development tool.

Why This Book Is Pertinent Now

In general, the cloud has become a viable option for making your applications accessible to a broad set of customers. In particular, Windows Azure now has in place a complete set of tools for developers and IT professionals. Developers can use the tools they already know, such as Visual Studio, to write their applications for the cloud. In addition, Windows Azure provides a complete, simulated environment known as the *development fabric* that developers can use to locally write, test, and debug their applications before they deploy them to the cloud. There are also tools and an API to manage your Windows Azure accounts. This book shows you how to use all these tools in the context of a common scenario—how to develop a brand new, multi-tenant, SaaS application for Windows Azure.

How This Book Is Structured



“The Tailspin Scenario” introduces you to the Tailspin company and the Surveys application. It provides an architectural overview of the Surveys application; the following chapters provide more information about how Tailspin designed and implemented the Surveys application for the cloud. Reading this chapter will help you understand Tailspin’s business model, its strategy for adopting the cloud platform, and some of its concerns.

“Hosting a Multi-Tenant Application on Windows Azure” discusses some of the issues that surround architecting and building multi-tenant applications to run on Windows Azure. It describes the benefits of a multi-tenant architecture and the trade-offs that you must consider. This chapter provides a conceptual framework that helps the reader understand some of the topics discussed in more detail in the subsequent chapters.

“Accessing the Surveys Application” describes some of the challenges that the developers at Tailspin faced when they designed and implemented some of the customer-facing components of the application. Topics include the choice of URLs for accessing the surveys application, security, hosting the application in multiple geographic locations, and using the Content Delivery Network to cache content.

“Building a Scalable, Multi-Tenant Application for Windows Azure” examines how Tailspin ensured the scalability of the multi-tenant Surveys application. It describes how the application is partitioned, how the application uses worker roles, and how the application supports on-boarding, customization, and billing for customers.

“Working with Data in the Surveys Application” describes how the application uses data. It begins by describing how the Surveys application stores data in both Windows Azure tables and blobs, and how the developers at Tailspin designed their storage classes to be testable. The chapter also describes how Tailspin solved some specific problems related to data, including paging through data, and implementing session state. Finally, this chapter describes the role that SQL Azure™ technology platform plays in the Surveys application.

“Updating a Windows Azure Service” describes the options for updating a Windows Azure application and how you can update an application with no interruption in service.

“Debugging and Troubleshooting Windows Azure Applications” describes some of the techniques specific to Windows Azure applications that will help you to detect and resolve issues when building, deploying, and running Windows Azure applications. It includes descriptions of how to use Windows Azure Diagnostics and how to use Microsoft IntelliTrace™ with applications deployed to Windows Azure.

What You Need to Use the Code

These are the system requirements for running the scenarios:

- Microsoft Windows Vista SP1, Windows 7, or Microsoft Windows Server® 2008 (32-bit or 64-bit)
- Microsoft Internet Information Services (IIS) 7.0
- Microsoft .NET Framework 4 or later
- Microsoft Visual Studio 2010
- Windows Azure Tools for Microsoft Visual Studio 2010
- ASP.NET MVC 2.0
- Windows Identity Foundation
- Microsoft Anti-Cross Site Scripting Library
- Moq (to run the unit tests)
- Enterprise Library Unity Application Block (binaries included in the samples)

Who's Who

As mentioned earlier, this book uses a sample application that illustrates how to implement applications for the cloud. A panel of experts comments on the development efforts. The panel includes a cloud specialist, a software architect, a software developer, and an IT professional. The delivery of the sample application can be considered from each of these points of view. The following table lists these experts.



Bharath is a cloud specialist. He checks that a cloud-based solution will work for a company and provide tangible benefits. He is a cautious person, for good reasons.

Implementing a single-tenant application for the cloud is easy. Realizing the benefits that a cloud-based solution can offer to a multi-tenant applications is not always so straight-forward.

Jana is a software architect. She plans the overall structure of an application. Her perspective is both practical and strategic. In other words, she considers not only what technical approaches are needed today, but also what direction a company needs to consider for the future.



It's not easy to balance the needs of the company, the users, the IT organization, the developers, and the technical platforms we rely on.



Markus is a senior software developer. He is analytical, detail-oriented, and methodical. He's focused on the task at hand, which is building a great cloud-based application. He knows that he's the person who's ultimately responsible for the code.

For the most part, a lot of what we know about software development can be applied to the cloud. But, there are always special considerations that are very important.

Poe is an IT professional who's an expert in deploying and running in a corporate data center. Poe has a keen interest in practical solutions; after all, he's the one who gets paged at 3:00 AM when there's a problem.



Running applications in the cloud that are accessed by thousands of users involves some big challenges. I want to make sure our cloud apps perform well, are reliable, and are secure. The reputation of Tailspin depends on how users perceive the applications running in the cloud.

If you have a particular area of interest, look for notes provided by the specialists whose interests align with yours.

1

The Tailspin Scenario

This chapter introduces a fictitious company named Tailspin. It describes Tailspin's plans to launch a new, online service named Surveys that will enable other companies or individuals to conduct their own online surveys. The chapter also describes why Tailspin wants to host its survey application on the Windows Azure™ technology platform. As with any company considering this process, there are many issues to consider and challenges to be met, particularly because this is the first time Tailspin is using the cloud. The chapters that follow this one show, step-by-step, how Tailspin architected and built its survey application to run on Windows Azure.

The Tailspin Company

Tailspin is a startup ISV company of approximately 20 employees that specializes in developing solutions using Microsoft® technologies. The developers at Tailspin are knowledgeable about various Microsoft products and technologies, including the .NET Framework, ASP.NET MVC, SQL Server®, and Microsoft Visual Studio® development system. These developers are aware of Windows Azure but have not yet developed any complete applications for the platform.

The Surveys application is the first of several innovative online services that Tailspin wants to take to market. As a startup, Tailspin wants to develop and launch these services with a minimal investment in hardware and IT personnel. Tailspin hopes that some of these services will grow rapidly, and the company wants to have the ability to respond quickly to increasing demand. Similarly, it fully expects some of these services to fail, and it does not want to be left with redundant hardware on its hands.

TAILSPIN'S STRATEGY

Tailspin is an innovative and agile organization, well placed to exploit new technologies and the business opportunities offered by the

cloud. As a startup, Tailspin is willing to take risks and use new technologies when it implements applications. Tailspin's plan is to embrace the cloud and gain a competitive advantage as an early adopter. It hopes to rapidly gain some experience, and then quickly expand on what it has learned. This strategy can be described as "try, fail fast, learn, and then try again." Tailspin has decided to start with the Surveys application as its first cloud-based service offering.

THE SURVEYS APPLICATION

The Surveys application enables Tailspin's customers to design a survey, publish the survey, and collect the results of the survey for analysis. A survey is a collection of questions, each of which can be one of several types such as multiple-choice, numeric range, or free text. Customers begin by creating a subscription with the Surveys service, which they use to manage their surveys and to apply branding by using styles and logo images. Customers can also select a geographic region for their account, so that they can host their surveys as close as possible to the survey audience. The Surveys application allows users to try out the application for free, and to sign up for one of several different packages that offer different collections of services for a monthly fee.

Figure 1 illustrates the Surveys application and highlights the three different groups of users who interact with application.

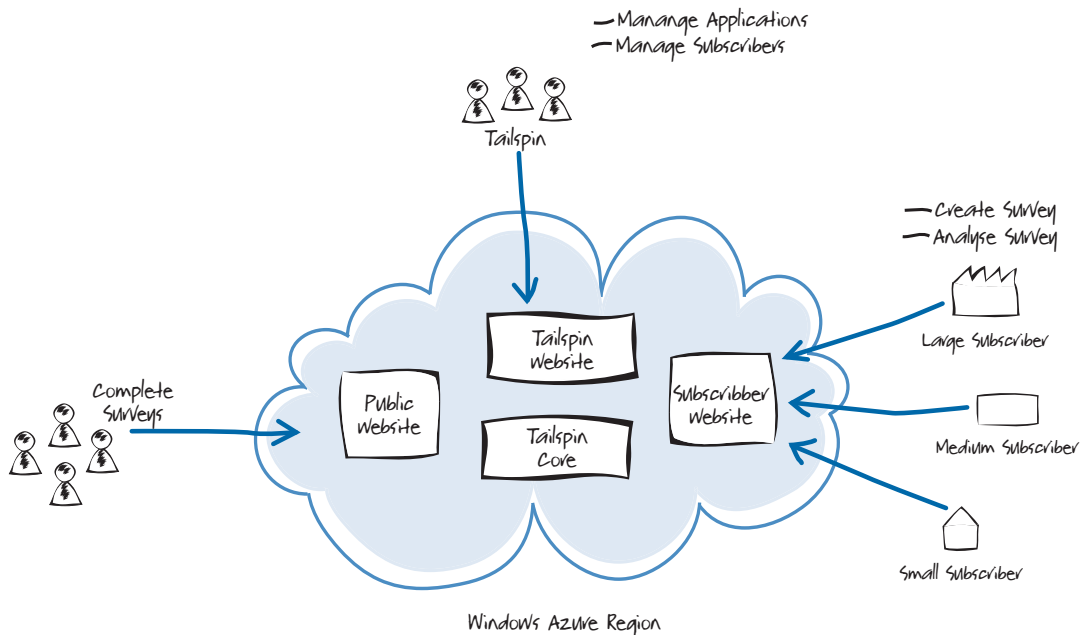


FIGURE 1
The Surveys application

Customers who have subscribed to the Surveys service (or who are using a free trial) access the Subscribers website that enables them to design their own surveys, apply branding and customization, and collect and analyze the survey results. Depending on the package they select, they have access to different levels of functionality within the Surveys application. Tailspin expects its customers to be of various sizes and from all over the world, and customers can select a geographic region for their account and surveys.

Tailspin wants to design the service in such a way that most of the administrative and configuration tasks are “self-service” and are performed by the subscriber with minimal intervention by Tailspin staff.

The Public website enables the people participating in the survey to complete their responses to the survey questions. The survey creator will inform their survey audience of the URL to visit to complete the survey.

The Tailspin website enables staff at Tailspin to manage the application and manage the subscriber accounts. All three websites interact with the core services that comprise the Surveys application and provide access to the application’s data storage.

TAILSPIN’S GOALS AND CONCERNS

Tailspin faces several challenges, both as an organization and with the Surveys application in particular. First, customers might want to create surveys associated with a product launch, a marketing campaign, or the surveys might be seasonal, perhaps associated with a holiday period. Often, customers who use the survey application will want to set up these surveys with a very short lead-time. Surveys will usually run for a fixed, short period of time but may have a large number of respondents. This means that usage of the Surveys application will tend to spike, and Tailspin will have very little warning of when these spikes will occur. Tailspin wants to be able to offer the Surveys application to customers around the world, and because of the nature of the Surveys application with sudden spikes in demand, it wants to be able to quickly expand or contract its infrastructure in different geographical locations. It doesn’t want to purchase and manage its own hardware or to maintain sufficient capacity to meet peak demand. It also doesn’t want to sign long-term contracts with hosting providers for capacity that it will only use for part of the time.

Tailspin wants to be able to maintain its competitive advantage by rapidly rolling out new features to existing services or to gain competitive advantage by being first to market with new products and services.

With the Surveys application, Tailspin wants to offer its customers a reliable, customizable, and flexible service for creating and conducting online surveys. It must give its customers the ability to create



In the world of Software as a Service (SaaS), subscribers are commonly known as “Tenants.” We commonly refer to applications like Tailspin Surveys as “multi-tenant” applications.



Resource elasticity and geo-distribution are key properties of the Windows Azure platform

surveys using a range of question types, and the ability to brand the surveys using corporate logos and color schemes.

Tailspin wants to be able to offer different packages (at different prices) to customers, based on a customer's specific requirements. Tailspin wants to offer its larger customers the ability to integrate the Surveys application into the customer's own infrastructure. For example, integration with the customer's own identity infrastructure could provide single sign-on (SSO) or enable multiple users to manage surveys or access billing information. Integration with the customer's own Business Intelligence (BI) systems could provide for a more sophisticated analysis of survey results. For small customers who don't need, or can't use, the sophisticated integration features, a basic package might include an authentication system. The range of available packages should also include a free trial to enable customers to try the Surveys application before they purchase it.

The subscriber and public websites also have different scalability requirements. It is likely that thousands of users might complete a survey, but only a handful of users from each subscriber will edit existing surveys or create new surveys. Tailspin wants to optimize the resources for each of these scenarios.

The Tailspin business model is to charge subscribers a monthly fee for a service such as the Surveys application and, because of the global market they are operating in, Tailspin wants its prices to be competitive. Tailspin must then pay the actual costs of running the application, so in order to maintain their profit margin Tailspin must tightly control the running costs of the services they offer to their customers.

Note: *In this scenario, Tailspin's customers (the subscribers) are not Windows Azure customers. Subscribers pay Tailspin, who in turn pays Microsoft for their use of Windows Azure platform components.*

Tailspin wants to ensure that customer's data is kept safe. For example, a customer's data must be private to that customer, there must be multiple physical copies of the survey data, and customers should not be able to lose data by accidentally deleting a survey. In addition, all existing survey data must be preserved whenever Tailspin updates the application.

Finally, Tailspin would like to be able to leverage the existing skills of its developers and minimize any retraining necessary to build the Surveys application.

The Surveys Application Architecture

To achieve the goals of the Surveys application, Tailspin decided to implement the application as a cloud-based service using the Windows Azure platform. Figure 2 shows a high-level view of this architecture.

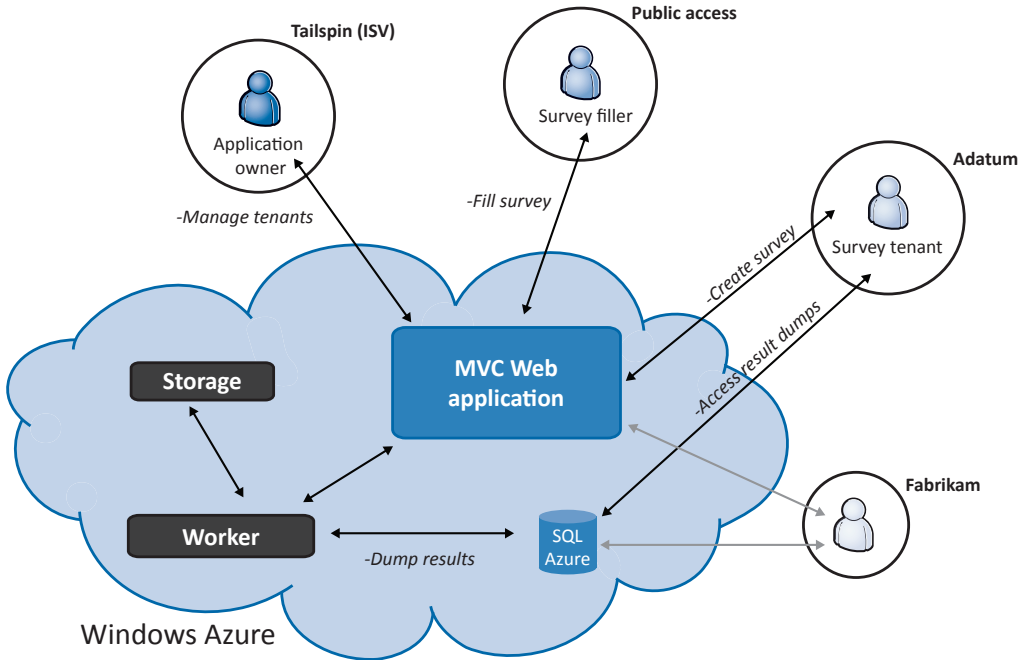


FIGURE 2
The Surveys application architecture

The architecture of the Surveys Application is straightforward and one that many other Windows Azure applications use. The core of the application uses Windows Azure web roles, worker roles, and storage. Figure 2 shows the three groups of users who access the application: the application owner, the public, and subscribers to the Surveys service (in this example, Adatum and Fabrikam). It also highlights how the application uses SQL Azure™ technology platform to provide a mechanism for subscribers to dump their survey results into a relational database to analyze the results in detail. This guide discusses the design and implementation in detail and describes the various web and worker roles that comprise the Surveys application.

Some of the specific issues that the guide covers include how Tailspin implemented the Surveys application as a multi-tenant application in Windows Azure and how the developers designed the application to be testable. The guide describes how Tailspin handles



Tailspin can use the Content Delivery Network feature of Windows Azure to provide caching services.

the integration of the application's authentication mechanism with a customer's own security infrastructure by using a federated identity with multiple partners model. The guide also covers the reasoning behind the decision to use a hybrid data model that uses both Windows Azure Storage and SQL Azure. Other topics covered include how the application uses caching to ensure the responsiveness of the Public website for survey respondents, how the application automates the on-boarding and provisioning process, how the application leverages the Windows Azure geographic location feature, and the customer-billing model adopted by Tailspin for the Surveys application.

Tailspin will build the application using the latest available technologies: Visual Studio 2010, ASP.NET MVC 2.0, and .NET Framework 4.

2

Hosting a Multi-Tenant Application on Windows Azure

This chapter discusses some of the issues that surround architecting and building multi-tenant applications to run on Windows Azure™ technology platform. A highly scalable, cloud-based platform offers a compelling set of features for building services that many users will pay a subscription to use. A multi-tenant architecture where multiple users share the application enables economies of scale as users share resources, but at the cost of a more complex application that has to manage multiple users independently of each other.

This chapter does not focus specifically on Tailspin or the Surveys application, but it uses the scenario described in the previous chapter to illustrate some of the factors that you might consider when choosing whether to implement a multi-tenant application on Windows Azure.

This chapter provides a conceptual framework that helps you understand some of the topics discussed in more detail in the subsequent chapters of this guide.

Single-Tenant vs. Multi-Tenant

One of the first architectural decisions that the team at Tailspin had to make about the Surveys application was whether it should be a single-tenant or multi-tenant application to best support multiple customers. Figure 1 shows the difference between these approaches at a high-level. The single-tenant model has a separate, logical instance of the application for each customer, while the multi-tenant model has a single logical instance of the application shared by many customers. It's important to note that the multi-tenant model still offers separate views of the application's data to its users. In the Surveys application, ClientB must not be able to see or modify ClientA's surveys or data. Tailspin, as the owner of the application, will have full access to all the data stored in the application.

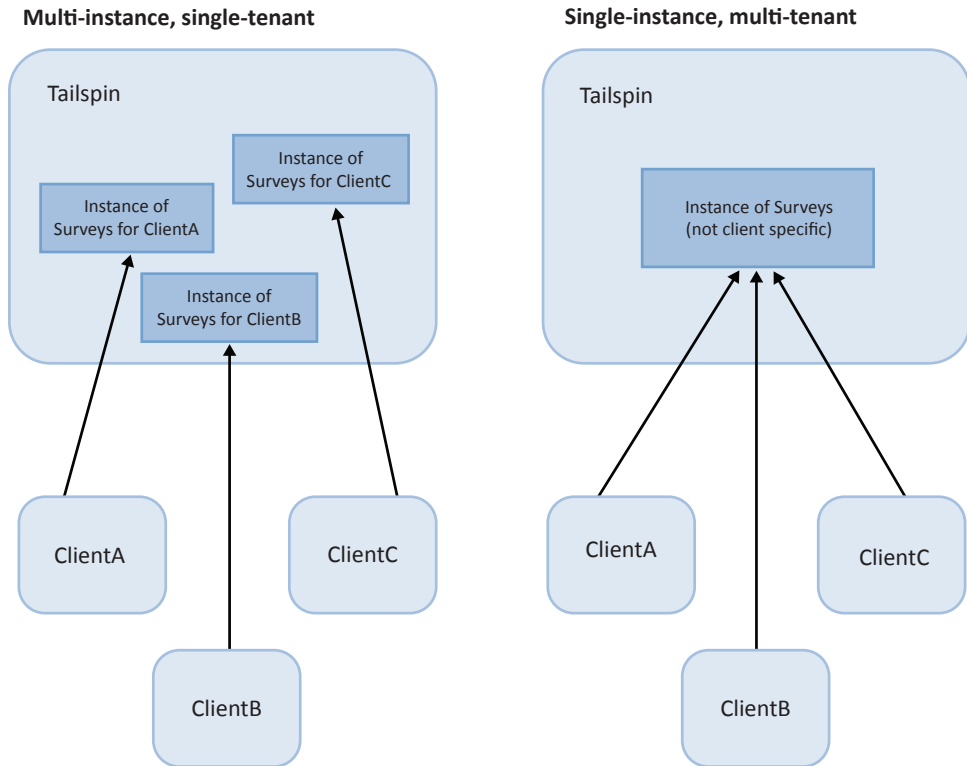
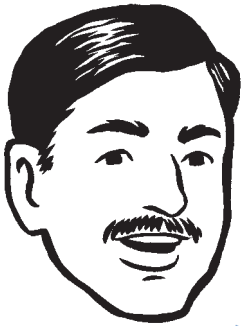


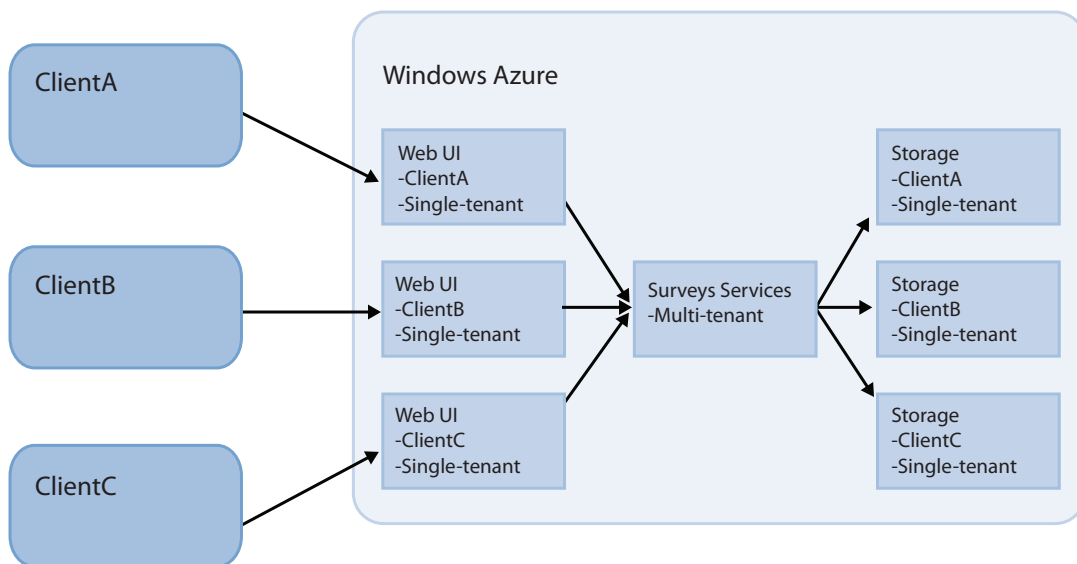
FIGURE 1
Logical view of single-tenant and multi-tenant architectures



This diagram shows logical instances of the Surveys application. In practice, you can implement each logical instance as multiple physical instances to scale the application

Multi-Tenancy Architecture in Azure

In Windows Azure, the distinction between the multi-tenant model and the single-tenant model is not as straightforward as the model in Figure 1 because an application in Windows Azure can be made up of multiple components, each of which can be single-tenanted or multi-tenanted. For example, if an application has a user interface (UI) component, a services component, and a storage component, a possible design could look like that shown in Figure 2.

**FIGURE 2****Sample architecture for Windows Azure**

This is not the only possible design, but it illustrates that you don't have to make the same choice of either single-tenancy or multi-tenancy model for every component in your application.

Should you design your Windows Azure application to be single-tenant or multi-tenant? There's no right or wrong answer, but as you will see in the following section, there are a number of factors that can influence your choice.

Selecting a Single-Tenant or Multi-Tenant Architecture

This section introduces some of the criteria that an architect would consider when deciding on a single-tenant or multi-tenant design. This book revisits many of these topics in more detail, and with specific reference to Tailspin and the Surveys application, in later chapters. The relative importance of the different criteria will vary for different application scenarios.

ARCHITECTURAL CONSIDERATIONS

The architectural requirements of your application will influence your choice of a single-tenant or multi-tenant architecture.



You can always have one tenant in a multi-tenant application, but you can't have multiple tenants in a single-tenant application.

Application Stability

A multi-tenant application is more vulnerable to instance failure than a single-tenant application. If a single-tenant instance fails, only the customer using that instance is affected, whereas if the multi-tenant instance fails, all customers are affected. However, Windows Azure can mitigate this risk by enabling you to deploy multiple, identical copies of your application into multiple Windows Azure role instances (this is really a multi-tenant, multi-instance model). Windows Azure load balances requests across those role instances, and you must design your application to ensure that it functions correctly when you deploy multiple instances. For example, if your application uses Session state, you must make sure that each web role instance can access the state. Windows Azure will monitor your role instances and automatically restart any failed role instances.

Making the Application Scalable

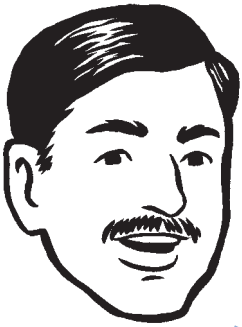
The scalability of an application running on Windows Azure depends largely on being able to deploy multiple instances of your web and worker roles to multiple compute nodes while being able to access the same data from those nodes. Both single-tenant and multi-tenant applications use this feature to scale out when they run on Windows Azure. Windows Azure also offers various sizes of compute nodes that enable you to scale up or scale down individual instances.

For some applications, you may not want to have all your customers sharing a single, multi-tenant instance. For example, you may want to group your customers based on the functionality they use or their expected usage patterns, and then optimize each instance for the customers who are using it. In this case, you may need to have two or more copies of your multi-tenant application deployed in different Windows Azure accounts.

Service Level Agreements

You may want to offer a different Service Level Agreement (SLA) with the different subscription levels for the service. If subscribers with different SLAs are sharing the same multi-tenant instance, you should aim to meet the highest SLA, thereby ensuring that you also satisfy the lower SLAs for other customers.

However, if you have a limited number of different SLAs, you could put all the customers that share the same SLA into the same multi-tenant instance and make sure that the instance has sufficient resources to satisfy the requirements of the SLA.



In Windows Azure, the preferred way to scale your application is to scale out by adding additional nodes instead of scaling up by using larger nodes. This enables you to add or remove capacity as and when it's needed.

The Legal and Regulatory Environment

For certain applications, you may need to take into account specific regulatory or legal issues. This may require some differences in functionality, specific legal messages to be displayed in the UI, guaranteed separate databases for storage, or storage located in a specific region. This may again lead to having separate multi-tenant deployments for groups of customers, or it may lead to requiring a single-tenant architecture.

Handling Authentication and Authorization

You may want to provide your own authentication and authorization systems for your cloud application that require customers to set up accounts for the users who will interact with the application. However, customers may prefer to use an existing authentication system and avoid having to create a new set of credentials for your application. In a multi-tenant application, this would imply being able to support multiple authentication providers, and it may possibly require a custom mapping to your application's authorization scheme. For example, someone who is a "Manager" in Microsoft® Active Directory® directory service at Adatum might map to being an "Administrator" in Adatum's Tailspin Surveys application.

Note: For more information about claims-based identity, see the book, *A Guide to Claims-Based Identity and Access Control*. You can download a PDF copy of this book from <http://msdn.microsoft.com/en-us/library/ff423674.aspx>.

APPLICATION LIFE CYCLE MANAGEMENT CONSIDERATIONS

Your choice of a single-tenant or multi-tenant architecture will affect how easy it is to develop, deploy, maintain, and monitor your application.

Maintaining the Code Base

Maintaining separate code bases for different customers will rapidly lead to escalating support and maintenance costs for an ISV because it becomes more difficult to track which customers are using which version. This will lead to costly mistakes being made. A multi-tenant system with a single, logical instance guarantees a single code base for the application. You can still maintain a single code base with a multi-instance, single-tenant model, but there could be a short-term temptation (with long-term consequences) to branch the code for

individual customers in order to meet specific customer requirements. In some scenarios, where there is a requirement for a high-degree of customization, multiple code bases may be a viable option, but you should explore how far you can get with custom configurations or custom business rule components before going down this route. If you do need multiple code bases, you should structure your application such that custom code is limited to as few components as possible.

Handling Application Upgrades

A multi-tenant application makes it easy to roll out application updates to all your customers at the same time. This approach means that you only have a single, logical instance to upgrade, which reduces the maintenance effort. In addition, you know that all your customers are using the latest version of the software, which makes the support job easier. Windows Azure upgrade domains facilitate this process by enabling you to roll out your upgrade across multiple role instances without stopping the application. If a client has operational procedures or software that are tied to a specific version of your application, any upgrades must be coordinated with that client.

To mitigate the risks associated with upgrading the application, you can implement a rolling upgrade program that upgrades some users, monitors the new version, and when you are confident in the new version, rolls out the changes to the remainder of the user base.

Note: *For more information about how to handle application upgrades, see Appendix A, “Updating a Windows Azure Service.”*

Monitoring the Application

Monitoring a single application instance is easier than monitoring multiple instances. In the multi-instance, single tenant model, any automated provisioning would need to include setting up the monitoring environment for the new instance, which will add to the complexity of the provisioning process for your application. Monitoring will also be more complex if you decide to use rolling upgrades because you must monitor two versions of the application simultaneously and use the monitoring data to evaluate the new version of the application.

Using .NET Providers and Third-Party Components

If you decide on a multi-tenant architecture, you must carefully evaluate how well any third-party components will work. You may need to take some additional steps ensure that a third-party component is “multi-tenant aware.” With a single-tenant, multi-instance deployment where you want to be able to scale out for large tenants, you

will also need to verify that third-party components are “multi-instance aware.”

Provisioning for Trials and New Customers

Provisioning a new client or initializing a free trial of your service will be easier and quicker to manage if it involves only a configuration change. A multi-instance, single-tenant model will require you to deploy a new instance of the application for every customer, including those using a free trial. Although you can automate this process, it will be considerably more complicated than changing or creating configuration data in a single-instance, multi-tenant application.

CUSTOMIZING THE APPLICATION

Whether you choose a single-tenant or multi-tenant architecture, customers will still need to be able to customize the application.

URLs to Access the Application

By default, Windows Azure gives each application a Domain Name System (DNS) name like this: *<yourappname>.cloudapp.net*. You can use a DNS CNAME record to map a custom DNS name to your application. For example, if the Tailspin Surveys application is named *tailspin.cloudapp.net*, Tailspin could use a CNAME entry to map the URL *https://surveys.tailspin.com* to the application. If each customer has its own, separate, single-tenant instance of the application running in a separate Windows Azure account, you could map a custom DNS name to each instance of the application. For example, you could map *https://surveys.adatum.com* and *https://surveys.fabrikam.com* to separate instances.

Because Internet Information Services (IIS) can have only one SSL certificate associated with a port, a multi-tenant application can use only a single domain name on the default port 443. Therefore, in a multi-tenant application, you can use an addressing scheme like this: *https://<azureaccount>.cloudapp.net/<app>/<tenant>*. Adatum, a subscriber to the Tailspin Surveys application, would access the Surveys application at *https://services.tailspin.com/surveys/adatum*.

Customizing the Application by Tenant

Customers will want to be able to style and brand the site for their own users. You must establish how much control customers will want in order to determine how best to enable the customization. At one end of the scale, you might provide the customer with the ability to customize the appearance of the application by allowing them to upload cascading style sheets and image files. At the other end of the scale, you might enable the customer to design complete pages that interact with the application’s services through a standard API.



If you don't need to use SSL, you can use custom domain names for each tenant in a multi-tenant application. Each tenant can create a CNAME record to map his or her domain name to the Windows Azure application.



This is, of course, nothing new. Microsoft Dynamics® CRM is a great example of an application that has these levels of customization available.

For some applications, you may want to provide customers with the ability to enable or disable certain functionality. For example, in the Surveys application, customers can choose whether to integrate the application's identity infrastructure with their own infrastructure, and they can choose the geographic location for their surveys. This type of configuration data can easily be stored in Windows Azure table storage.

Other applications may require the ability to enable users to customize the business process within the application to some degree. Options here would include implementing a plug-in architecture so that customers could upload their own code or using some form of rules engine that enables process customization through configuration.

You may also want to provide customers with ways to extend the application without using custom code. Users of the survey application may want to capture additional information about a survey respondent that the standard application does not collect. This means that users must have a mechanism for customizing the UI to collect the data and a way of extending the data storage schema to include the new data.

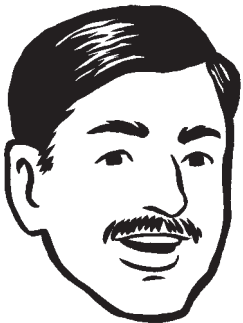
MULTI-TENANT DATA ARCHITECTURE

Your architecture must ensure that a customer's data is kept private from other customers. Your application may also need to support customized data storage.

Note: For more information about multi-tenant data architectures, see "Multi-Tenant Data Architecture" (<http://msdn.microsoft.com/en-us/library/aa479086.aspx>) and "Architecture Strategies for Catching the Long Tail" (<http://msdn.microsoft.com/en-us/library/aa479069.aspx>) on MSDN®.

Protecting Data from Other Tenants

The perceived risk of either accidental or malicious data disclosure is greater in a multi-tenant model. It will be harder to convince customers that their private data is safe if they know they are physically sharing the application with other customers. However, a robust design that logically isolates each tenant's data can provide a suitable level of protection. This type of design might use database schemas where each tenant's tables are in a separate schema, database security features that enable you to use access control mechanisms within the database, a partitioning scheme to separate tenant's data, or a combination of these approaches.



Allowing tenants to upload their own code increases the risk of application failure, because you have less control over the code that is running in the application. Many Software as a Service (SaaS) systems apply limits to this. Most simply disallow it. Allowing tenants to upload code or scripts also increases the security risks associated with the application.

The security considerations just described apply to both Windows Azure storage and SQL Azure™ technology platform. However, they each have a different billing model. Usage of Windows Azure storage is billed by the amount of storage used and by the number of storage transactions, so from a cost perspective it doesn't matter how many separate storage accounts or containers you have. SQL Azure is billed based on the number of databases you have, so it makes sense from a cost perspective to have as many tenants as possible sharing each instance.

Data Architecture Extensibility

There are a number of ways that you can design your data storage to enable tenants to extend the data model to include their own custom data. These approaches range from each tenant having a separate schema, to providing a set of pre-defined custom columns, to more flexible schemas that enable a tenant to add an arbitrary number of custom fields to a table.

If you use SQL Azure, much of the application's complexity will result from having to work within the constraints of fixed data schemas. If you are using Windows Azure table storage, the complexity will arise from working with variable schemas. Windows Azure table storage allows records in the same table to have completely different structures, which allows for a great deal of flexibility at the cost of more complexity in the code.

Custom extensions to the application's data model should not require changes to application code. To enable the application's business logic and presentation logic to integrate with the data model extensions, you will require either a set of configuration files that describe the extensions or code that can dynamically discover the extensions.

Data Architecture Scalability

If you can partition your data horizontally, you will be able to scale out your data storage. In the case of SQL Azure, if you decide that you need to scale out, you should be able to move all of an individual tenant's data to a new SQL Azure instance.

Note: *Partitioning data horizontally, also known as sharding, implies taking some of the records in a table and moving them to a new table. Partitioning data vertically implies taking some fields from every row and placing them in a different table. For a discussion of partitioning strategies in SQL Azure, see the paper, [SQL Azure Considerations Guide](http://wag.codeplex.com); you can download it at <http://wag.codeplex.com>.*



You are billed for SQL Azure based on the number of databases you have, and the size of the databases. If you transfer data in and out of SQL Azure from within the same data center, there's no data transfer cost, but if you transfer data in and out of SQL Azure from outside the data center, you'll be charged for the data transfer.



Microsoft SharePoint® is an example of an application with a fixed schema database that is made to look extremely flexible.

The key factor that determines the scalability of Windows Azure table storage is the choice of partition key. All queries should include the partition key to avoid scanning multiple partitions. For more information, see “Phase 2: Automating Deployment and Using Windows Azure Storage” of *Windows Azure Architecture Guide, Part 1, Moving Applications to the Cloud*; it is available at <http://msdn.microsoft.com/en-us/library/ff728592.aspx>.

FINANCIAL CONSIDERATIONS

Your billing and cost model may affect your choice of single-tenant or multi-tenant architecture.

Billing Customers

For an application deployed to Windows Azure, Microsoft will bill you each month for the services (compute, storage, transactions, and so on) that each of your Windows Azure accounts consumes. If you are selling a service to your customers, like the Tailspin Surveys application, you need to bill your customers for the service.

One approach to billing is to use a pay-per-use plan. With this approach, you need to monitor the resources used by each of your customers, calculate the cost of those resources, and apply a markup to ensure you make a profit. If you use a single-tenant architecture and create a separate Windows Azure account for each of your customers, it's easy to determine how much an individual customer is costing in terms of compute time, storage, and so on, and then to bill the customer appropriately. However, for a single-tenant instance running in a separate Windows Azure account, some costs will effectively be fixed; for example, paying for a 24x7 compute instance, or a SQL Azure instance, may make the starting cost too high for small customers. With a multi-tenant architecture, you can share the fixed costs between tenants, but calculating the costs per tenant is not so straightforward and you will have to add some additional code to your application to meter each tenant's application usage. Furthermore, customers will want some way of tracking their costs, so you will need to be transparent about how the costs are calculated and provide access to the captured usage data.

It is difficult to predict exactly what usage an individual subscriber will make of the service; for the Surveys application, Tailspin cannot predict how many surveys a subscriber will create or how many survey answers the subscriber will receive in a specified period. If Tailspin adopts a billing model that offers the Surveys service for a fixed monthly fee, the profit margin will vary between subscribers (and could even be negative in some cases). By making Surveys a multi-tenant application, Tailspin can smooth out the differences in usage patterns between subscribers, making it much easier to predict costs

and revenue, and reduce the risk of making a loss. The more customers you have, the easier it becomes to predict aggregate usage patterns for a service.

From the customer's perspective, charging a fixed fee for the service means that the customer knows in advance exactly what their costs will be for the next billing period. This also means that you have a much simpler billing system. Some costs, like those associated with storage and transactions, will be variable and depend on the number of customers you have and how they use the service. Other costs, such as compute costs or the cost of a SQL Azure instance, will effectively be fixed. To be profitable, you need to sell sufficient subscriptions to cover both the fixed and variable costs.

If your customer base is a mixture of heavy users and light users, a standard monthly charge may be too high to attract smaller users. In this scenario, you'll need a variation on the second approach and offer a range of packages for different usage levels. For example, in the Surveys application, Tailspin might offer a light package at a lower monthly cost than the standard package. The light package may limit the number of surveys a customer can create or the number of survey responses that a customer can collect each month.

Offering a product where different customers can choose different features and/or quotas requires that you architect and design the product with that in mind. Such a requirement affects the product at all levels: presentation, logic, and data. You'll also need to undertake some market research to determine the expected demand for the different packages at different costs to try to estimate your expected revenue stream and costs.

Managing Application Costs

You can divide the running costs of a Windows Azure application into fixed and variable costs. For example, if the cost of a compute node is \$0.12/hour, the cost of running two compute nodes (to gain redundancy) 24x7 for one month is a fixed cost of approximately \$180. If this is a multi-tenant application, all the tenants share that cost. To reduce the cost per tenant, you should try to have as many tenants as possible sharing the application, without causing a negative impact on the performance of the application. You also need to analyze the application's performance characteristics to determine whether scaling up by using larger compute nodes or scaling out by adding additional instances would be the best approach for your application when demand increases.

Variable costs will depend on how many customers you have or how those customers use the application. In the Tailspin Surveys application, the number of surveys and the number of respondents for each survey will largely determine monthly storage and transaction

costs. Whether your application is single-tenant or multi-tenant will not affect the cost per tenant; regardless of the model, a specific tenant will require the same amount of storage and use the same number of compute cycles. To manage these costs, you must make sure that your application uses these resources as efficiently as possible.

Note: For more information about estimating Windows Azure costs, see Chapter 4, “How Much Will It Cost?”, of the book, Windows Azure Architecture Guide, Part 1, Moving Applications to the Cloud; it is available at <http://msdn.microsoft.com/en-us/library/ff728592.aspx>.

You can find additional information about storage costs in this post on the Windows Azure Storage Team blog: <http://blogs.msdn.com/b/windowsazurestorage/archive/2010/07/09/understanding-windows-azure-storage-billing-bandwidth-transactions-and-capacity.aspx>.

3

Accessing the Surveys Application

This chapter discusses some of the challenges faced by the developers at Tailspin when they were designing the customer-facing components of the Surveys application. The chapter focuses on the ways in which customers will interact with the application and begins by describing Tailspin's choice of URLs for accessing the application and its use of Secure Sockets Layer (SSL).

Tailspin plans to offer subscriptions to the Surveys application to a range of users, from large enterprises to individuals. These subscribers could be based anywhere in the world and may want to run surveys in other geographic locations. This chapter describes how Tailspin designed the Surveys application to be "geo-aware." The chapter also looks at how subscribers are authenticated and authorized, and how the application can use the Windows Azure Content Delivery Network (CDN) to improve the user experience.

DNS Names, Certificates, and SSL in the Surveys Application

In Chapter 1, "The Tailspin Scenario," you saw how the Surveys application has three different groups of users. This section describes how Tailspin can use Domain Name System (DNS) entries to manage the URLs that each group can use to access the service, and how Tailspin plans to use SSL to protect some elements of the Surveys application.

WEB ROLES IN THE SURVEYS APPLICATION

Tailspin uses web roles to deliver the user interface (UI) elements of the Surveys application. This section describes the design and implementation of these web roles.

Goals and Requirements

Three distinct groups of users will access the Surveys application: administrators at Tailspin who will manage the application, subscribers who will be creating their own custom surveys and analyzing the results, and users who will be filling in their survey responses. The first two groups will account for a very small proportion of the total number of users at any given time; the vast majority of users will be people who are filling in surveys. A large survey could have hundreds of thousands of users filling out a survey, while a subscriber might create a new survey only every couple of weeks. Furthermore, the numbers of users filling out surveys will be subject to sudden, large, short-lived spikes as subscribers launch new surveys. In addition to the different scalability requirements that arise from the two usage profiles, other requirements, such as security, will also vary.

Subscribers and administrators will be subject to the authentication and authorization controls that are described later in this chapter. It is a key requirement of the application to protect survey designs and results from unauthorized access, and the application will use a claims-based infrastructure to achieve this goal. Although some surveys might be designed for a closed user group that will require some form of authentication, many surveys may be open to the general public and will be accessible without any form of log on. Additionally, all access to the application by subscribers and administrators will use HTTPS to protect the data transferred between the application and the client. Public surveys do not require HTTPS, and this enables the use of custom URLs to access these surveys by using custom DNS CNAME entries.

Subscribers and survey respondents may be in different geographical locations. For example, a subscriber may be in the U.S. but wanting to perform some market research in Europe. Tailspin can minimize the latency for survey respondents by enabling subscribers to host their surveys in a data center located in an appropriate geographical region. However, subscribers may need to analyze the results collected from these surveys in their own geographical location.

Overview of the Solution

To make it easy for the Surveys application to meet the requirements outlined earlier, the developers at Tailspin decided to use separate web roles. One web role will contain the subscriber and administrative functionality, while a separate web role will host the surveys themselves. Tailspin can tune each web role to support its usage profile independently of the other.

Having multiple web roles in the same hosted service affects the choice URLs that you can use to access the application. Windows Azure assigns a single DNS name (for example, tailspin.cloudapp.net)

There are three distinct groups of users who will use the Surveys application.

The Windows Azure™ technology platform enables you to deploy role instances to data centers in different geographic locations.



Tailspin can host both the subscriber and survey web roles in different geographical locations. We'll talk more about this in the section "Geo Location" later in this chapter.

to a hosted service, which means that different websites within the same hosted service must have different port numbers. For example two websites within Tailspin's hosted service could have the addresses listed in the following table.

Site A	Site B
http://tailspin.cloudapp.net:80	http://tailspin.cloudapp.net:81

Because of the specific security requirements of the Surveys application, Tailspin decided to use the following URLs:

- <https://tailspin.cloudapp.net>
- <http://tailspin.cloudapp.net>

The next sections describe each of these.

<https://tailspin.cloudapp.net>

This HTTPS address uses the default port 443 to access the web role that hosts the administrative functionality for both subscribers and Tailspin. Because an SSL certificate protects this site, it is possible to map only a single, custom DNS name. Tailspin plans to use an address such as <https://surveys.tailspin.com> to access this site.

<http://tailspin.cloudapp.net>

This HTTP address uses the default port 80 to access the web role that hosts the surveys. Because there is no SSL certificate, it is possible to map multiple DNS names to this site. Tailspin will configure a default DNS name such as <http://surveys.tailspin.com> to access the surveys, but individual tenants could create their own CNAME entries to map to the site; for example, <http://surveys.adatum.com>, <http://surveys.tenant2.org>, or <http://survey.tenant3.co.de>.

It would also be possible to create separate hosted services for individual tenants that would also enable subscribers to use custom URLs for their surveys. However, this approach would complicate the provisioning process and for small subscribers, it would not be cost effective. Tailspin plans to scale out the application by adding more role instances within a hosted service.

Inside the Implementation

To implement the two different websites within a single hosted service, the developers at Tailspin defined two web roles in the solution. The first website, named TailSpin.Web, is an MVC 2 project that handles the administrative functionality within the application. This website requires authentication and authorization, and users access it using HTTPS. The second website, named Tailspin.Web.Survey.Public, is an MVC 2 project that handles users filling out surveys. This website is public, and users access it using HTTP.



Remember, you can use DNS CNAME entries to map custom domain names to the default DNS names provided by Windows Azure.



Tailspin will need to publish some guidance to subscribers that describes how they can set up their CNAMEs in their DNS settings.

The following code example shows the contents of an example ServiceDefinition.csdef file and the definitions of the two web roles:

```
<ServiceDefinition name="TailSpin.Cloud" xmlns=...>
  <WebRole name="TailSpin.Web" enableNativeCodeExecution="true">
    <InputEndpoints>
      <InputEndpoint name="HttpIn" protocol="http" port="80" />
      <InputEndpoint name="HttpsIn" protocol="https" port="443"
        certificate="tailspinweb" />
    </InputEndpoints>
    <ConfigurationSettings>
      <Setting name="DataConnectionString" />
      <Setting name="DiagnosticsConnectionString" />
    </ConfigurationSettings>
    <Certificates>
      <Certificate name="tailspinweb"
        storeLocation="LocalMachine" storeName="My" />
    </Certificates>
  </WebRole>
  <WebRole name="TailSpin.Web.Survey.Public">
    <InputEndpoints>
      <InputEndpoint name="HttpIn" protocol="http" port="80" />
      <InputEndpoint name="HttpsIn" protocol="https" port="443"
        certificate="tailspinpublicweb" />
    </InputEndpoints>
    <ConfigurationSettings>
      <Setting name="DiagnosticsConnectionString" />
      <Setting name="DataConnectionString" />
    </ConfigurationSettings>
    <Certificates>
      <Certificate name="tailspinpublicweb"
        storeLocation="LocalMachine" storeName="My" />
    </Certificates>
  </WebRole>
</ServiceDefinition>
```

Note: *This example ServiceDefinition.csdef file does not exactly match the file in the downloadable solution, which uses different names for the certificates.*

Although subscribers can access the Subscribers website (defined in the TailSpin.Web web role) only by using HTTPS, Tailspin has also

defined an HTTP endpoint. They will use the URL Rewrite Module for IIS to forward all traffic to the HTTPS endpoint on port 443. By defining the HTTP endpoint now, in the future Tailspin can choose to add non-HTTPS content to the website without deleting the Surveys application and then re-deploying it. The public website also uses the URL Rewrite Module and uses it to forward HTTPS traffic to the HTTP endpoint on port 80 for similar reasons.

Note: Remember, you may want to use different SSL certificates when you are testing the application on the development fabric. You must make sure that the configuration files reference the correct certificates before you publish the application to Windows Azure.

For more information about managing the deployment, see Chapter 7, “Application Life Cycle Management,” of the book, *Windows Azure Architecture Guide, Part 1, Moving Applications to the Cloud*. It is available at <http://msdn.microsoft.com/en-us/library/ff728592.aspx>.

In addition to the two web role projects, the solution also contains a worker role project and a library project named TailSpin.Web.Survey. Shared that contains code shared by both web roles and the worker role. This shared code includes the model classes and the data access layer.

Geo-Location

Windows Azure allows you to select a geographic location for your Windows Azure services so that you can host your application close to your users. This section describes how Tailspin uses this feature in the Surveys application.

GOALS AND REQUIREMENTS

Tailspin wants to allow subscribers to the Surveys service to be able to specify which geo-specific instance of the Surveys application they’d like to use. For example, a U.S.–based customer would probably want to choose a U.S.–based service, and a European customer would probably want to choose a European-based service. However, it’s possible that a subscriber might want to run a survey in a different geographic region than the one the subscriber is located in. Figure 1 shows how a U.S.–based subscriber might want to run a survey in Europe:



Use the URL Rewrite Module to forward traffic from unused endpoints. This will future-proof your applications in case you later decide to use these endpoints.

The Surveys application is a “geo-aware” service.



You can check the current status of any Windows Azure data center here: <http://www.microsoft.com/windowsazure/support/status/service-dashboard.aspx>.

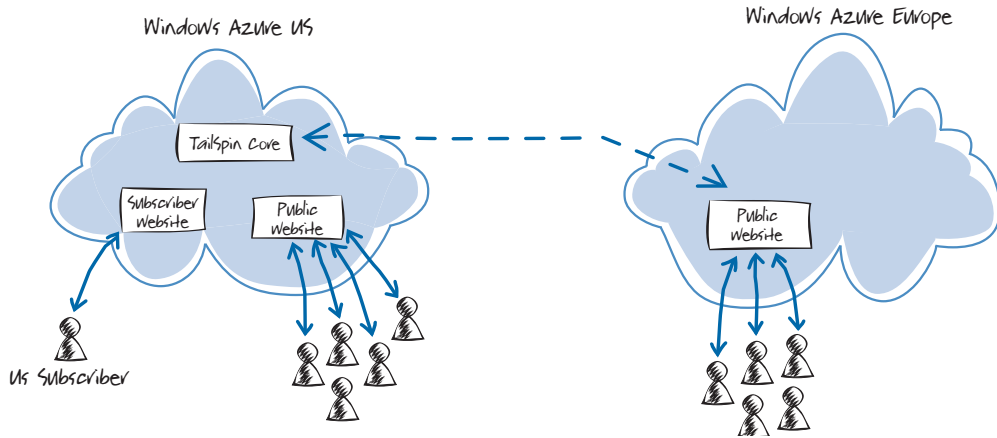


FIGURE 1
A U.S.-based subscriber running a survey in Europe

OVERVIEW OF THE SOLUTION

Hosting a survey in a web role in a different geographic location doesn't, by itself, mean that people filling out the survey will see the best response times when they use the site. To render the survey, the application must retrieve the survey definition from storage, and the application must save the completed survey results to storage. If, in the example shown in Figure 1, the application storage is in the U.S. data center, there is little benefit to European customers accessing a website hosted in the European data center.

Figure 2 shows how Tailspin designed the application to handle this scenario and resolve the issue just described.

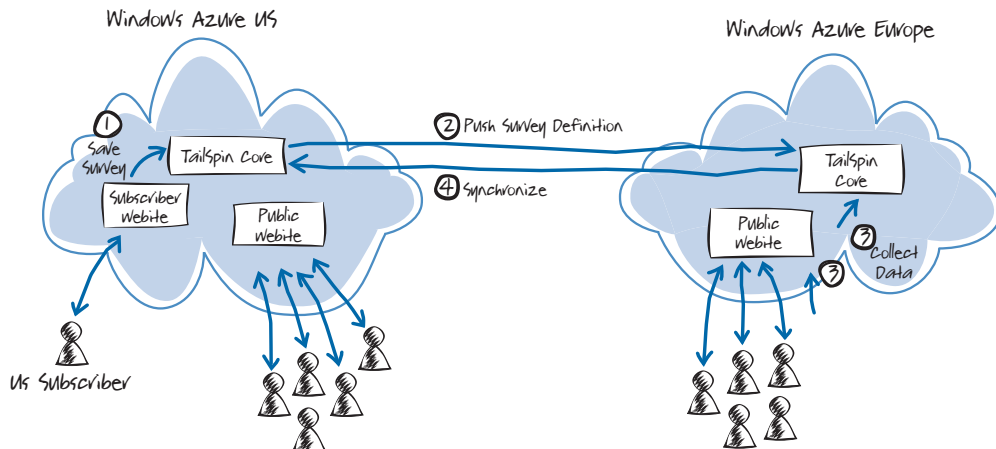


FIGURE 2
Hosting a survey in a different geographic location

The following describes the steps illustrated in Figure 2:

1. The subscriber designs the survey, and the application saves the definition in storage hosted in the U.S. data center.
2. The Surveys application pushes the survey definition to another application instance in a European data center. This needs to happen only once.
3. Survey respondents in Europe fill out the survey, and the application saves the data to storage hosted in the European data center.
4. The application transfers the survey results data back to storage in the U.S. data center, where it is available to the subscriber for analysis.

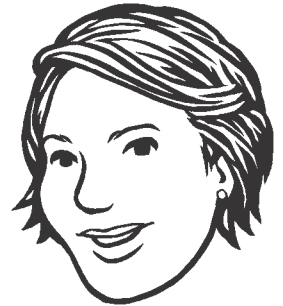
In some scenarios, it may make sense to pre-process or summarize the data in the region where it's collected and transfer back only the summarized data to reduce bandwidth costs. For the Surveys application, Tailspin decided to move all the data back to the subscriber's region; this simplifies the implementation, helps to optimize the paging feature, and ensures that each response is moved between data centers only once.

Note: *When you deploy a Windows Azure application, you can select the subregion (which, at the moment, determines the data center) where you want to host the application. You can also define affinity groups that you can use to group inter-dependent Windows Azure applications and storage accounts together in order to improve performance and reduce costs. Performance improves because Windows Azure co-locates members of the affinity group in the same data center. This reduces costs because data transfers within the same data center do not incur bandwidth charges. Affinity groups offer a small advantage over simply selecting the same subregion for your hosted services, because Windows Azure makes a "best effort" to optimize the location of those services.*

Authentication and Authorization

This section describes how Tailspin has implemented authentication and authorization in the Surveys application.

Note: *For more information about this scenario, see Chapter 6, "Federated Identity with Multiple Partners," in the book, *A Guide to Claims-Based Identity and Access Control*. This book is available for download at <http://msdn.microsoft.com/en-us/library/ff423674.aspx>.*



We'll talk more about how the application moves survey data between data centers in Chapter 5, "Working with Data in the Surveys Application."

GOALS AND REQUIREMENTS

The Tailspin Surveys application targets a wide range of customers, from large enterprises all the way down to individuals. All customers of the Surveys application will require authentication and authorization services, but they will want to implement these services differently. For example, a large enterprise customer is likely to require integration with their existing identity infrastructure, a smaller customer may not be in a position to integrate their systems and will require a basic security system as part of the Surveys application, and an individual may want to reuse an existing identity such as a Windows Live® ID or OpenID.

OVERVIEW OF THE SOLUTION

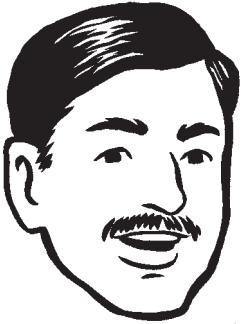
Tailspin has identified three different identity scenarios that the Surveys application must support:

- Organizations may want to integrate their existing identity infrastructure and be able to manage access to the Surveys application themselves, in order to include Surveys as a part of the Single Sign-On (SSO) experience for their employees.
- Smaller organizations may require Tailspin to provide a complete identity system because they are not able to integrate their existing systems with Tailspin.
- Individuals and small organizations may want to re-use an existing identity they have, such as a Windows Live ID or OpenID.

To support these scenarios, Tailspin uses the WS-Federation protocol to implement identity federation. At the time of writing, Access Control Services (ACS) does not implement the WS-Federation protocol, and Tailspin uses the Windows Identity Foundation (WIF) implementation of this protocol.

The following three diagrams describe how the authentication and authorization process works for each of these three scenarios.

Note: *The three scenarios are all claims-based and share the same core identity infrastructure. The only difference is the source of the original claims.*



Tailspin uses a claims-based infrastructure to provide the flexibility it needs to support its diverse customer base.

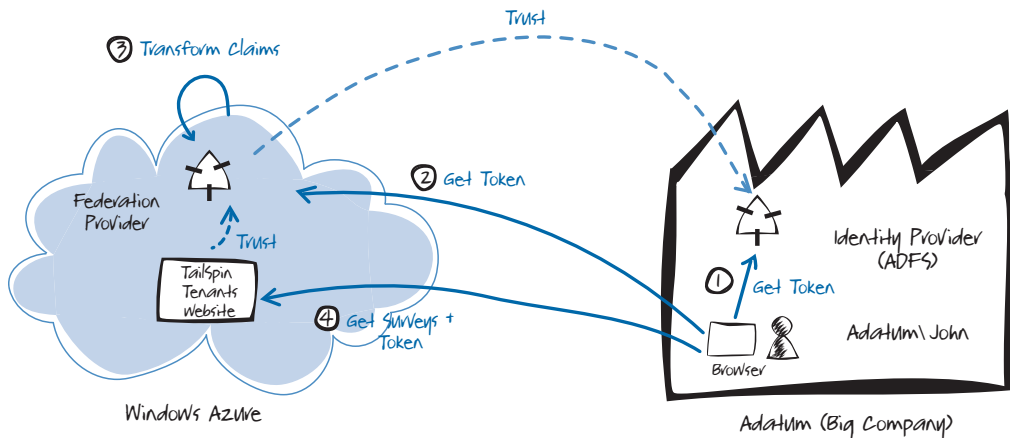


FIGURE 3
How users at a large enterprise subscriber access the Surveys application

In the scenario shown in Figure 3, users at Adatum, a large enterprise subscriber, authenticate with Adatum's own identity provider (step 1), in this case Active Directory® Federation Services (ADFS). After successfully authenticating an Adatum user, ADFS issues a token. The Tailspin federation provider trusts tokens issued by Adatum's ADFS (step 2), and if necessary can perform a transformation on the claims in the token to claims that Tailspin Surveys recognizes (step 3) before returning a token to the user. Tailspin Surveys trusts tokens issued by the Tailspin federation provider and uses the claims in the token to apply authorization rules (step 4). Users at Adatum will not need to remember separate credentials to access the Surveys application and an administrator at Adatum will be able to configure in ADFS which Adatum users have access to the Surveys application.

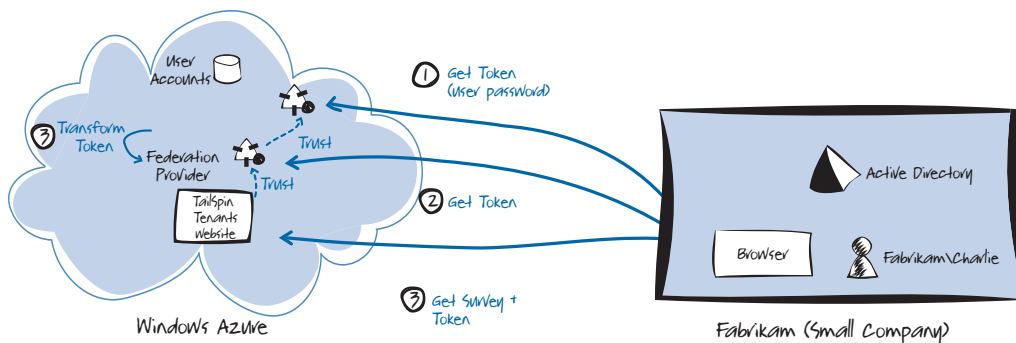


FIGURE 4
How users at a small subscriber access the Surveys application

In the scenario shown in Figure 4, users at Fabrikam, a smaller company, authenticate with the Tailspin identity provider (step 1) because their own Active Directory can't issue tokens that will be understood by the Tailspin federation provider. Other than the choice of identity provider, this approach is the same as the one used for Adatum. The downside of this approach for Fabrikam users is that they must remember a separate password to access the Surveys application.

Tailspin plans to implement this scenario by using an ASP.NET Membership Provider to manage the user accounts and to use a security token service (STS) that integrates with the membership provider.

Note: For guidance on how to implement this scenario, take a look at the Starter STS project at <http://startersts.codeplex.com>.

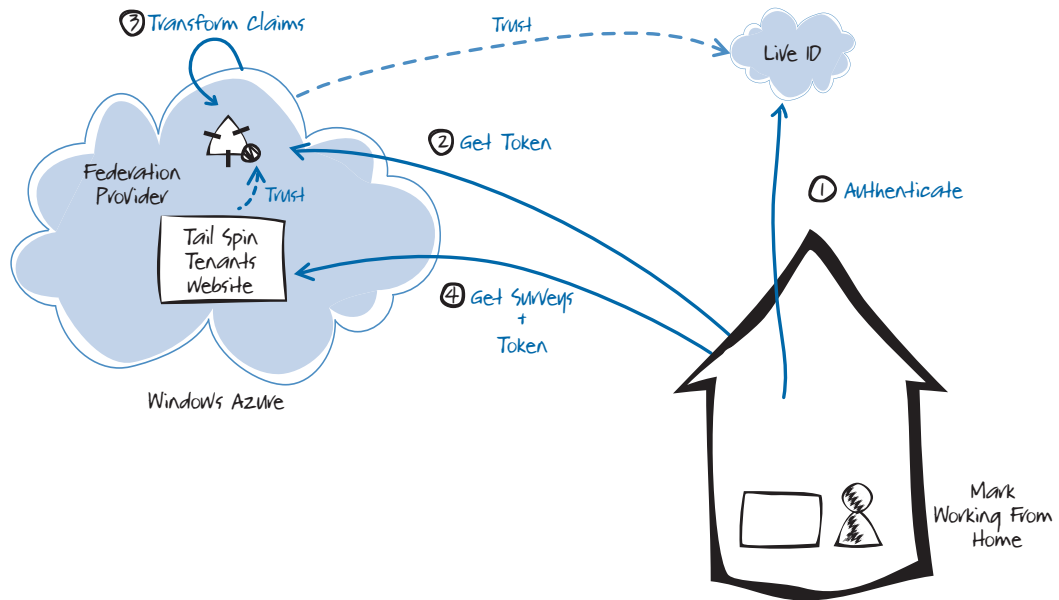


FIGURE 5
How an individual subscriber accesses the Surveys application

For individual users, the process is again very similar. In the scenario shown in Figure 5, the Tailspin federation provider is configured to trust tokens issued by a third-party provider such as Windows Live ID or OpenID (step 1). When the user tries to access their surveys, the application will redirect them to their external identity provider for authentication.

Tailspin plans to build a protocol translation STS to convert the various protocols that the third-party providers use to the protocol used by the Surveys application.

Note: For guidance on how to implement this scenario, take a look at the project named “protocol-bridge-claims-provider” at <http://github.com/southworks/protocol-bridge-claims-provider>.

INSIDE THE IMPLEMENTATION

Now is a good time to walk through the code that authentication and authorization in more detail. As you go through this section, you may want to download the Microsoft® Visual Studio® development system solution for the Tailspin Surveys application from <http://wag.codeplex.com/>. The diagram in Figure 6 will be useful to help you keep track of how this process works as you look at the detailed description and code samples later in this chapter.

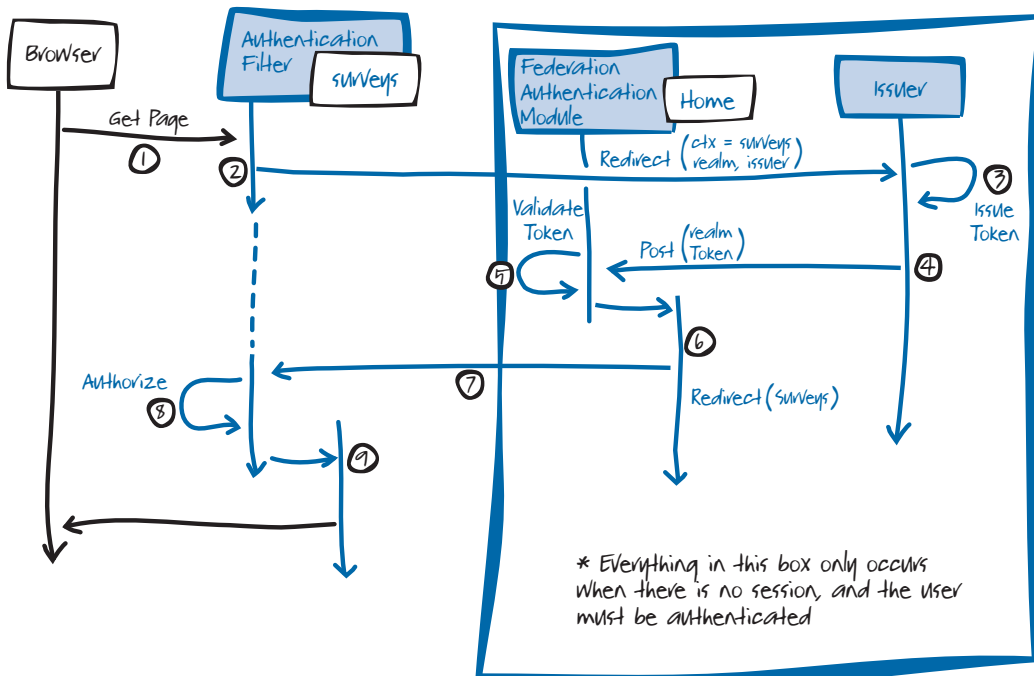


FIGURE 6
Federation with multiple partners sequence diagram

The sequence shown in this diagram applies to all three scenarios. In the context of the diagram, the Issuer is the Tailspin federation provider, so step 3 includes redirecting to another issuer to handle the authentication.



For clarity, Figure 6 shows the “logical” sequence, not the “physical” sequence. Wherever the diagram has an arrow with a **Redirect** label, this actually sends a **redirect** response back to the browser, and the browser then sends a request to wherever the **redirect** message specifies.

The following describes the steps illustrated in Figure 6:

1. The process starts when an unauthenticated user sends a request for a protected resource; for example the `adatum/surveys` page. This invokes a method in the **SurveysController** class.
2. The **AuthenticateAndAuthorizeAttribute** attribute that implements the MVC **IAuthorizationFilter** interface is applied to this controller class. Because the user has not yet been authenticated, this will redirect the user to the Tailspin federation provider at `https://localhost/TailSpin.SimulatedIssuer` with the following querystring parameter values:

```

wa. Wsignin1.0
wrealm. https://tailspin.com
wctx. https://127.0.0.1:444/survey/adatum
whr. http://adatum/trust
wreply. https://127.0.0.1:444/federationresult

```

The following code example shows the **AuthenticateUser** method in the **AuthenticateAndAuthorizeAttribute** class that builds the query string.

```

private static void AuthenticateUser(
    AuthorizationContext context)
{
    var tenantName =
        (string) context.RouteData.Values["tenant"];

    if (!string.IsNullOrEmpty(tenantName))
    {
        var returnUrl =
            GetReturnUrl(context.RequestContext);

        // User is not authenticated and is entering
        // for the first time.
        var fam = FederatedAuthentication
            .WSFederationAuthenticationModule;
        var signIn = new SignInRequestMessage(
            new Uri(fam.Issuer), fam.Realm)

```

```

{
    Context = returnUrl.ToString(),
    HomeRealm =
        RetrieveHomeRealmForTenant(tenantName)
};

// In the Windows Azure environment,
// build a wreply parameter for the SignIn
// request that reflects the real address of
// the application.
HttpRequest request = HttpContext.Current.Request;
Uri requestUrl = request.Url;

StringBuilder wreply = new StringBuilder();
wreply.Append(requestUrl.Scheme); // HTTP or HTTPS
wreply.Append(":/");
wreply.Append(request.Headers["Host"] ??
    requestUrl.Authority);
wreply.Append(request.ApplicationPath);

if (!request.ApplicationPath.EndsWith("/"))
{
    wreply.Append("/");
}

wreply.Append("FederationResult");

signIn.Reply = wreply.ToString();

context.Result = new
    RedirectResult(signIn.WriteQueryString());
}
}

```

3. The Issuer, in this case the Tailspin simulated issuer, authenticates the user and generates a token with the requested claims. In the Tailspin scenario, the Tailspin federation provider uses the value of the *whr* parameter to delegate the authentication to another issuer, in this example, to the Adatum issuer. If necessary, the Tailspin federation issuer can transform the claims it receives from the issuer to claims that the Tailspin Surveys application understands. The following code from the **FederationSecurityTokenService** class shows how the Tailspin simulated issuer transforms the **Group** claims in the token from the Adatum issuer.

```

switch (issuer.ToUpperInvariant())
{
    case "ADATUM":
        var adatumClaimTypesToCopy = new[]
        {
            WSIIdentityConstants.ClaimTypes.Name
        };
        CopyClaims(input, adatumClaimTypesToCopy, output);
        TransformClaims(input,
            AllOrganizations.ClaimTypes.Group,
            Adatum.Groups.MarketingManagers,
            ClaimTypes.Role,
            TailSpin.Roles.SurveyAdministrator, output);
        output.Claims.Add(
            new Claim(TailSpin.ClaimTypes.Tenant,
                Adatum.OrganizationName));
        break;
    case "FABRIKAM":
        ...
    default:
        throw new InvalidOperationException(
            "Issuer not trusted.");
}

```

Note: *The sample code in the simulated issuer for Tailspin contains some hard-coded names, such as Adatum and Fabrikam, and some hard-coded claim types. In a real issuer, these values would be retrieved from a configuration file or store.*

4. The Tailspin federation provider then posts the token and the value of the *wctx* parameter (<https://127.0.0.1:444/survey/adatum>) back to the address in the *wreply* parameter (<https://127.0.0.1:444/federationresult>). This address is another MVC controller (that does not have the **AuthenticateAndAuthorizeAttribute** attribute applied). The following code example shows the **FederationResult** method in the **ClaimsAuthenticationController** controller.

```

[RequireHttps]
public class ClaimsAuthenticationController : Controller
{
    [ValidateInput(false)]
    [HttpPost]
    public ActionResult FederationResult()

```

```
{
    var fam = FederatedAuthentication
        .WSFederationAuthenticationModule;
    if (fam.CanReadSignInResponse(
        System.Web.HttpContext.Current.Request, true))
    {
        string returnUrl = GetReturnUrIFromCtx();

        return this.Redirect(returnUrl);
    }

    return this.RedirectToAction(
        "Index", "OnBoarding");
}
```

5. The **WSFederationAuthenticationModule** validates the token by calling the **CanReadSignInResponse** method.
6. The **ClaimsAuthenticationController** controller retrieves the value of the original *wctx* parameter and issues a redirect to that address.
7. This time, when the request for the *adatum/surveys* page goes through the **AuthenticateAndAuthorizeAttribute** filter, the user has been authenticated. The following code example shows how the filter checks whether the user is authenticated.

```
public void OnAuthorization(
    AuthorizationContext filterContext)
{
    ...

    if (!filterContext.HttpContext.User
        .Identity.IsAuthenticated)
    {
        AuthenticateUser(filterContext);
    }
    else
    {
        this.AuthorizeUser(filterContext);
    }
}
```

8. The **AuthenticateAndAuthorizeAttribute** filter then applies any authorization rules. In the Tailspin Surveys application, the **AuthorizeUser** method verifies that the user is a member

of one of the roles listed where the **AuthenticateAndAuthorize** attribute decorates the MVC controller, as shown in the following code example.

```
[AuthenticateAndAuthorize(Roles = "Survey Administrator")]
[RequireHttps]
public class SurveysController : TenantController
{
    ...
}
```

9. The controller method finally executes.

PROTECTING SESSION TOKENS IN WINDOWS AZURE

By default, when you use the Windows Identity Foundation (WIF) framework to manage your identity infrastructure, it encrypts the contents of the cookies that it sends to the client by using the Windows Data Protection API (DPAPI). Using the DPAPI for cookie encryption is not a workable solution for an application that has multiple role instances because each role instance will have a different key, and the Windows Azure load balancer could route a request to any instance. You must use an encryption mechanism, such as RSA, that uses shared keys. The following code example shows how the Surveys application configures the session security token handler to use RSA encryption.

Note: For more information about using the DPAPI and shared key encryption mechanisms to encrypt configuration settings, see “How To: Encrypt Configuration Sections in ASP.NET 2.0 Using DPAPI” on MSDN (<http://msdn.microsoft.com/en-us/library/ff647398.aspx>).

For more information about the DPAPI, see “Windows Data Protection” on MSDN (<http://msdn.microsoft.com/en-us/library/ms995355.aspx>).

```
private void OnServiceConfigurationCreated(object sender,
    ServiceConfigurationCreatedEventArgs e)
{
    var sessionTransforms =
        new List<CookieTransform>(
            new CookieTransform[]
            {
                new DeflateCookieTransform(),
                new RsaEncryptionCookieTransform(
                    e.ServiceConfiguration.ServiceCertificate),
            }
        );
}
```



An ASP.NET web application running in a web farm would also need to use shared key encryption instead of DPAPI.

```
        new RsaSignatureCookieTransform(
            e.ServiceConfiguration.ServiceCertificate)
    });
    var sessionHandler = new
        SessionSecurityTokenHandler(sessionTransforms.AsReadOnly());
    e.ServiceConfiguration.SecurityTokenHandlers.AddOrReplace(
        sessionHandler);
}
```

The **Application_OnStart** method in the `Global.asax.cs` file hooks up this event handler to the **FederatedAuthentication** module.

Content Delivery Network

The Windows Azure Content Delivery Network (CDN) allows you to have binary large object (BLOB) content cached at strategic locations around the world in order to make that content available with the maximum possible bandwidth to users and minimize any latency. The CDN is designed to be used with BLOB content that is relatively static. For the Surveys application, the developers at Tailspin have identified two scenarios where they could use the CDN:

The CDN enables you to have data that is stored in BLOBs cached at strategic locations around the world.

- Tailspin is planning to commission a set of training videos with titles such as “Getting Started with the Surveys Application,” “Designing Great Surveys,” and “Analyzing your Survey Results.”
- Hosting the custom images and style sheets that subscribers upload.

In both of these scenarios, users will access the content many times before it’s updated. The training videos are likely to be refreshed only when the application undergoes a major upgrade, and Tailspin expects subscribers to upload standard corporate logos and style sheets that reflect corporate branding. Both of these scenarios will also account for a significant amount of bandwidth used by the application. Online videos will require sufficient bandwidth to ensure good playback quality, and every request to fill out a survey will result in a request for a custom image and style sheet.

One of the requirements for using the CDN is that the content must be in a BLOB container that you configure for public, anonymous access. Again, in both of the scenarios, the content is suitable for unrestricted access.

For information about the current pricing for the CDN, visit the Windows Azure Platform website at <http://www.microsoft.com/windowsazure/offers/>.

Note: For data cached on the CDN, you are charged for outbound transfers based on the amount of bandwidth you use and the number of transactions. You are also charged at the standard Windows Azure BLOB storage rates for the transfers that move data from BLOB storage to the CDN. Therefore, it makes sense to use the CDN for relatively static content. With highly dynamic content, you could, in effect pay double, because each request for data from the CDN triggers a request for the latest data from BLOB storage.

THE SOLUTION

To use the CDN with the Surveys application, Tailspin will have to make a number of changes to the application. The following sections describe these changes. This section describes the planned solution; the current version of the Surveys application does not include the use of the CDN.

Setting the Access Control for the BLOB Containers

Any BLOB data that you want to host on the CDN must be in a BLOB container with permissions set to allow full public read access. You can set this option when you create the container by calling the **BeginCreate** method of the **CloudBlobContainer** class or by calling the **SetPermissions** method on an existing container. The following code shows an example of how to set the permissions for a container.

```
protected void SetContainerPermissions(String containerName)
{
    CloudStorageAccount cloudStorageAccount =
        CloudStorageAccount.FromConfigurationSetting(
            "DataConnectionString");
    CloudBlobClient cloudBlobClient =
        cloudStorageAccount.CreateCloudBlobClient();

    CloudBlobContainer cloudBlobContainer =
        new CloudBlobContainer(containerName, cloudBlobClient);

    BlobContainerPermissions blobContainerPermissions =
        new BlobContainerPermissions();
    blobContainerPermissions.PublicAccess =
        BlobContainerPublicAccessType.Container;
    cloudBlobContainer.SetPermissions(blobContainerPermissions);
}
```

Notice that the permission type used to set full public access is **BlobContainerPublicAccessType.Container**.

Configuring the CDN and Storing the Content

You configure the CDN at the level of a Windows Azure storage account through the Windows Azure Developer Portal. After you enable CDN delivery for a storage account, any data in public BLOB containers is available for delivery by the CDN.

The application must place all the content to be hosted on the CDN into BLOBs in the appropriate containers. In the Surveys application, media files, custom images, and style sheets can all be stored in these BLOBs. For example, if a training video is packaged with a player application in the form of some HTML files and scripts, all of these related files can be stored as BLOBs in the same container.

Note: *You must be careful if scripts or HTML files contain relative paths to other files in the same BLOB container because the path names will be case sensitive. This is because there is no real folder structure within a BLOB container, and any “folder names” are just a part of the file name in a single, flat namespace.*

Configuring URLs to Access the Content

Windows Azure allocates URLs to access BLOB data based on the account name and the container name. For example, if Tailspin created a public container named “video” for hosting their training videos, you could access the “Getting Started with the Surveys Application” video directly in Windows Azure BLOB storage at <http://tailspin.blob.core.windows.net/video/gettingstarted.html>. This assumes that the `gettingstarted.html` page is a player for the media content. The CDN provides access to hosted content using a URL in the form `http://<uid>.vo.msecnd.net/`, so the Surveys training video would be available on the CDN at `http://<uid>.vo.msecnd.net/video/gettingstarted.html`.

Figure 7 illustrates this relationship between the CDN and BLOB storage.

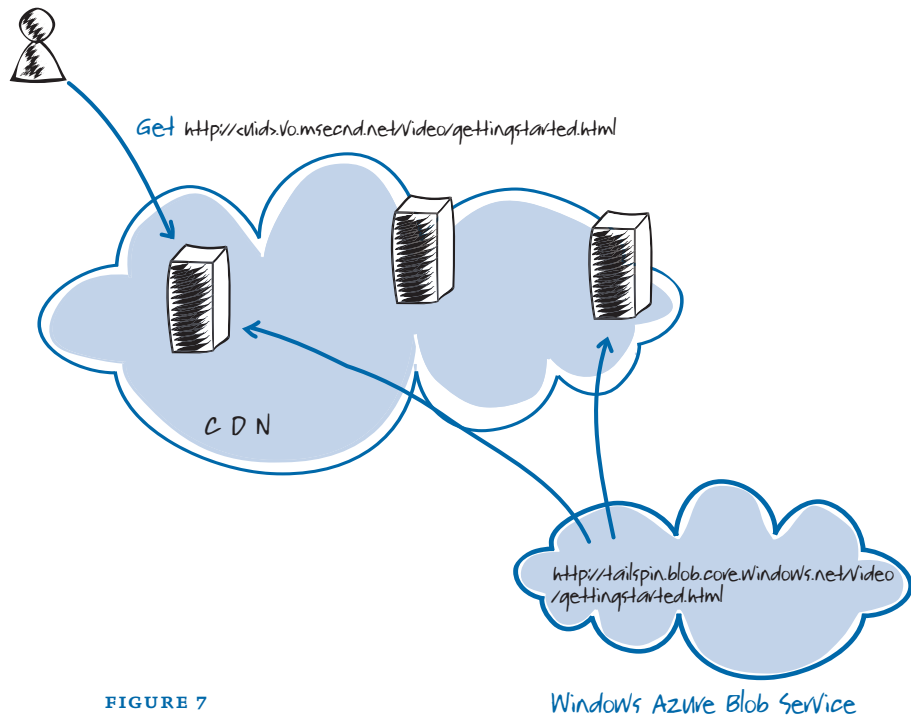


FIGURE 7
The Content Delivery Network

You can configure a CNAME entry in DNS to map a custom URL to the CDN URL. For example, Tailspin might create a CNAME entry to make `http://files.tailspin.com/video/gettingstarted.html` point to the video hosted on the CDN. You should verify that your DNS provider configures the DNS resolution to behave efficiently; the performance benefits of using the CDN could be offset if the name resolution of your custom URL involves multiple hops to a DNS authority in a different geographic region.

Note: *In addition to creating the CNAME entry for your custom domain name in the tool you use for managing your DNS entries, you must also configure the custom domain name in the storage account settings. You should use the Custom Domains section on the Summary Page in the Windows Azure Developer Portal to complete this task.*

When a user requests content from the CDN, Windows Azure automatically routes their request to the closest available CDN endpoint. If the BLOB data is found at that endpoint, it's returned to the user. If the BLOB data is not found at the endpoint, it's automatically retrieved from BLOB storage before being returned to the user and cached at the endpoint for future requests.

Setting the Caching Policy

All BLOBs cached by the CDN have a time-to-live (TTL) period that determines how long they will remain in the cache before the CDN goes back to BLOB storage to check for updated data. The default caching policy used by the CDN uses an algorithm to calculate the TTL for cached content based on when the content was last modified in BLOB storage. The longer the content has remained unchanged in BLOB storage, the greater the TTL, up to a maximum of 72 hours.

Note: *The CDN retrieves content from BLOB storage only if it is not in the endpoint's cache, or if it has changed in BLOB storage.*

You can also explicitly set the TTL by using the **CacheControl** property of the **BlobProperties** class. The following code example shows how to set the TTL to two hours.

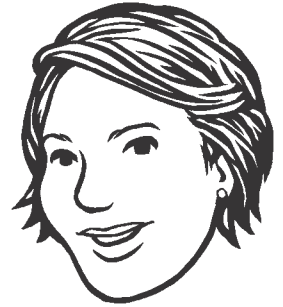
```
blob.Properties.CacheControl = "max-age=7200";
```

More Information

For more information about using CNAME entries in DNS, see the post, "Custom Domain Names in Windows Azure," on Steve Marx's blog:

<http://blog.smarx.com/posts/custom-domain-names-in-windows-azure>

For more information about the claims-based authentication and authorization model used in the Surveys application, see Chapter 6, "Federated Identity with Multiple Partners," of the book, *A Guide to Claims-Based Identity and Access Control*. You can download a PDF copy of this book from MSDN:



If the BLOB data is not found at the endpoint, you will incur Windows Azure storage charges when the CDN retrieves the data from blob storage.

<http://msdn.microsoft.com/en-us/library/ff423674.aspx>

For a walkthrough of how to secure an ASP.NET site on Windows Azure with WIF, see “Exercise 1: Enabling Federated Authentication for ASP.NET applications in Windows Azure” on Channel 9:

<http://channel9.msdn.com/learn/courses/Azure/IdentityAzure/WIFonWAZLab/Exercise-1-Enabling-Federated-Authentication-for-ASPNET-applications-in-Windows-Azure/>

For more information about the CDN, see “Delivering High-Bandwidth Content with the Windows Azure Content Delivery Network” on MSDN:

<http://msdn.microsoft.com/en-us/library/ee795176.aspx>

For information about an application that uses the CDN, see the post, “EmailTheInternet.com: Sending and Receiving Email in Windows Azure,” on Steve Marx’s blog:

<http://blog.smarx.com/posts/emailtheinternet-com-sending-and-receiving-email-in-windows-azure>

For an episode of Cloud Cover that covers CDN, see Channel 9:

<http://channel9.msdn.com/shows/Cloud+Cover/Cloud-Cover-Episode-4-CDN/>

4

Building a Scalable, Multi-Tenant Application for Windows Azure

This chapter examines architectural and implementation issues in the Surveys application from the perspective of building a multi-tenant application. Questions such as how to partition the application and how to bill customers for their usage are directly relevant to a multi-tenant architecture. Questions such as how to make the application scalable and how to handle the on-boarding process for new subscribers are relevant to both single-tenant and multi-tenant architectures, but they involve some special considerations in multi-tenant model.

This chapter describes how Tailspin resolved these questions for the Surveys application. For other applications, different choices may be appropriate.

Partitioning the Application

Chapter 5, “Data Storage in the Surveys Application,” describes how the Surveys application data model partitions the data by subscriber. This section describes how the Surveys application uses MVC routing tables and areas to make sure that a subscriber sees only his or her own data.

THE SOLUTION

The developers at Tailspin decided to use the path in the application’s URL to indicate which subscriber is accessing the application. For the Subscriber website, users must authenticate before they can access the application, for the public Surveys website, the application doesn’t require authentication.

The following are three sample paths on the Subscriber website:

- /survey/adatum/newsurvey
- /survey/adatum/newquestion
- /survey/adatum

The URL path identifies the functional area in the application, the subscriber, and the action.

The following are two example paths on the public Surveys website:

- /survey/adatum/launch-event-feedback
- /survey/adatum/launch-event-feedback/thankyou

The application uses the first element in the path to indicate the different areas of functionality within the application. All the preceding examples relate to surveys, but other areas relate to on-boarding and security. The second element indicates the subscriber name, in these examples “Adatum,” and the last element indicates the action that is being performed, such as creating a new survey or adding a question to a survey.

You should take care when you design the path structure for your application that there is no possibility of name clashes that result from a value entered by a subscriber. In the Surveys application, if a subscriber creates a survey named “newsurvey,” the path to this survey is the same as the path to the page subscribers use to create new surveys. However, the application hosts surveys on an HTTP endpoint and the page to create surveys on an HTTPS endpoint, so there is no name clash in this particular case.

Note: *The third example element of the public Surveys website, “launch-event-feedback,” is a “slugified” version of the survey title, originally “Launch Event Feedback,” to make it URL friendly.*

INSIDE THE IMPLEMENTATION

Now is a good time to walk through the code that handles the request routing within the application in more detail. As you go through this section, you may want to download the Microsoft® Visual Studio® development system solution for the Tailspin Surveys application from <http://wag.codeplex.com/>.

The implementation uses a combination of ASP.NET routing tables and MVC areas to identify the subscriber and map requests to the correct functionality within the application.

The following code example shows how the public Surveys Web site uses routing tables to determine which survey to display based on the URL.

```
using System.Web.Mvc;
using System.Web.Routing;

public static class AppRoutes
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.MapRoute(
            "Home",
```



A slug name is a string where all whitespace and invalid characters are replaced with a hyphen (-). The term comes from the newsprint industry and has nothing to do with those things in your garden!

```

        string.Empty,
        new { controller = "Surveys", action = "Index" });

routes.MapRoute(
    "ViewSurvey",
    "survey/{tenant}/{surveySlug}",
    new { controller = "Surveys", action = "Display" });

routes.MapRoute(
    "ThankYouForFillingTheSurvey",
    "survey/{tenant}/{surveySlug}/thankyou",
    new { controller = "Surveys", action = "ThankYou" });
}
}

```

The code extracts the tenant name and survey name from the URL and passes them to the appropriate action method in the **Surveys Controller** class. The following code example shows the **Display** action method that handles HTTP Get requests.

```

[HttpGet]
public ActionResult Display(string tenant, string surveySlug)
{
    var surveyAnswer = CallGetSurveyAndCreateSurveyAnswer(
        this.surveyStore, tenant, surveySlug);

    var model = new
        TenantPageViewData<SurveyAnswer>(surveyAnswer);
    model.Title = surveyAnswer.Title;
    return this.View(model);
}

```

If the user requests a survey using a URL with a path value of `/survey/adatum/launch-event-feedback`, the value of the *tenant* parameter will be “adatum” and the value of the *surveySlug* parameter will be “launch-event-feedback.” This action method uses the parameter values to retrieve the survey definition from the store, populate the model with this data, and pass the model to the view that renders it to the browser.

The Subscriber website is more complex because it must handle authentication and on-boarding new subscribers in addition to enabling subscribers to design new surveys and analyze survey results. Because of this complexity, it uses MVC areas as well as a routing table. The following code from the **AppRoutes** class in the TailSpin.Web project shows how the application maps top-level requests to the controller classes that handle on-boarding and authentication.



There is also a Display action to handle HTTP POST requests. This controller action is responsible for saving the filled out survey data.



MVC areas enable you to group multiple controllers together within the application, making it easier work with large MVC projects. Each area typically represents a different function within the application.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.MapRoute(
        "OnBoarding",
        string.Empty,
        new { controller = "OnBoarding", action = "Index" });

    routes.MapRoute(
        "FederationResultProcessing",
        "FederationResult",
        new { controller = "ClaimsAuthentication",
            action = "FederationResult" });

    routes.MapRoute(
        "FederatedSignout",
        "Signout",
        new { controller = "ClaimsAuthentication",
            action = "Signout" });
}
...
}
```

The application also defines an MVC area for the core survey functionality. MVC applications register areas by calling the **RegisterAllAreas** method. In the TailSpin.Web project, you can find this call in the **Application_Start** method in the Global.asax.cs file. The **RegisterAllAreas** method searches the application for classes that extend the **AreaRegistration** class, and then it invokes the **RegisterArea** method. The following code example shows a part of this method in the **SurveyAreaRegistration** class.

```
public override void RegisterArea(
    AreaRegistrationContext context)
{
    context.MapRoute(
        "MySurveys",
        "survey/{tenant}",
        new { controller = "Surveys", action = "Index" });
}
```

```
context.MapRoute(
    "NewSurvey",
    "survey/{tenant}/newsurvey",
    new { controller = "Surveys", action = "New" });

context.MapRoute(
    "NewQuestion",
    "survey/{tenant}/newquestion",
    new { controller = "Surveys", action = "NewQuestion" });

context.MapRoute(
    "AddQuestion",
    "survey/{tenant}/newquestion/add",
    new { controller = "Surveys", action = "AddQuestion" });

...
}
```

Notice how all the routes in this routing table include the tenant name that MVC passes as a parameter to the controller action methods.

On-Boarding for Trials and New Customers

Whenever a new subscriber signs up for the Surveys service, the application must perform configuration tasks to enable the new account. Tailspin wants to automate as much of this process as possible to simplify the on-boarding process for new customers and minimize the costs associated with setting up a new subscriber. The on-boarding process touches many components of the Surveys application, and this section describes how the on-boarding process affects those components.

The on-boarding process touches many components in the Surveys application.

BASIC SUBSCRIPTION INFORMATION

The following table describes the basic information that every subscriber provides when they sign up for the Surveys service.

Information	Example	Notes
Subscriber Name	Adatum Ltd.	The commercial name of the subscriber. The application uses this as part of customization of the subscriber's pages on the Surveys websites. The Subscriber can also provide a corporate logo.
Subscriber Alias	adatum	A unique alias used within the application to identify the subscriber. For example, it forms part of the URL for the subscriber's web pages. The application generates a value based on the Subscriber Name, but it allows the subscriber to override this suggestion.
Subscription Type	Trial, Individual, Standard, Premium	The subscription type determines the feature set available to the subscriber and may affect what additional on-boarding information must be collected from the subscriber.
Payment Details	Credit card details	Apart from a trial subscription, all other subscription types are paid subscriptions. The application uses a third-party solution to handle credit card payments.

Apart from credit card details, all this information is stored in Windows Azure™ storage; it is used throughout the on-boarding process and while the subscription is active.

AUTHENTICATION AND AUTHORIZATION INFORMATION

The section, "Authentication and Authorization," in Chapter 3, "Accessing the Surveys Application," of this book describes the three alternatives for managing access to the application. Each of these alternatives requires different information from the subscriber as part of the on-boarding process, and each alternative is associated with a different subscription type. For example, the Individual subscription type uses a social identity provider, such as Windows Live® ID or Google ID, for authentication, and the Premium subscription type uses the subscriber's own identity provider.

Provisioning a Trust Relationship with the Subscriber's Identity Provider

One of the features of the Premium subscription type is integration with the subscriber's identity provider. The on-boarding process collects the information needed to configure the trust relationship between subscriber's Security Token Service (STS) and the Tailspin federation provider (FP) STS. The following table describes this information.

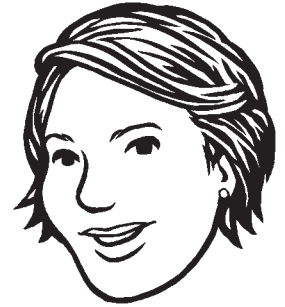
Information	Example	Notes
Subscriber Federation-Metadata URL	<code>https://login.adatum.net/FederationMetadata/2007-06/FederationMetadata.xml</code>	This should be a public endpoint. An alternative is to enable the subscriber to manually upload this data.
Administrator identifier (email or Security Account Manager Account Name)	<code>john@adatum.com</code>	The Surveys application creates a rule in its FP to map this identifier to the administrator role in the Surveys application.
User identifier claim type	<code>http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name</code>	This is the claim type that the subscriber's STS will issue to identify a user.
Subscriber's public key	<code>adatum.cer</code>	The subscriber can provide a certificate if they want to encrypt their tokens.
Claims transformation rules	<code>Group:Domain Users => Role:Survey Creator</code>	These rules map a subscriber's claim types to claim types understood by the Surveys application.

The Surveys application will use this data to add the appropriate configuration information to the Tailspin FP STS. The on-boarding process will also make the Tailspin FP federation metadata available to the subscriber because the subscriber may need it to configure the trust relationship in their STS.

Note: For more information, see the section, "Setup and Physical Deployment," on page 97 of the book, *A Guide to Claims-Based Identity and Access Control*. You can download a PDF copy of this book at <http://msdn.microsoft.com/en-us/library/ff423674.aspx>.

Provisioning Authentication and Authorization for Basic Subscribers

Subscribers to the Standard subscription type cannot integrate the Surveys application with their own STS. Instead, they can define their own users in the Surveys application. During the on-boarding process, they provide details for the administrator account that will have full access to everything in their account, including billing information. They can later define additional users who are members of the Survey Creator role, who can only create surveys and analyze the results.



The application does not yet implement this functionality. Tailspin could decide to use ADFS, ACS, or a custom STS as its federation provider. As part of the on-boarding process, the Surveys application will have to programmatically create the trust relationship between the Tailspin FP STS and the customer's identity provider, and programmatically add any claims transformation rules to the Tailspin STS.

Provisioning Authentication and Authorization for Individual Subscribers

Individual subscribers use a third-party, social identity, such as a Windows Live ID, OpenID, or Google ID, to authenticate with the Surveys application. During the on-boarding process, they must provide details of the identity they will use. This identity has administrator rights for the account and is the only identity that can be used to access the account.

GEO LOCATION INFORMATION

During the on-boarding process, the subscriber selects the geographic location where the Surveys application will host their account. The list of locations to choose from is the list of locations where there are currently Windows Azure data centers. This geographic location identifies the location of the Subscriber website instance that the subscriber will use and where the application stores all the data associated with the account. It is also the default location for hosting the subscriber's surveys, although the subscriber can opt to host individual surveys in alternate geographical locations.

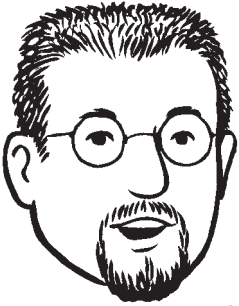
DATABASE INFORMATION

During the sign-up process, subscribers can also opt to provision a SQL Azure™ database to store and analyze their survey data. The application creates this database in the same geographical locations as the subscribers' accounts. The application uses the subscriber alias to generate the database name and the database user name. The application also generates a random password. The application saves the database connection string in Windows Azure storage, together with the other subscriber account data.

Note: *The SQL Azure database is still owned and paid for by Tailspin. Tailspin charges subscribers for this service. For more information about how the Surveys application uses SQL Azure, see the section, "Using SQL Azure," in Chapter 5, "Working with Data in the Surveys Application," of this book.*

Billing Customers

Tailspin plans to bill each customer a fixed monthly fee to use the Surveys application. Customers will be able to subscribe to one of several packages, as outlined in the following table.



You could automatically suggest a location based on the user's IP address by using a service such as http://ipinfodb.com/ip_location_api.php.

Subscription type	User accounts	Maximum survey duration	Maximum active surveys
Trial	A single user account linked to a social identity provider, such as Windows Live ID or OpenID.	5 days	1
Basic	A single user account linked to a social identity provider, such as Windows Live ID or OpenID.	14 days	1
Standard	Up to five user accounts provided by the Surveys application.	28 days	10
Premium	Unlimited user accounts linked from the subscriber's own identity provider.	56 days	20

The advantage of this approach is simplicity for both Tailspin and the subscribers, because the monthly charge is fixed for each subscriber. Tailspin must undertake some market research to estimate the number of monthly subscribers at each level so that they can set appropriate charges for each subscription level.

In the future, Tailspin wants to be able to offer extensions to the basic subscription types. For example, Tailspin wants to enable subscribers to extend the duration of a survey beyond the current maximum, or to increase the number of active surveys beyond the current maximum. To do this, Tailspin will need to be able to capture usage metrics from the application to help it calculate any additional charges incurred by a subscriber. Tailspin expects that forthcoming Windows Azure APIs that expose billing information and storage usage metrics will simplify the implementation of these extensions.

Note: *At the time of writing, the best approach to capturing usage metrics is via logging. Several log files are useful. You can use the Internet Information Services (IIS) logs to determine which tenant generated the web role traffic. Your application can write custom messages to the WADLogsTable. The `sys.bandwidth_usage` view in the master database of each SQL Azure server shows bandwidth consumption by database.*



Tailspin must have good estimates of expected usage to be able to estimate costs, revenue, and profit.

Customizing the User Interface

A common feature of multi-tenant applications is enabling subscribers to customize the appearance of the application for their customers. The current version of the Surveys application enables subscribers to customize the appearance of their account page by using a custom logo image. Subscribers can upload an image to their account, and the Surveys application saves the image as part of the subscriber's account data in BLOB storage.

Tailspin plans to extend the customization options available to subscribers in future versions of the application. These extensions include customizing the survey pages with the logo and enabling subscribers to upload a cascading style sheets (.css) file to customize the appearance of their survey pages to follow corporate branding schemes.

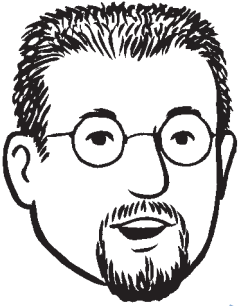
Tailspin are evaluating the security implications of allowing subscribers to upload custom .css files and plan to limit the cascading style sheets features that the site will support. They will implement a scanning mechanism to verify that the .css files that subscribers upload do not include any of the features that the Surveys site does not support.

The current solution allows subscribers to upload an image to a public BLOB container named logos. As part of the upload process, the application adds the URL for the logo image to the tenant's BLOB data stored in the BLOB container named tenants. The **Tenant Controller** class retrieves the URL and forwards it on to the view.

Scaling Applications by Using Worker Roles

Scalability is an issue for both single-tenant and multi-tenant architectures. Although it may be acceptable to allow certain operations at certain times to utilize most of the available resources in a single-tenant application (for example, calculating aggregate statistics over a large dataset at 2:00 A.M.), this is not an option for most multi-tenant applications where different tenants have different usage patterns.

You can use worker roles in Windows Azure to offload resource-hungry operations from the web roles that handle user interaction. These worker roles can perform tasks asynchronously when the web roles do not require the output from the worker role operations to be immediately available.



Cascading style sheets Behaviors are one feature that the Surveys site will not support.

EXAMPLE SCENARIOS FOR WORKER ROLES

The following table describes some example scenarios where you can use worker roles for asynchronous job processing. Not all of these scenarios come from the Surveys application; but, for each scenario, the table specifies how to trigger the job and how many worker role instances it could use.

Scenario	Description	Solution
Update survey statistics	The survey owner wants to view the summary statistics of a survey, such as the total number of responses and average scores for a question. Calculating these statistics is a resource intensive task.	<p>Every time a user submits a survey response, the application puts a message in a queue named statistics-queue with a pointer to the survey response data.</p> <p>Every 10 minutes, a worker retrieves the pending messages from the statistics-queue queue and adjusts the survey statistics to reflect those survey responses. Only one worker instance should do the calculation over a queue to avoid any concurrency issues when it updates the statistics table.</p> <p>Triggered by: time Execution model: single worker</p>
Dump survey data to SQL Azure database	The survey owner wants to analyze the survey data using a relational database. Transferring large volumes of data is a time consuming operation.	<p>The survey owner requests the back end to export the responses for a survey. This action creates a row in a table named exports and puts a message in a queue named export-queue pointing to that row. Any worker can de-queue messages from export-queue and execute the export. After it finishes, it updates the row in the <i>exports</i> table with the status of the export procedure.</p> <p>Triggered by: message in queue Execution model: multiple workers</p>
Store a survey response	Every time a respondent completes a survey, the response data must be reliably persisted to storage. The user should not have to wait while the application persists the survey data.	<p>Every time a user submits a survey response, the application writes the raw survey data to BLOB storage and puts a message in a queue named responses-queue.</p> <p>A worker role polls the responses-queue queue and when a new message arrives, it stores the survey response data in table storage and puts a message in the statistics-queue queue to calculate statistics.</p> <p>Triggered by: message in queue Execution model: multiple workers</p>
Heartbeat	Many workers running in a grid-like system have to send a "ping" at a fixed time interval to indicate to a controller that they are still active. The heartbeat message must be sent reliably without interrupting the worker's main task.	<p>Every minute, each worker executes a piece of code that sends a "ping."</p> <p>Triggered by: time Execution model: multiple workers</p>

Note: You can scale the Update Survey Statistics scenario described in the preceding table by using one queue and one worker role instance for every tenant or even every survey. What is important is that only one worker role instance should process and update data that is mutually exclusive within the dataset.

Looking at these example scenarios suggests that you can categorize worker roles that perform background processing according to the scheme in the following table.

Trigger	Execution	Types of tasks
Time	Single worker	An operation on a set of data that updates frequently and requires an exclusive lock to avoid concurrency issues. Examples include aggregation, summarization, and denormalization.
Time	Multiple workers	An operation on a set of data that is mutually exclusive from other sets so that there are no concurrency issues. Independent operations that don't work over data such as a "ping."
Message in a queue	Single or multiple workers	An operation on a small number of resources (for example, a BLOB or several table rows) that should start as soon as possible.

Triggers for Background Tasks

The trigger for a background task could be a timer or a signal in the form of a message in a queue. Time-based background tasks are appropriate when the task must process a large quantity of data that trickles in little by little. This approach is cheaper and will offer higher throughput than an approach that processes each piece of data as it becomes available. This is because you can batch the operations and reduce the number of storage transactions required to process the data.

If the frequency at which new items of data becomes available is lower and there is a requirement to process the new data as soon as possible, using a message in a queue as a trigger is appropriate.

You can implement a time-based trigger by using a **Timer** object in a worker role that executes a task at fixed time interval. You can implement a message-based trigger in a worker role by creating an infinite loop that polls a message queue for new messages. You can retrieve either a single message or multiple messages from the queue and execute a task to process the message or messages.



You can pull multiple messages from a queue in a single transaction.

Execution Model

In Windows Azure, you process background tasks by using worker roles. You could have a separate worker role type for each type of background task in your application, but this approach means that you will need at least one separate worker role instance for each type of task. Often, you can make better use of the available compute resources by having one worker role handle multiple types of tasks, especially when you have high volumes of data because this approach reduces the risk of under-utilizing your compute nodes. This approach, often referred to as role conflation, involves two trade-offs. The first trade-off balances the complexity of and cost of implementing role conflation against the potential cost savings that result from reducing the number of running worker role instances. The second trade-off is between the time required to implement and test a solution that uses role conflation and other business priorities, such as time-to-market. In this scenario, you can still scale out the application by starting up additional instances of the worker role. The diagrams in Figure 1 show these two scenarios.

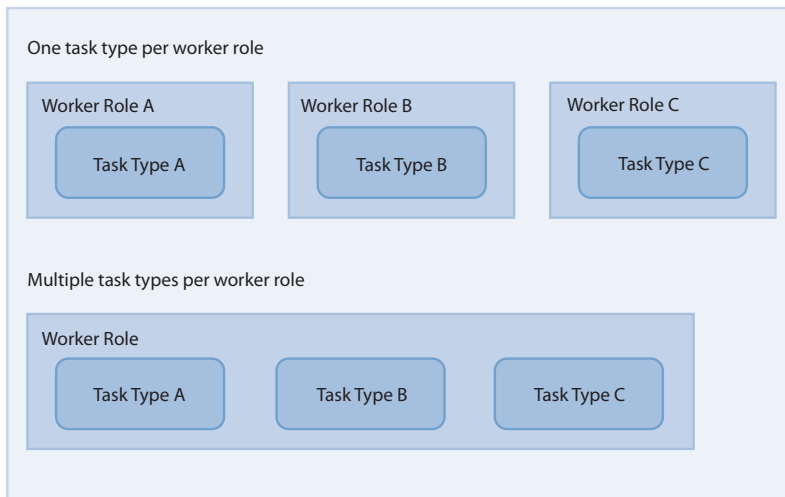


FIGURE 1
Handling multiple background task types

In the scenario where you have multiple instances of a worker role that can all execute the same set of task types, you need to distinguish between the task types where it is safe to execute the task in multiple worker roles simultaneously, and the task types where it is only safe to execute the task in a single worker role at a time.

To ensure that only one copy of a task can run at a time, you must implement a locking mechanism. In Windows Azure, you could use a message on a queue or a lease on a BLOB for this purpose. The diagram in Figure 2 shows that multiple copies of Tasks A and C can run simultaneously, but only one copy of Task B can run at any one time. One copy of Task B acquires a lease on a BLOB and runs; other copies of Task B will not run until they can acquire the lease on the BLOB.

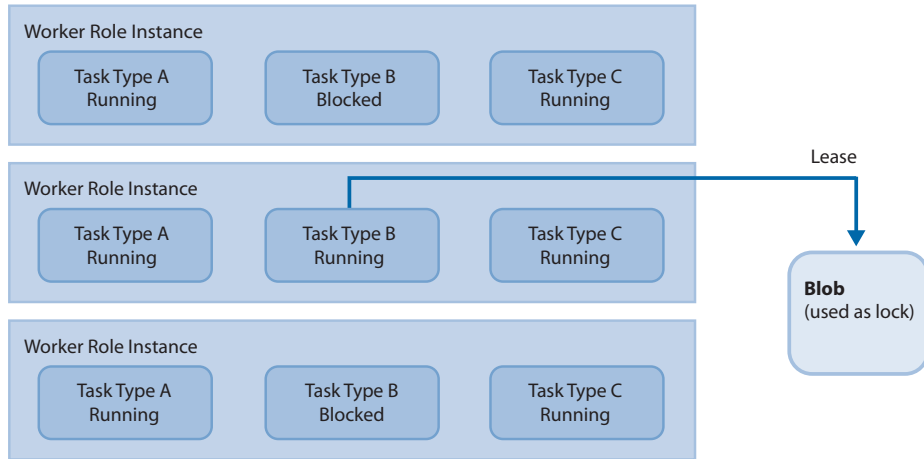


FIGURE 2
Multiple worker role instances

The MapReduce Algorithm

For some Windows Azure applications, being limited to a single task instance for certain large calculations may have a significant impact on performance. In these circumstances, the MapReduce algorithm may provide a way to parallelize the calculations across multiple task instances in multiple worker roles.

The original concepts behind MapReduce come from the **map** and **reduce** functions that are widely used in functional programming languages such as Haskell, F#, and Erlang. In the current context, MapReduce is a programming model (patented by Google), that enables you to parallelize operations on a large dataset. In the case of the Surveys application, you could use this approach to calculate the summary statistics by using multiple, parallel tasks instead of a single task. The benefit would be to speed up the calculation of the summary statistics, but at the cost of having multiple worker role instances.



For the Surveys application, speed is not a critical factor for the calculation of the summary statistics. Tailspin is willing to tolerate a delay while this summary data is calculated, so it does not use MapReduce.

The following example shows how Tailspin could use this approach if it wants to speed up the calculation of the summary statistics.

This example assumes that the application saves survey responses in BLOBs that contain the data shown in Figure 3.

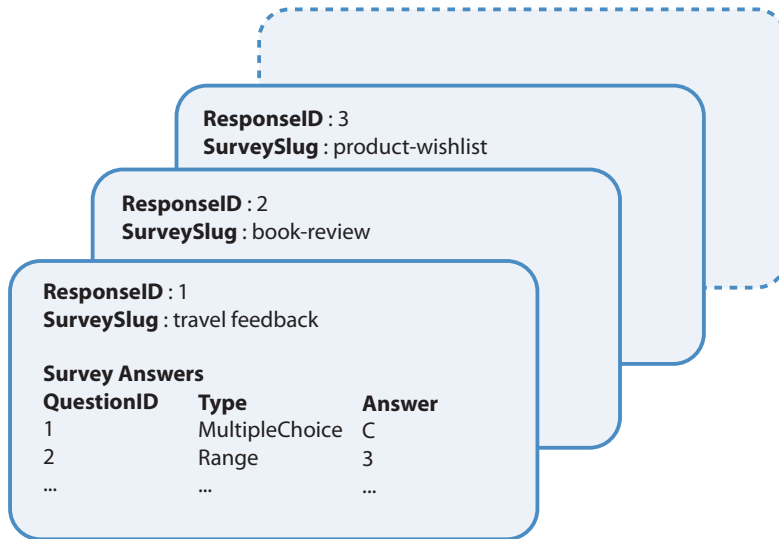


FIGURE 3
 Example BLOBs containing survey response data

The following table shows the initial set of data from which the application must calculate the summary statistics. In practice, MapReduce is used to process very large datasets; this example uses a very small dataset to show how MapReduce works. This example also only shows the summarization steps for the first multiple-choice question and the first range question found in the survey answers, but you could easily extend the process to handle all the questions in each survey.

ResponseID	SurveySlug	Answer to first multiple choice question in survey	Answer to first range question in survey	Answers to other questions
1	travel-feedback	C	3	...
2	book-review	D	3	...
3	product-wishlist	A	4	...
4	service-satisfaction	E	3	...
5	travel-feedback	D	5	...
6	travel-feedback	C	4	...
7	purchase-experience	C	2	...
8	brand-rating	B	3	...
9	book-review	A	3	...
10	travel-feedback	E	4	...
11	book-review	D	3	...

The first stage of MapReduce is to map the data into a format that can be progressively reduced until you obtain the required results. Both the map and reduce phases can be parallelized, which is why MapReduce can improve the performance for calculations over large datasets.

For this example, both the map and reduce phases will divide their input into blocks of three. The map phase in this example uses four parallel tasks, each one processes three survey result BLOBs, to build the map shown in the following table.

AggregationID	SurveySlug	Number of responses	Histogram of first multiple choice question	Average of first range question
1.1	travel-feedback	1	C	3
1.2	book-review	1	D	3
1.3	product-wishlist	1	A	4
2.1	service-satisfaction	1	E	3
2.2	travel-feedback	2	CD	4.5
3.1	purchase-experience	1	C	2
3.2	brand-rating	1	B	3
3.2	book-review	1	A	3
4.1	travel-feedback	1	E	4
4.2	book-review	1	D	3

The next phase reduces this data further. In this example, there will be two parallel tasks, one that processes aggregations 1.X, 2.X, and 3.X, and one that processes aggregation 4.X. It's important to realize that each reduce phase only needs to reference the data from the previous phase and not the original data. The following table shows the results of this reduce phase.

AggregationID	SurveySlug	Number of responses	Histogram of first multiple choice question	Average of first range question
1.1	travel-feedback	3	CCD	4
1.2	book-review	2	AD	3
1.3	product-wishlist	1	AD	4
1.4	service-satisfaction	1	E	3
1.5	purchase-experience	1	C	2
1.6	brand-rating	1	B	3
2.1	travel-feedback	1	E	4
2.2	book-review	1	D	3

In the next phase, there is only one task because there are only two input blocks. The following table shows the results from this reduction phase.

AggregationID	SurveySlug	Number of responses	Histogram of first multiple choice question	Average of first range question
1.1	travel-feedback	4	CCDE	4
1.2	book-review	3	ADD	3
1.3	product-wishlist	1	AD	4
1.4	service-satisfaction	1	E	3
1.5	purchase-experience	1	C	2
1.6	brand-rating	1	B	3

At this point, it's not possible to reduce the data any further, and the summary statistics for all the survey data that the application read during the original map phase have been calculated.

It's now possible to update the summary data based on survey responses received after the initial map phase ran. You process all new survey data using MapReduce and then combine the results from the new data with the old in the final step.

Scaling the Surveys Application

This section describes how Tailspin designed one functional area of the Surveys application for scalability. Tailspin anticipates that some surveys may have thousands, or even hundreds of thousands of respondents, and Tailspin wants to make sure that the public website remains responsive for all users at all times. At the same time, survey owners want to be able to view summary statistics calculated from the survey responses to date.

GOALS AND REQUIREMENTS

In Chapter 3, “Accessing the Surveys Application,” you saw how Tailspin uses two websites for the Surveys application: one where subscribers design and administer their surveys, and one where users fill out their survey responses. The Surveys application currently supports three question types: free text, numeric range (values from one to five), and multiple choice. Survey owners must be able to view some basic summary statistics that the application calculates for each survey, such as the total number of responses received, histograms of the multiple-choice results, and aggregations such as averages of the range results. The Surveys application provides a pre-determined set of summary statistics that cannot be customized by subscribers. Subscribers who want to perform a more sophisticated analysis of their survey responses can export the survey data to a SQL Azure instance.

Calculating summary statistics is an expensive operation if there are a large number of responses to process.

Because of the expected volume of survey response data, Tailspin anticipates that generating the summary statistics will be an expensive operation because of the large number of storage transactions that must occur when the application reads the survey responses. However, Tailspin does not require the summary statistics to be always up to date and based on all of the available survey responses. Tailspin is willing to accept a delay while the application calculates the summary data if this reduces the cost of generating them.

The public site where respondents fill out surveys must always have fast response times when users save their responses, and it must record the responses accurately so that there is no risk of any errors in the data when a subscriber comes to analyze the results.

The developers at Tailspin also want to be able to run comprehensive unit tests on the components that calculate the summary statistics without any dependencies on Windows Azure storage.



There are also integration tests that verify the end-to-end behavior of the application using Windows Azure storage.

THE SOLUTION

To meet the requirements, the developers at Tailspin decided to use a worker role to handle the task of generating the summary statistics from the survey results. Using a worker role enables the application to perform this resource-intensive process as a background task, ensuring that the web role responsible for collecting survey answers is not blocked while the application calculates the summary statistics.

Based on the framework for worker roles that the previous section outlined, this asynchronous task is one that will be triggered on a schedule, and it must be run as a single instance process because it updates a single set of results.

The application can use additional tasks in the same worker role to perform any additional processing on the response data; for example, it can generate a list of ordered answers to enable paging through the response data.

To calculate the survey statistics, Tailspin considered two basic approaches. The first approach is for the task in the worker role to retrieve all the survey responses to date at a fixed time interval, recalculate the summary statistics, and then save the summary data over the top of the existing summary data. The second approach is for the task in the worker role to retrieve all the survey response data that the application has saved since the last time the task ran, and use this data to adjust the summary statistics to reflect the new survey results.

The first approach is the simplest to implement, because the second approach requires a mechanism for tracking which survey results are new. The second approach also depends on it being possible to calculate the new summary data from the old summary data and the new survey results without re-reading all the original survey results.

Note: *You can recalculate all the summary data in the Surveys application using the second approach. However, suppose you want one of your pieces of summary data to be a list of the 10 most popular words used in answering a free-text question. In this case, you would always have to process all of the survey answers, unless you also maintained a separate list of all the words used and a count of how often they appeared. This adds to the complexity of the second approach.*

The key difference between the two approaches is cost. The graph in Figure 4 shows the result of an analysis that compares the costs of the two approaches for three different daily volumes of survey answers. The graph shows the first approach on the upper line with the Recalculate label, and the second approach on the lower line with the Merge label.



You can use a queue to maintain a list of all new survey responses. This task is still triggered on a schedule that determines how often the task should look at the queue for new survey results to process.

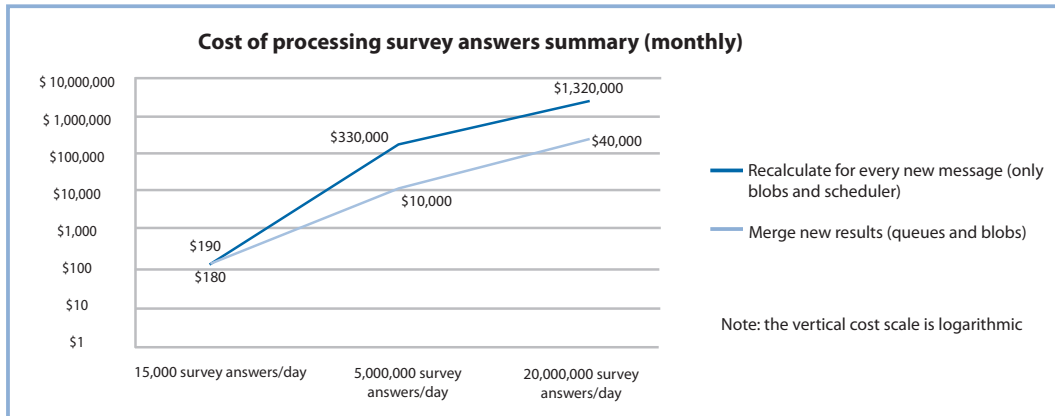


FIGURE 4
Comparison of costs for alternative approaches to calculating summary statistics

The graph clearly shows how much cheaper the merge approach is than the recalculate approach after you get past a certain volume of transactions. The difference in cost is due almost entirely to the transaction costs associated with the two different approaches. Tailspin decided to implement the merge approach in the Surveys application.

Note: *The vertical cost scale on the chart is logarithmic. The analysis behind this chart makes a number of “worst-case” assumptions about the way the application processes the survey results. The chart is intended to illustrate the relative difference in cost between the two approaches; it is not intended to give “hard” figures.*

It is possible to optimize the recalculate approach if you decide to sample the survey answers instead of processing every single one when you calculate the summary data. You would need to perform some detailed statistical analysis to determine what proportion of results you need to select to calculate the summary statistics within an acceptable margin of error.

In the Surveys application, it would also be possible to generate the summary statistics by using an approach based on MapReduce. The advantage of this approach is that it is possible to use multiple task instances to calculate the summary statistics. However, Tailspin is willing to accept a delay in calculating the summary statistics, so performance is not critical for this task. For a description of the MapReduce programming model, see the section, “MapReduce,” earlier in this chapter.

INSIDE THE IMPLEMENTATION

Now is a good time to walk through the code that implements the asynchronous task that calculates the summary statistics in more detail. As you go through this section, you may want to download the Visual Studio solution for the Tailspin Surveys application from <http://wag.codeplex.com/>.

Using a Worker Role to Calculate the Summary Statistics

The team at Tailspin decided to implement the asynchronous background task that calculates the summary statistics from the survey results by using a merge approach. Each time the task runs, it processes the survey responses that the application has received since the last time the task ran; it calculates the new summary statistics by merging the new results with the old statistics.

The worker role in the TailSpin.Workers.Surveys project periodically scans the queue for pending survey answers to process.

The following code example from the **UpdatingSurveyResultsSummaryCommand** class shows how the worker role processes each temporary survey answer and then uses them to recalculate the summary statistics.

```
private readonly IDictionary<string, SurveyAnswersSummary>
    surveyAnswersSummaryCache;
private readonly ISurveyAnswerStore surveyAnswerStore;
private readonly ISurveyAnswersSummaryStore
    surveyAnswersSummaryStore;

public UpdatingSurveyResultsSummaryCommand(
    IDictionary<string, SurveyAnswersSummary>
    surveyAnswersSummaryCache,
    ISurveyAnswerStore surveyAnswerStore,
    ISurveyAnswersSummaryStore surveyAnswersSummaryStore)
{
    this.surveyAnswersSummaryCache =
        surveyAnswersSummaryCache;
    this.surveyAnswerStore = surveyAnswerStore;
    this.surveyAnswersSummaryStore =
        surveyAnswersSummaryStore;
}

public void PreRun()
{
    this.surveyAnswersSummaryCache.Clear();
}

public void Run(SurveyAnswerStoredMessage message)
```

```
{
    this.surveyAnswerStore.AppendSurveyAnswerIdToAnswersList(
        message.Tenant,
        message.SurveySlugName,
        message.SurveyAnswerBlobId);

    var surveyAnswer =
        this.surveyAnswerStore.GetSurveyAnswer(
            message.Tenant,
            message.SurveySlugName,
            message.SurveyAnswerBlobId);

    var keyInCache = string.Format(CultureInfo.InvariantCulture,
        "{0}_{1}", message.Tenant, message.SurveySlugName);
    SurveyAnswersSummary surveyAnswersSummary;

    if (!this.surveyAnswersSummaryCache.ContainsKey(keyInCache))
    {
        surveyAnswersSummary = new
            SurveyAnswersSummary(message.Tenant,
            message.SurveySlugName);
        this.surveyAnswersSummaryCache[keyInCache] =
            surveyAnswersSummary;
    }
    else
    {
        surveyAnswersSummary =
            this.surveyAnswersSummaryCache[keyInCache];
    }

    surveyAnswersSummary.AddNewAnswer(surveyAnswer);
}

public void PostRun()
{
    foreach (var surveyAnswersSummary in
        this.surveyAnswersSummaryCache.Values)
    {
        var surveyAnswersSummaryInStore =
            this.surveyAnswersSummaryStore
                .GetSurveyAnswersSummary(surveyAnswersSummary.Tenant,
                surveyAnswersSummary.SlugName);
```

```
        surveyAnswersSummary.MergeWith(
            surveyAnswersSummaryInStore);

        this.surveyAnswersSummaryStore
            .SaveSurveyAnswersSummary(surveyAnswersSummary);
    }
}
```

The Surveys application uses the Unity Application Block (Unity) to initialize an instance of the **UpdatingSurveyResultsSummary Command** class and the **surveyAnswerStore** and **surveyAnswersSummaryStore** variables. The **surveyAnswerStore** variable is an instance of the **SurveyAnswerStore** type that the **Run** method uses to read the survey responses from BLOB storage. The **surveyAnswersSummaryStore** variable is an instance of the **SurveyAnswersSummary** type that the **PostRun** method uses to write summary data to BLOB storage. The **surveyAnswersSummaryCache** dictionary holds a **SurveyAnswersSummary** object for each survey.

***Note:** Unity is a lightweight, extensible dependency injection container that supports interception, constructor injection, property injection, and method call injection. You can use Unity in a variety of ways to help decouple the components of your applications, to maximize coherence in components, and to simplify design, implementation, testing, and administration of these applications.*

For more information about Unity and to download the application block, see the patterns & practices Unity page on CodePlex (<http://unity.codeplex.com/>).

The **PreRun** method runs before the task reads any messages from the queue and initializes a temporary cache for the new survey response data.

The **Run** method runs once for each new survey response. It uses the message from the queue to locate the new survey response, and then it adds the survey response to the **SurveyAnswersSummary** object for the appropriate survey by calling the **AddNewAnswer** method. The **AddNewAnswer** method updates the summary statistics in the **surveyAnswersSummaryStore** instance. The **Run** method also calls the **AppendSurveyAnswerIdToAnswersList** method to update the list of survey responses that the application uses for paging.

The **PostRun** method runs after the task reads all the outstanding answers in the queue. For each survey, it merges the new results with the existing summary statistics, and then it saves the new values back to BLOB storage.

The worker role uses some “plumbing” code developed by Tailspin to invoke the **PreRun**, **Run**, and **PostRun** methods in the **UpdatingSurveyResultsSummaryCommand** class on a schedule. The following code example shows how the Surveys application uses the “plumbing” code in the **Run** method in the worker role to run the three methods that comprise the job.

```
public override void Run()
{
    var updatingSurveyResultsSummaryJob =
        this.container.Resolve
            <UpdatingSurveyResultsSummaryCommand>();
    var surveyAnswerStoredQueue =
        this.container.Resolve
            <IAzureQueue<SurveyAnswerStoredMessage>>();
    BatchProcessingQueueHandler
        .For(surveyAnswerStoredQueue)
        .Every(TimeSpan.FromSeconds(10))
        .Do(updatingSurveyResultsSummaryJob);

    var transferQueue = this.container
        .Resolve<IAzureQueue<SurveyTransferMessage>>();
    var transferCommand = this
        .container.Resolve<TransferSurveysToSqlAzureCommand>();
    QueueHandler
        .For(transferQueue)
        .Every(TimeSpan.FromSeconds(5))
        .Do(transferCommand);

    while (true)
    {
        Thread.Sleep(TimeSpan.FromSeconds(5));
    }
}
```

This method first uses Unity to instantiate the **UpdatingSurveyResultsSummaryCommand** object that defines the job and the **AzureQueue** object that holds notifications of new survey responses.

The method then passes these objects as parameters to the **For** and **Do** “plumbing” methods. The **Every** “plumbing” method specifies how frequently the job should run. These methods cause the plumbing code to invoke the **PreRun**, **Run**, and **PostRun** method in the **UpdatingSurveyResultsSummaryCommand** class, passing a message from the queue to the **Run** method.

The preceding code example also shows how the worker role initializes the task defined in the **TransferSurveysToSqlAzureCommand** class that dumps survey data to SQL Azure. This task is slightly simpler and only has a **Run** method.

You should tune the frequency at which these tasks run based on your expected workloads by changing the value passed to the **Every** method.

Finally, the method uses a **while** loop to keep the worker role instance alive.

Note: *The **For**, **Every**, and **Do** methods implement a fluent API for instantiating tasks in the worker role. Fluent APIs help to make the code more legible.*

The Worker Role “Plumbing” Code

The “plumbing” code in the worker role enables you to invoke commands of type **IBatchCommand** or **ICommand** by using the **Do** method, on a Windows Azure queue of type **IAzureQueue** by using the **For** method, at a specified interval. Figure 5 shows the key types that make up the “plumbing” code.

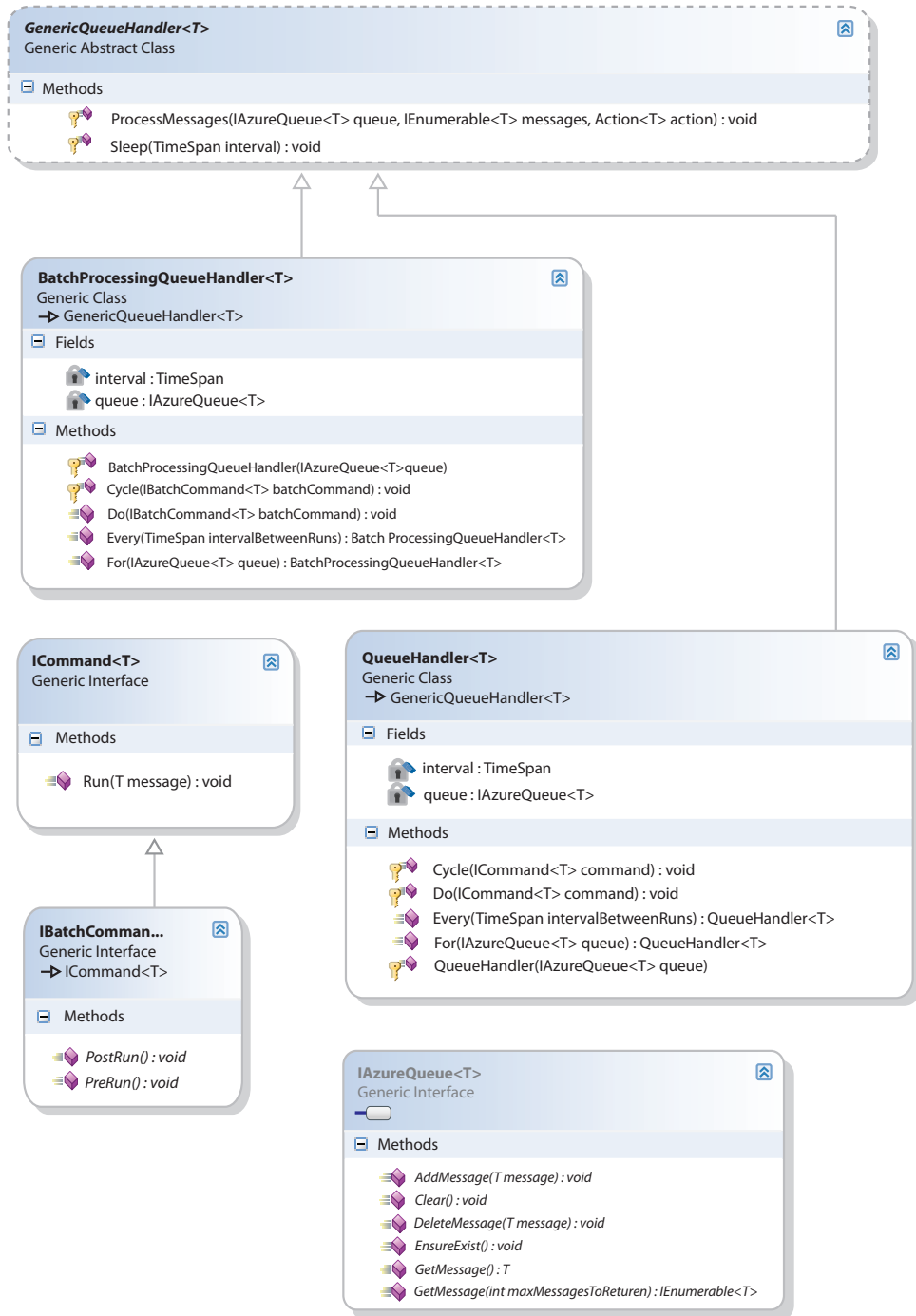


FIGURE 5
Key “plumbing” types

Figure 5 shows both a **BatchProcessingQueueHandler** class and a **QueueHandler** class. The **QueueHandler** class runs tasks that implement the simpler **ICommand** interface instead of the **IBatchCommand** interface. The discussion that follows focuses on the **BatchProcessingQueueHandlerTask** that the application uses to create the summary statistics.

The worker role first invokes the **For** method in the static **BatchProcessingQueueHandler** class, which invokes the **For** method in the **BatchProcessingQueueHandler<T>** class to return a **BatchProcessingQueueHandler<T>** instance that contains a reference to the **IAzureQueue<T>** instance to monitor. The “plumbing” code identifies the queue based on a queue message type that derives from the **AzureQueueMessage** type. The following code example shows how the **For** method in the **BatchProcessingQueueHandler<T>** class instantiates a **BatchProcessingQueueHandler<T>** instance.

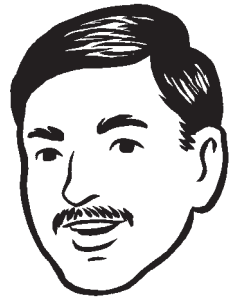
```
private readonly IAzureQueue<T> queue;
private TimeSpan interval;

protected BatchProcessingQueueHandler(IAzureQueue<T> queue)
{
    this.queue = queue;
    this.interval = TimeSpan.FromMilliseconds(200);
}

public static BatchProcessingQueueHandler<T> For(
    IAzureQueue<T> queue)
{
    if (queue == null)
    {
        throw new ArgumentNullException("queue");
    }

    return new BatchProcessingQueueHandler<T>(queue);
}
```

Next, the worker role invokes the **Every** method of the **BatchProcessingQueueHandler<T>** object to specify how frequently the task should be run.



The current implementation uses a single queue, but you could modify the **BatchProcessingQueueHandler** to read from multiple queues instead of only one. According to the benchmarks published at <http://azurescope.clouppapp.net>, the maximum write throughput for a queue is between 500 and 700 items per second. If the Surveys application needs to handle more than approximately 2 million survey responses per hour, the application will hit the threshold for writing to a single queue. You could change the application to use multiple queues, perhaps with different queues for each subscriber.

*Use **Task.Factory.StartNew** in preference to **ThreadPool.QueueUserWorkItem**.*

Next, the worker role invokes the **Do** method of the **BatchProcessingQueueHandler<T>** object, passing an **IBatchCommand** object that identifies the command that the “plumbing” code should execute on each message in the queue. The following code example shows how the **Do** method uses the **Task.Factory.StartNew** method from the Task Parallel Library (TPL) to run the **PreRun**, **ProcessMessages**, and **PostRun** methods on the queue at the requested interval.

```
public virtual void Do(IBatchCommand<T> batchCommand)
{
    Task.Factory.StartNew(() =>
        {
            while (true)
            {
                this.Cycle(batchCommand);
            }
        }, TaskCreationOptions.LongRunning);
}

protected void Cycle(IBatchCommand<T> batchCommand)
{
    try
    {
        batchCommand.PreRun();

        bool continueProcessing;
        do
        {
            var messages = this.queue.GetMessages(32);
            ProcessMessages(this.queue, messages,
                batchCommand.Run());

            continueProcessing = messages.Count() > 0;
        }
        while (continueProcessing);

        batchCommand.PostRun();

        this.Sleep(this.interval);
    }
    catch (TimeoutException)
    {
    }
}
```

The **Cycle** method repeatedly pulls up to 32 messages from the queue in a single transaction for processing until there are no more messages left.

The following code example shows the **ProcessMessages** method in the **GenericQueueHandler** class.

```
protected static void ProcessMessages(IAzureQueue<T> queue,
    IEnumerable<T> messages, Action<T> action)
{
    ...

    foreach (var message in messages)
    {
        var success = false;

        try
        {
            action(message);
            success = true;
        }
        catch (Exception)
        {
            success = false;
        }
        finally
        {
            if (success || message.DequeueCount > 5)
            {
                queue.DeleteMessage(message);
            }
        }
    }
}
```

This method uses the *action* parameter to invoke the custom command on each message in the queue. Finally, the method checks for poison messages by looking at the **DequeueCount** property of the message; if the application has tried more than five times to process the message, the method deletes the message.

Note: *Instead of deleting poison messages, you should send them to a dead message queue for analysis and troubleshooting.*

Testing the Worker Role

The implementation of the “plumbing” code in the worker role, and the use of Unity, makes it possible to run unit tests on the worker role components using mock objects instead of Windows Azure queues and BLOBs. The following code from the **BatchProcessingQueueHandlerFixture** class shows two example unit tests.

```
[TestMethod]
public void ForCreatesHandlerForGivenQueue()
{
    var mockQueue = new Mock<IAzureQueue<StubMessage>>();

    var queueHandler = BatchProcessingQueueHandler
        .For(mockQueue.Object);

    Assert.IsInstanceOfType(queueHandler,
        typeof(BatchProcessingQueueHandler<StubMessage>));
}

[TestMethod]
public void DoRunsGivenCommandForEachMessage()
{
    var message1 = new StubMessage();
    var message2 = new StubMessage();
    var mockQueue = new Mock<IAzureQueue<StubMessage>>();
    mockQueue.Setup(q =>
        q.GetMessages(32)).Returns(
        () => new[] { message1, message2 });
    var command = new Mock<IBatchCommand<StubMessage>>();
    var queueHandler =
        new BatchProcessingQueueHandlerStub(mockQueue.Object);

    queueHandler.Do(command.Object);

    command.Verify(c => c.Run(It.IsAny<StubMessage>()),
        Times.Exactly(2));
    command.Verify(c => c.Run(message1));
    command.Verify(c => c.Run(message2));
}
```

```
public class StubMessage : AzureQueueMessage
{
}

private class BatchProcessingQueueHandlerStub :
    BatchProcessingQueueHandler<StubMessage>
{
    public BatchProcessingQueueHandlerStub(
        IAzureQueue<StubMessage> queue) : base(queue)
    {
    }

    public override void Do(
        IBatchCommand<StubMessage> batchCommand)
    {
        this.Cycle(batchCommand);
    }
}
```

The **ForCreateHandlerForGivenQueue** unit test verifies that the static **For** method instantiates a **BatchProcessingQueueHandler** correctly by using a mock queue. The **DoRunsGivenCommandForEachMessage** unit test verifies that the **Do** method causes the command to be executed against every message in the queue by using mock queue and command objects.

References and Resources

For more information about ASP.NET routing, see “ASP.NET Routing” on MSDN:

<http://msdn.microsoft.com/en-us/library/cc668201.aspx>

For more information about the URL Rewrite Module for IIS, see “URL Rewrite” on IIS.net:

<http://www.iis.net/download/urlrewrite>

For more information about fluent APIs, see the entry for “Fluent interface” on Wikipedia:

http://en.wikipedia.org/wiki/Fluent_interface

For more information about the MapReduce algorithm, see the following:

- The entry for “MapReduce” on Wikipedia:
<http://en.wikipedia.org/wiki/MapReduce>
- The article, “Google patents Map/Reduce,” on The H website:
<http://www.h-online.com/open/news/item/Google-patents-Map-Reduce-908602.html>

For information about the Task Parallel Library, see “Task Parallel Library” on MSDN:

<http://msdn.microsoft.com/en-us/library/dd460717.aspx>

For information about the advantages of using the Task Parallel library instead of working with the thread pool directly, see the following:

- The article, “Optimize Managed Code for Multi-Core Machines,” in *MSDN Magazine*:
<http://msdn.microsoft.com/en-us/magazine/cc163340.aspx>
- The blog post, “Choosing Between the Task Parallel Library and the ThreadPool,” on the Parallel Programming with .NET blog: <http://blogs.msdn.com/b/pfxteam/archive/2009/10/06/9903475.aspx>

5

Working with Data in the Surveys Application

This chapter describes how the Surveys application uses data. It first describes the data model used by the Surveys application and then discusses why the team at Tailspin chose this particular approach with reference to a number of specific scenarios in the application. It also describes how the developers ensured the testability of their solution. Finally, it describes how and why the application uses SQL Azure™.

A Data Model for a Multi-Tenant Application

This section describes the data model in the surveys application and explains how the table design partitions the data by tenant.

The Surveys application uses a mix of table storage and BLOB storage to store its data. The sections, “Saving Survey Response Data” and “Paging through Survey Results,” later in this chapter discuss why the application uses BLOB storage for some data. Figure 1 shows, at a high level, which data is stored in the different storage types.

The Surveys application uses BLOB and table storage.

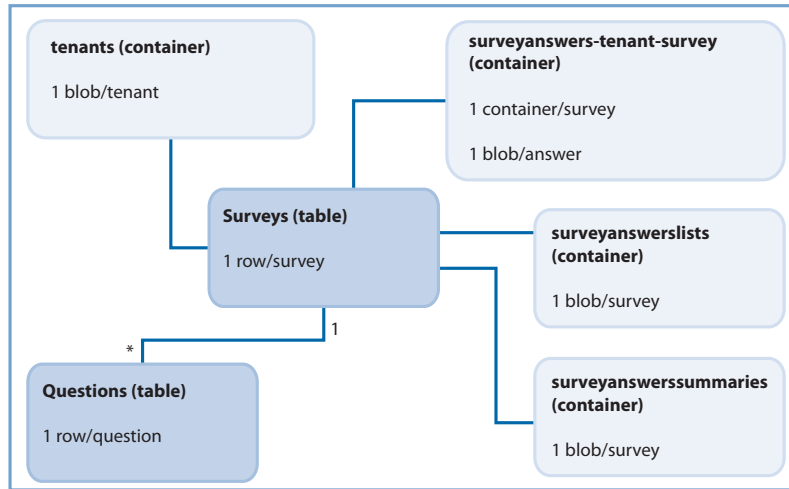


FIGURE 1
Data storage in the Surveys application

STORING SURVEY DEFINITIONS

The Surveys application stores the definition of surveys in two Windows Azure™ tables. This section describes these tables and explains why Tailspin adopted this design.

The following table describes the fields in the Surveys table. This table holds a list of all of the surveys in the application.

Field name	Notes
PartitionKey	This field contains the tenant name. Tailspin chose this value because they want to be able to filter quickly by tenant name.
RowKey	This field contains the tenant name from the PartitionKey field concatenated with the “slugified” version of the survey name. This makes sure that a subscriber cannot create two surveys with the same name. Two subscribers could choose the same name for their surveys.
Timestamp	Windows Azure table storage automatically maintains the value in this field.
SlugName	The “slugified” version of the survey name.
CreatedOn	This field records when the subscriber created the survey. This will differ from the Timestamp value if the subscriber edits the survey.
Title	The survey name.



A slug name is a string where all whitespace and invalid characters are replaced with a hyphen (-). The term comes from the newsprint industry and has nothing to do with those things in your garden!

The following table describes the fields in the Questions table. The application uses this table to store the question definitions and to render a survey.

Field name	Notes
PartitionKey	This field contains the tenant name from the PartitionKey field in the Surveys table concatenated with the “slugified” version of the survey name. This enables the application to insert all questions for a survey in a single transaction and to retrieve all the questions in a survey quickly from a single partition.
RowKey	This field contains a formatted tick count concatenated with position of the question within the survey. This guarantees a unique RowKey value and defines the ordering of the questions.
Timestamp	Windows Azure table storage automatically maintains the value in this field.
Text	The question text.
Type	The question type: Simple text, multiple choice, or five stars (a numeric range).
PossibleAnswers	This field contains a list of the possible answers if the question is a multiple-choice question.

To read a more detailed discussion of **RowKeys** and **PartitionKeys** in Windows Azure table storage, see Chapter 5, “Phase 2: Automating Deployment and Using Windows Azure Storage,” of the book, *Windows Azure Architecture Guide, Part 1: Moving Applications to the Cloud*, available at <http://msdn.microsoft.com/en-us/library/ff728592.aspx>.

STORING TENANT DATA

The Surveys application saves tenant data in BLOB storage in a container named tenants. The following code shows the **Tenant** class; the application serializes **Tenant** instances to BLOBs identified by the subscriber’s name (the term “tenant” is used in the storage classes to refer to subscribers).

```
[Serializable]
public class Tenant
{
    public string ClaimType { get; set; }

    public string ClaimValue { get; set; }
```



Remember that Windows Azure table storage only supports transactions within a single partition on a single table.


```
public string HostGeolocation { get; set; }

public string IssuerThumbPrint { get; set; }

public string IssuerUrl { get; set; }

public string Logo { get; set; }

public string Name { get; set; }

public string SqlAzureConnectionString { get; set; }

public string DatabaseName { get; set; }

public string DatabaseUserName { get; set; }

public string DatabasePassword { get; set; }

public string SqlAzureFirewallIpStart { get; set; }

public string SqlAzureFirewallIpEnd { get; set; }
}
```

The application collects most of the subscriber data during the onboarding process. The **Logo** property contains the URL for the subscriber's logo. The application stores logo images in a public BLOB container named logos.

STORING SURVEY ANSWERS

The Surveys application saves survey answers in BLOB storage. The application creates a BLOB container for each survey with a name that follows this pattern: `surveyanswers-<tenant name>-<survey slug name>`. This guarantees a unique container name for every survey.

For each completed survey response, the Surveys application saves a BLOB into the survey's container. The BLOB name is a tick count derived from the current date and time, which ensures that each BLOB in the container has a unique name. The content of each BLOB is a **SurveyAnswer** object serialized in the JavaScript Object Notation (JSON) format. The following code example shows the **SurveyAnswer** and **QuestionAnswer** classes.

```
public class SurveyAnswer
{
    ...

    public string SlugName { get; set; }
}
```

```

public string Tenant { get; set; }
public string Title { get; set; }
public DateTime CreatedOn { get; set; }
public List<QuestionAnswer> QuestionAnswers { get; set; }
}

public class QuestionAnswer
{
    public string QuestionText { get; set; }
    public QuestionType QuestionType { get; set; }

    [Required(ErrorMessage = "* You must provide an answer.")]
    public string Answer { get; set; }
    public string PossibleAnswers { get; set; }
}

```

The Surveys application also uses BLOB storage to store an ordered list of the responses to each survey. For each survey, the application stores a BLOB that contains a serialized **List** object containing the ordered names of all the survey response BLOBs for that survey. The **List** object is serialized in the JSON format. The section, “Paging Through Survey Results,” later in this chapter explains how the Surveys application uses these **List** objects to enable paging through the survey results.

STORING SURVEY ANSWER SUMMARIES

The Surveys application uses BLOB storage to save the summary statistical data for each survey. For each survey, it creates a BLOB named `<tenant-name>-<survey slug name>` in the `surveyanswerssummaries` container. The application serializes a **SurveyAnswersSummary** object in the JSON format to save the data. The following code example shows the **SurveyAnswersSummary** and **QuestionAnswersSummary** classes that define the summary data.

```

public class SurveyAnswersSummary
{
    ...
    public string Tenant { get; set; }

    public string SlugName { get; set; }

    public int TotalAnswers { get; set; }

    public List<QuestionAnswersSummary> QuestionAnswersSummaries
        { get; set; }
    ...
}

```

```
}  
  
public class QuestionAnswersSummary  
{  
    public string AnswersSummary { get; set; }  
  
    public QuestionType QuestionType { get; set; }  
  
    public string QuestionText { get; set; }  
  
    public string PossibleAnswers { get; set; }  
}
```

Notice that the summary is stored as a string for all question types, including numeric. This helps to minimize the number of changes that would be required to add a new question type to the Surveys application.

THE STORE CLASSES

The Surveys application uses store classes to manage storage. This section briefly outlines the responsibilities of each of these store classes.

SurveyStore Class

This class is responsible for saving survey definitions to table storage and retrieving the definitions from table storage.

SurveyAnswerStore Class

This class is responsible for saving survey answers to BLOB storage and retrieving survey answers from BLOB storage. This class creates a new container when it saves the first response to a new survey. It uses a queue to track new survey responses; the application uses this queue to calculate the summary statistical data for surveys.

This class also provides support for browsing sequentially through survey responses.

SurveyAnswersSummaryStore Class

This class is responsible for saving summary statistical data for surveys to BLOBs in the surveyanswerssummaries container, and for retrieving this data.

SurveySqlStore Class

This class is responsible for saving survey response data to SQL Azure. For more information, see the section, “Using SQL Azure,” later in this chapter.

SurveyTransferStore Class

This class is responsible placing a message on a queue when a subscriber requests the application to dump survey data to SQL Azure.

TenantStore Class

This class is responsible for saving and retrieving subscriber data and saving uploaded logo images. In the sample code, this class generates some default data for the Adatum and Fabrikam subscribers.

Testing and Windows Azure Storage

This section describes how the design and implementation of the data store classes in the Surveys application make unit testing and updating the storage mechanism easier.

GOALS AND REQUIREMENTS

The Surveys application uses Windows Azure table and BLOB storage, and the developers at Tailspin were concerned about how this would affect their unit testing strategy. From a testing perspective, a unit test should focus on the behavior of a specific class and not on the interaction of that class with other components in the application. From the perspective of Windows Azure, any test that depends on Windows Azure storage requires complex setup and tear-down logic to make sure that the correct data is available for the test to run. For both of these reasons, the developers at Tailspin designed the data access functionality in the Surveys application with testability in mind, and specifically to make it possible to run unit tests on their data store classes without a dependency on Windows Azure storage.

THE SOLUTION

The solution adopted by the developers at Tailspin was to wrap the Windows Azure storage components in such a way as to facilitate replacing them with mock objects during unit tests and to use the Unity Application Block (Unity). A unit test should be able to instantiate a suitable mock storage component, use it for the duration of the test, and then discard it. Any integration tests can continue to use the original data access components to test the functionality of the application.

The Surveys application uses Unity to decouple its components and facilitate testing.

Note: *Unity is a lightweight, extensible dependency injection container that supports interception, constructor injection, property injection, and method call injection. You can use Unity in a variety of ways to help decouple the components of your applications, to maximize coherence in components, and to simplify design, implementation, testing, and administration of these applications.*

You can learn more about Unity and download the application block at <http://unity.codeplex.com/>.

INSIDE THE IMPLEMENTATION

Now is a good time to walk through some code that illustrates testing the store classes in more detail. As you go through this section, you may want to download the Microsoft® Visual Studio® development system solution for the Tailspin Surveys application from <http://wag.codeplex.com/>.

This section describes how the design of the Surveys application supports unit testing of the **SurveyStore** class that provides access to the table storage. This description focuses on one specific set of tests, but the application uses the same approach with other store classes.

The following code example shows the **IAzureTable** interface and the **AzureTable** class that are at the heart of the implementation.

```
public interface IAzureTable<T> where T : TableServiceEntity
{
    IQueryable<T> Query { get; }
    void EnsureExist();
    void Add(T obj);
    void Add(IEnumerable<T> objs);
    void AddOrUpdate(T obj);
    void AddOrUpdate(IEnumerable<T> objs);
    void Delete(T obj);
    void Delete(IEnumerable<T> objs);
}

public class AzureTable<T>
    : IAzureTable<T> where T : TableServiceEntity
{
    private readonly string tableName;
    private readonly CloudStorageAccount account;

    ...

    public IQueryable<T> Query
    {
        get
```

```
{
    TableServiceContext context = this.CreateContext();
    return context.CreateQuery<T>(this.tableName)
        .AsTableServiceQuery();
}

public void Add(T obj)
{
    this.Add(new[] { obj });
}

public void Add(IEnumerable<T> objs)
{
    TableServiceContext context = this.CreateContext();

    foreach (var obj in objs)
    {
        context.AddObject(this.tableName, obj);
    }

    var saveChangesOptions = SaveChangesOptions.None;
    if (objs.Distinct(new PartitionKeyComparer())
        .Count() == 1)
    {
        saveChangesOptions = SaveChangesOptions.Batch;
    }

    context.SaveChanges(saveChangesOptions);
}

...

private TableServiceContext CreateContext()
{
    return new TableServiceContext(
        this.account.TableEndpoint.ToString(),
        this.account.Credentials);
}

private class PartitionKeyComparer :
    IEqualityComparer<TableServiceEntity>
{
    public bool Equals(TableServiceEntity x,
        TableServiceEntity y)
```

```

    {
        return string.Compare(x.PartitionKey, y.PartitionKey,
            true,
            System.Globalization.CultureInfo
                .InvariantCulture) == 0;
    }

    public int GetHashCode(TableServiceEntity obj)
    {
        return obj.PartitionKey.GetHashCode();
    }
}
}

```

Note: The **Add** method that takes an **IEnumerable** parameter should check the number of items in the batch and the size of the payload before calling the **SaveChanges** method with the **SaveChangesOptions.Batch** option. For more information about batches and Windows Azure table storage, see the section, “Transactions in aExpense,” in Chapter 5, “Phase 2: Automating Deployment and Using Windows Azure Storage,” of the book, *Windows Azure Architecture Guide, Part 1: Moving Applications to the Cloud*, available at <http://msdn.microsoft.com/en-us/library/ff728592.aspx>.

The generic interface and class have a type parameter **T** that derives from the Windows Azure **TableServiceEntity** type that you use to create your own table types. For example, in the Surveys application, the **SurveyRow** and **QuestionRow** types derive from the **TableServiceEntity** class. The interface defines several operations: the **Query** method returns an **IQueryable** collection of the type **T**, and the **Add**, **AddOrUpdate**, and **Delete** methods each take a parameter of type **T**. In the **AzureTable** class, the **Query** method returns a **TableServiceQuery** object, the **Add** and **AddOrUpdate** methods save the object to table storage, and the **Delete** method deletes the object from table storage. To create a mock object for unit testing, you must instantiate an object of type **IAzureTable**.

The following code example from the **SurveyStore** class shows the constructor.

```

public SurveyStore(IAzureTable<SurveyRow> surveyTable,
    IAzureTable<QuestionRow> questionTable)
{
    this.surveyTable = surveyTable;
    this.questionTable = questionTable;
}

```

The constructor takes parameters of type **IAzureTable** that enable you to pass in either real or mock objects that implement the interface.

This parameterized constructor is invoked in two different scenarios. The Surveys application invokes the constructor indirectly when the application uses the **SurveysController** MVC class. The application uses the Unity dependency injection framework to instantiate MVC controllers. The Surveys application replaces the standard MVC controller factory with the **UnityControllerFactory** class in the **OnStart** method in both web roles, so when the application requires a new MVC controller instance, Unity is responsible for instantiating that controller. The following code example shows part of the **ContainerBootstrapper** class from the TailSpin.Web project that the Unity container uses to determine how to instantiate objects.

```
public static class ContainerBootstrapper
{
    public static void RegisterTypes(IUnityContainer container)
    {
        var account = CloudConfiguration
            .GetStorageAccount("DataConnectionString");
        container.RegisterInstance(account);

        container.RegisterType<ISurveyStore, SurveyStore>();

        container.RegisterType<IAzureTable<SurveyRow>,
            AzureTable<SurveyRow>>(
            new InjectionConstructor(typeof(
                Microsoft.WindowsAzure.CloudStorageAccount),
                AzureConstants.Tables.Surveys));

        container.RegisterType<IAzureTable<QuestionRow>,
            AzureTable<QuestionRow>>(
            new InjectionConstructor(typeof(
                Microsoft.WindowsAzure.CloudStorageAccount),
                AzureConstants.Tables.Questions));

        ...
    }
}
```

The last two calls to the **RegisterType** method define the rules that tell the Unity container how to instantiate the **AzureTable** instances that it must pass to the **SurveyStore** constructor.

When the application requires a new MVC controller instance, Unity is responsible for creating the controller, and in the case of the **SurveysController** class, Unity instantiates a **SurveyStore** object using the parameterized constructor shown earlier, and passes the **SurveyStore** object to the **SurveysController** constructor.

In the second usage scenario for the parameterized **SurveyStore** constructor, you create unit tests for the **SurveyStore** class by directly invoking the constructor and passing in mock objects. The following code example shows a unit test method that uses the constructor in this way.

```
[TestMethod]
public void GetSurveyByTenantAndSlugNameReturnsTenantNameFrom
PartitionKey()
{
    string expectedRowKey = string.Format(
        CultureInfo.InvariantCulture, "{0}_{1}", "tenant",
        "slug-name");
    var surveyRow = new SurveyRow { RowKey = expectedRowKey,
        PartitionKey = "tenant" };
    var surveyRowsForTheQuery = new[] { surveyRow };
    var mock = new Mock<IAzureTable<SurveyRow>>();
    mock.SetupGet(t => t.Query)
        .Returns(surveyRowsForTheQuery.AsQueryable());
    var store = new SurveyStore(mock.Object,
        default(IAzureTable<QuestionRow>));

    var survey = store.GetSurveyByTenantAndSlugName("tenant",
        "slug-name", false);

    Assert.AreEqual("tenant", survey.Tenant);
}
```

The test creates a mock **IAzureTable<SurveyRow>** instance, uses it to instantiate a **SurveyStore** object, invokes the **GetSurveyByTenantAndSlugName** method, and checks the result. It performs this test without touching Windows Azure table storage.

The Surveys application uses a similar approach to enable unit testing of the other store components that use Windows Azure BLOB and table storage.

Saving Survey Response Data

When a user completes a survey, the application must save the user's answers to the survey questions to storage where the survey creator can access and analyze the results.

GOALS AND REQUIREMENTS

The format that application uses to save the summary response data must enable the Surveys application to meet the following three requirements:

- The owner of the survey must be able to browse the results.
- The application must be able to calculate summary statistics from the answers.
- The owner of the survey must be able to export the answers in a format that enables detailed analysis of the results.

Tailspin expects to see a very large number of users completing surveys; therefore, the process that initially saves the data should be as efficient as possible. The application can handle any processing of the data after it has been saved by using an asynchronous worker process. For information about the design of this background processing functionality in the Surveys application, see the section, "Scaling the Surveys Application," in Chapter 4, "Building a Scalable, Multi-Tenant Application for Windows Azure," earlier in this book.

The focus here is on the way the Surveys application stores the survey answers. Whatever type of storage the Surveys application uses, it must be able to support the three requirements listed earlier. Storage costs are also a significant factor in the choice of storage type because survey answers account for the majority of the application's storage requirements; both in terms of space used and by the number of storage transactions.

THE SOLUTION

To meet the requirements, the developers at Tailspin analyzed two possible storage solutions: a delayed write pattern using queues and table storage, and a solution that saves directly to BLOB storage. In both cases, the application first saves the survey responses to storage, and then it uses an asynchronous task in a worker role to calculate and save the summary statistics.



Transaction costs will be significant because calculating summary statistical data and exporting survey results will require the application to read survey responses from storage.

The Surveys application saves each survey response as a BLOB.

Solution 1: The Delayed Write Pattern

Figure 2 shows the delayed write pattern that the Surveys application could use to save the results of a filled out survey to Windows Azure table storage.

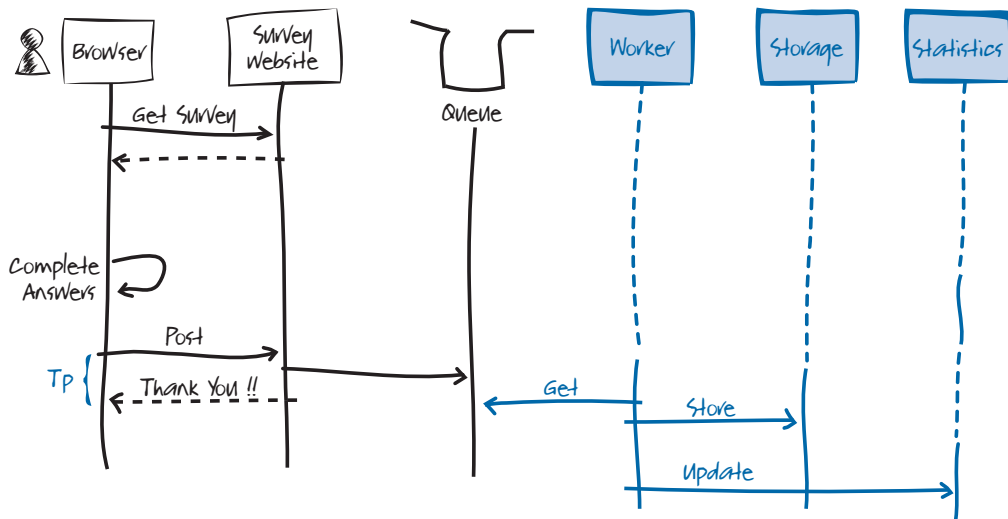
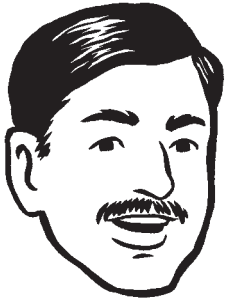


FIGURE 2

Delayed write pattern for saving survey responses in the Surveys application



Surveys is a “geo-aware” application. For example, the Surveys website and queue could be hosted in a data center in the U.S., and the worker role and table storage could be hosted in a data center in Europe.

In this scenario, a user browses to a survey, fills it out, and then submits his or her answers back to the Surveys website. The Surveys website puts the survey answers into a message on a queue and returns a “Thank you” message to the user as quickly as possible, minimizing the value of **Tp** in Figure 2. A task in a worker role is then responsible for reading the survey answers from the queue and saving them to table storage. This operation must be idempotent, to avoid any possibility of double counting and skewing the results.

Note: You could use separate worker roles, one to calculate and save the summary statistics, and one to save the survey results to table storage if you need to scale the application.

There is an 8-kilobyte (KB) maximum size for a message on a Windows Azure queue, so this approach works only if the size of each survey response is less than that maximum. Figure 3 shows how you could modify this solution to handle survey results that are greater than 8 KB in size.

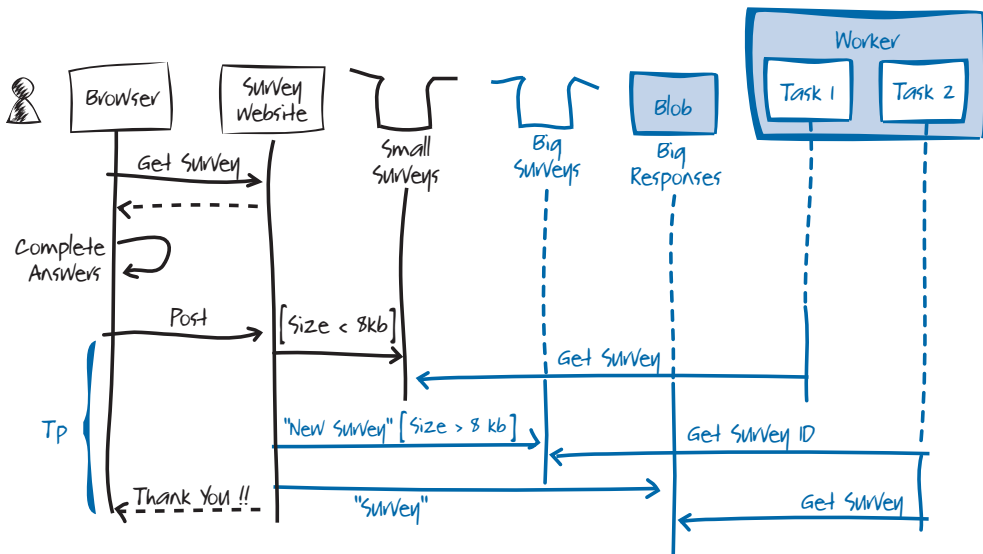


FIGURE 3
Handling survey results greater than 8 KB in size

Figure 3 includes an optimization, whereby the application places messages that are smaller than 8 KB directly onto a queue, as in the previous example. For messages that are larger than 8 KB in size, the application saves them to Windows Azure BLOB storage and places a message on the “Big Surveys” queue to notify the worker role. The worker role now contains two tasks: Task 1 retrieves and processes small surveys from the “Small Surveys” queue; Task 2 polls the “Big Surveys” queue for notifications of large surveys that it retrieves and processes from BLOB storage.

Solution 2: Writing Directly to BLOB Storage

As you saw in the previous section, the delayed write pattern becomes more complex if the size of a survey answer can be greater than 8 KB. In this case, it is necessary to save the response as a BLOB and notify the worker role of the new response data by using a message on a queue. The developers at Tailspin also analyzed a simpler approach to saving and processing query responses using only BLOB storage. Figure 4 illustrates this alternative approach.



When you calculate the size of messages, you must consider the effect of any encoding, such as Base64, that you use to encode the data before you place it in a message.

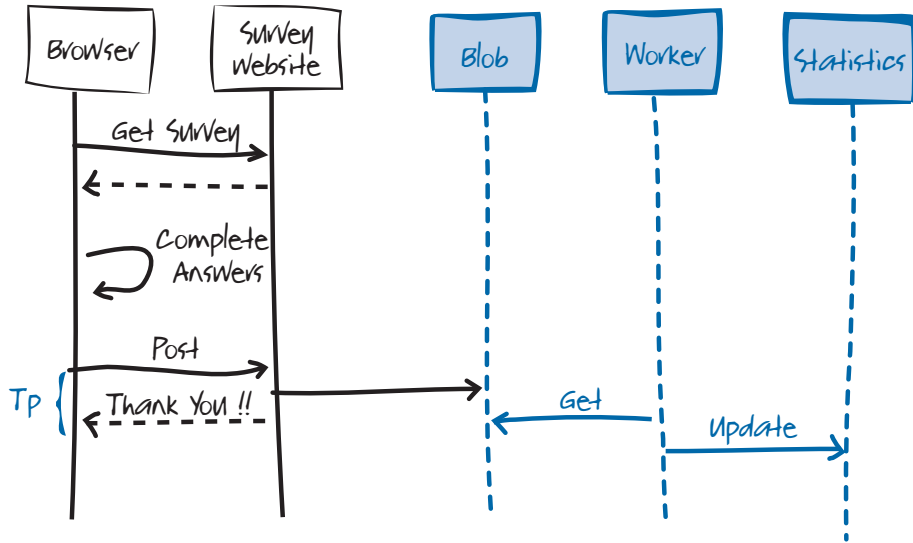


FIGURE 4
Saving survey responses directly to BLOB storage

As you can see from the sequence diagram in Figure 4, the first stages of the saving survey response process are the same as for the delayed write pattern. In this approach, there is no queue and no table storage, and the application stores the survey results directly in BLOB storage. The worker role now generates the summary statistical data directly from the survey responses in BLOB storage.

Figure 5 illustrates a variation on this scenario where the subscriber has chosen to host a survey in a different data center from his or her account.

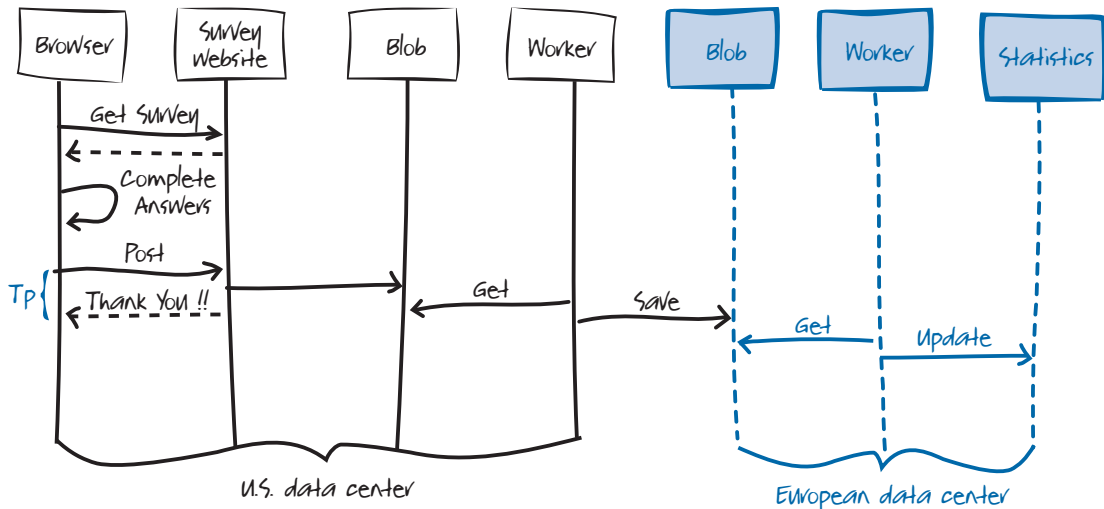


FIGURE 5
Saving the responses from a survey hosted in a different data center

In this scenario, there is an additional worker role. This worker role is responsible for moving the survey response data from the data center where the subscriber chose to host the survey to the data center hosting the subscriber's account. This way, the application transfers the survey data between data centers only once, instead of every time the application needs to read it; this minimizes the costs associated with this scenario.

Comparing the Solutions

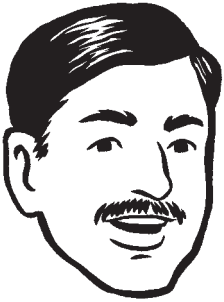
The second solution is much simpler than the first. However, you also need to check whether keeping the survey responses in BLOBs instead of tables adds complexity to any of the processes that use the survey results. In the Surveys application, using BLOBs does not add significantly to the complexity of generating summary statistics, enabling the survey owner to browse the responses, or exporting the data to SQL Azure.

Although the second solution does not limit the functionality that the Surveys application requires, this design may be limiting in other applications. Using the delayed write pattern means that you can easily perform operations on the data before it's saved to a table, so in scenarios where the raw data requires some processing to make it usable, the first solution may be more appropriate. Secondly, storing data in tables makes it much easier to access the data with dynamically constructed queries.



The application reads survey response data when it calculates the statistics, when a user browses through the responses, and when it exports the data to SQL Azure.

The delayed write pattern enables you to transform the data before saving it without affecting the performance of the web role.



You should also verify that the second solution does not add to the number of storage transactions that your application needs to perform when it processes or uses the saved data.

The third difference between the solutions is the storage costs. The following table summarizes this difference, showing the number of storage transactions that the application must perform in order to save a single survey response.

Solution 1 The delayed write pattern	Solution 2 Writing directly to BLOB storage
1 save to BLOB 1 add message to queue 1 get message from queue 1 read BLOB 1 save to table	1 save to BLOB
Total 5 storage transactions	Total 1 storage transactions

INSIDE THE IMPLEMENTATION

Now is a good time to walk through the code that saves the survey responses in more detail. As you go through this section, you may want to download the Visual Studio solution for the Tailspin Surveys application from <http://wag.codeplex.com/>.

Saving the Survey Response Data to a Temporary Blob

The following code from the **SurveysController** class in the TailSpin.Web.Survey.Public project shows how the application initiates saving the survey response asynchronously.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Display(string tenant, string surveySlug,
SurveyAnswer contentModel)
{
    var surveyAnswer = CallGetSurveyAndCreateSurveyAnswer(
        this.surveyStore, tenant, surveySlug);

    ...

    for (int i = 0; i < surveyAnswer.QuestionAnswers.Count; i++)
    {
        surveyAnswer.QuestionAnswers[i].Answer =
            contentModel.QuestionAnswers[i].Answer;
    }

    if (!this.ModelState.IsValid)
```

```
{
    var model =
        new TenantPageViewData<SurveyAnswer>(surveyAnswer);
    model.Title = surveyAnswer.Title;
    return this.View(model);
}

this.surveyAnswerStore.SaveSurveyAnswer(surveyAnswer);

return this.RedirectToAction("ThankYou");
}
```

The **surveyAnswerStore** variable holds a reference to an instance of the **SurveyAnswerStore** type. The application uses Unity to initialize this instance with the correct **IAzureBlob** and **IAzureQueue** instances. The BLOB container stores the answers to the survey questions, and the queue maintains a list of new survey answers that haven't been included in the summary statistics or the list of survey answers.

The **SaveSurveyAnswer** method writes the survey response data to the BLOB storage and puts a message onto the queue. The action method then immediately returns a "Thank you" message.

The following code example shows the **SaveSurveyAnswer** method in the **SurveyAnswerStore** class.

```
public void SaveSurveyAnswer(SurveyAnswer surveyAnswer)
{
    var surveyBlobContainer = this.surveyAnswerContainerFactory
        .Create(surveyAnswer.Tenant, surveyAnswer.SlugName);
    surveyBlobContainer.EnsureExist();
    DateTime now = DateTime.UtcNow;
    surveyAnswer.CreatedOn = now;
    var blobId = now.GetFormattedTicks();
    surveyBlobContainer.Save(blobId, surveyAnswer);
    this.surveyAnswerStoredQueue.AddMessage(
        new SurveyAnswerStoredMessage
        {
            SurveyAnswerBlobId = blobId,
            Tenant = surveyAnswer.Tenant,
            SurveySlugName = surveyAnswer.SlugName
        });
}
```



Make sure that the storage connection strings in your deployment point to storage in the deployment's geographical location. The application should use local queues and BLOB storage to minimize latency.



It's possible that the role could fail after it adds the survey data to BLOB storage but before it adds the message to the queue. In this case, the response data would not be included in the summary statistics or the list of responses used for paging. However, the response would be included if the user exported the survey to SQL Azure.

This method first checks that the BLOB container exists and creates it if necessary. It then creates a unique BLOB ID by using a tick count and saves the BLOB to the survey container. Finally, it adds a message to the queue. The application uses the queue to track new survey responses that must be included in the summary statistics and list of responses for paging through answers.

Note: *It is possible, but very unlikely, that the application could try to save two BLOBs with the same ID if two users completed a survey at exactly the same time. The code should check for this possibility and, if necessary, retry the save with a new tick count value.*

Displaying Data

This section describes several interesting scenarios in the Surveys application where the application displays data to users and how the underlying data model supports this functionality.

PAGING THROUGH SURVEY RESULTS

The owner of a survey must be able to browse through the survey results, displaying a single survey response at a time. This feature is in addition to being able to view summary statistical data, or being able to analyze the results using SQL Azure. The Surveys application contains a Browse Responses page for this function.

Goals and Requirements

The design of this feature of the application must address two specific requirements. The first requirement is that the application must display the survey responses in the order that they were originally submitted. The second requirement is to ensure that this feature does not adversely affect the performance of the web role.

The Solution

The developers at Tailspin considered two solutions, each based on a different storage model. The first option assumed that the application stored the survey response data in table storage. The second option, which was the one chosen, assumed that the application stored the survey response data in BLOB storage.

Paging with Table Storage

The developers at Tailspin looked at two features of the Windows Azure table storage API to help them design this solution. The first feature is the continuation token that you can request from a query

that enables you to execute a subsequent query that starts where the previous query finished. You can use a stack data structure to maintain a list of continuation tokens that you can use to go forward one page or back one page through the survey responses. You must then keep this stack of continuation tokens in the user's session state to enable navigation for the user.

Note: For an example of this approach, see the section, "Implementing Paging with Windows Azure Table Storage" in Chapter 8, "Phase 4: Adding More Tasks and Tuning the Application," of the book, *Windows Azure Architecture Guide, Part 1: Moving Applications to the Cloud*, available at <http://msdn.microsoft.com/en-us/library/ff728592.aspx>.

The second useful API feature is the ability to run asynchronous queries against Windows Azure table storage. This can help avoid thread starvation in the web server's thread pool in the web role by offloading time-consuming tasks to a background thread.

Paging with Blob Storage

The assumption behind this solution is that each survey answer is stored in a separate BLOB. To access the BLOBs in a predefined order, you must maintain a list of all the BLOBs. You can then use this list to determine the identity of the previous and next BLOBs in the sequence and enable the user to navigate backward and forward through the survey responses.

To support alternative orderings of the data, you must maintain additional lists.

Comparing the Solutions

The previous section, which discusses alternative approaches to saving survey response data, identified lower transaction costs as the key advantage of saving directly to BLOB storage instead of using a delayed write pattern to save to table storage. Paging with table storage is complex because you must manage the continuation stack in the user's session state.

However, you must consider the costs and complexity associated with maintaining the ordered list of BLOBs in the second of the two alternative solutions. This incurs two additional storage transactions for every new survey; one as the list is retrieved from BLOB storage, and one as it is saved back to BLOB storage. This is still fewer transactions per survey response than the table-based solution. Furthermore, it's possible to avoid using any session state by embedding the links to the next and previous BLOBs directly in the web page.



The obvious solution (in this case to use table storage) is not always the best solution.

Inside the Implementation

Now is a good time to walk through the data paging functionality that Tailspin implemented in more detail. As you go through this section, you may want to download the Visual Studio solution for the Tailspin Surveys application from <http://wag.codeplex.com/>.

This walkthrough is divided into two sections. The first section describes how the application maintains an ordered list of BLOBs. The second section describes how the application uses this list to page through the responses.

Maintaining the Ordered List of Survey Responses

The surveys application already uses an asynchronous task in a worker role to calculate the summary statistical data for each survey. This task periodically process new survey answers from a queue, and as it processes each answer, it updates the ordered list of BLOBs containing survey results. The application assigns each BLOB an ID that is based on the tick count when it is saved, and the application adds a message to a queue to track new survey responses.

The following code example from the **SurveyAnswerStore** class shows how the application creates a BLOB ID, saves the BLOB to the correct BLOB container for the survey, and adds a message to the queue that tracks new survey responses.

```
public void SaveSurveyAnswer(SurveyAnswer surveyAnswer)
{
    var surveyBlobContainer = this.surveyAnswerContainerFactory
        .Create(surveyAnswer.Tenant, surveyAnswer.SlugName);
    surveyBlobContainer.EnsureExist();
    DateTime now = DateTime.UtcNow;
    surveyAnswer.CreatedOn = now;
    var blobId = now.GetFormattedTicks();
    surveyBlobContainer.Save(blobId, surveyAnswer);
    this.surveyAnswerStoredQueue.AddMessage(
        new SurveyAnswerStoredMessage
        {
            SurveyAnswerBlobId = blobId,
            Tenant = surveyAnswer.Tenant,
            SurveySlugName = surveyAnswer.SlugName
        });
}
```

The **Run** method in the **UpdatingSurveyResultsSummaryCommand** class in the worker role calls the **AppendSurveyAnswerIdToAnswerList** method for each survey response in the queue of new survey responses.

Surveys uses an asynchronous task in a worker role to maintain the ordered list of BLOBs.

The following code example shows how the **AppendSurveyAnswerIdToAnswerList** method in the **SurveyAnswerStore** class.

```
public void AppendSurveyAnswerIdToAnswersList(string tenant,
    string slugName, string surveyAnswerId)
{
    string id = string.Format(CultureInfo.InvariantCulture,
        "{0}-{1}", tenant, slugName);
    var answerIdList = this.surveyAnswerIdsListContainer.Get(id)
        ?? new List<string>(1);
    answerIdList.Add(surveyAnswerId);
    this.surveyAnswerIdsListContainer.Save(id, answerIdList);
}
```

The application stores list of survey responses in a **List** object, which it serializes in the JSON format and stores in a BLOB. There is one BLOB for every survey.

Implementing the Paging

When the Surveys application displays a survey response, it finds the BLOB that contains the survey response by using a BLOB ID. It can use the ordered list of BLOB IDs to create navigation links to the next and previous survey responses.

The following code example shows the **BrowseResponses** action method in the **SurveysController** class in the TailSpin.Web project.

```
public ActionResult BrowseResponses(string tenant,
    string surveySlug, string answerId)
{
    SurveyAnswer surveyAnswer = null;
    if (string.IsNullOrEmpty(answerId))
    {
        answerId = this.surveyAnswerStore
            .GetFirstSurveyAnswerId(tenant, surveySlug);
    }

    if (!string.IsNullOrEmpty(answerId))
    {
        surveyAnswer = this.surveyAnswerStore
            .GetSurveyAnswer(tenant, surveySlug, answerId);
    }

    var surveyAnswerBrowsingContext = this.surveyAnswerStore
        .GetSurveyAnswerBrowsingContext(tenant,
            surveySlug, answerId);
}
```



The application adds new responses to the queue in the order that they are received. When it retrieves messages from the queue and adds the BLOB IDs to the list, it preserves the original ordering.

```
var browseResponsesModel = new BrowseResponseModel
{
    SurveyAnswer = surveyAnswer,
    PreviousAnswerId =
        surveyAnswerBrowsingContext.PreviousId,
    NextAnswerId = surveyAnswerBrowsingContext.NextId
};

var model = new TenantPageViewData<BrowseResponseModel>
    (browseResponsesModel);
model.Title = surveySlug;
return this.View(model);
}
```

This action method uses the **GetSurveyAnswer** method in **SurveyAnswerStore** class to retrieve the survey response from BLOB storage and the **GetSurveyAnswerBrowsingContext** method to retrieve a **SurveyBrowsingContext** object that contains the BLOB IDs of the next and previous BLOBs in the sequence. It then populates a model object with this data to forward on to the view.

SESSION DATA STORAGE

The Surveys application must maintain some state data for each user as they design a survey. This section describes the design and implementation of user state management in the Surveys application.

Goals and Requirements

When a user designs a new survey in the Surveys application, they create the survey and then add questions one-by-one to the survey until it's complete. Figure 6 shows the sequence of screens when a user creates a survey with two questions.

The Surveys application must maintain session state while a user designs a survey.

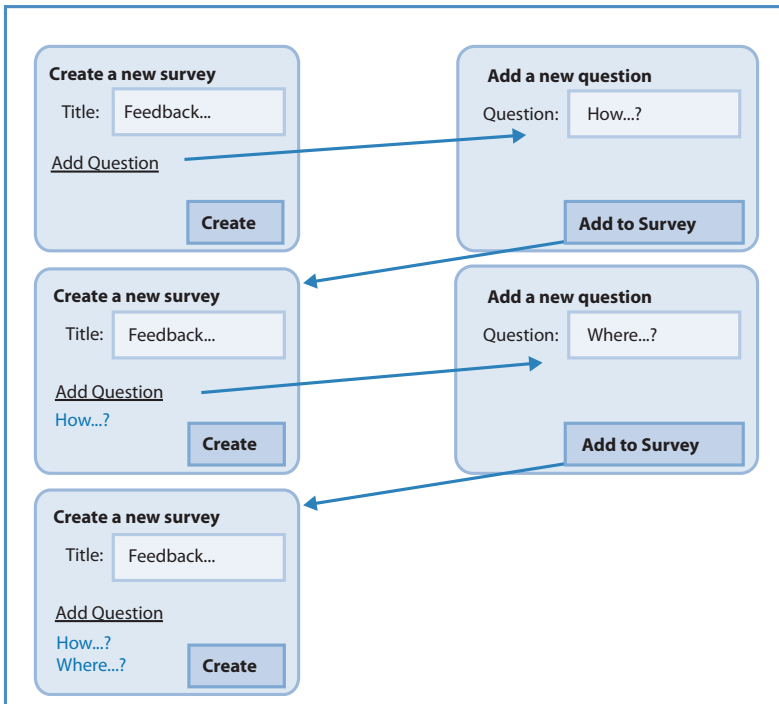


FIGURE 6
Creating a survey with two questions

As you can see in the diagram, this scenario involves two different screens that require the application to maintain state as the user adds questions to the survey. The developers at Tailspin considered three options for managing the session state:

- Use JavaScript and manage the complete workflow on the client. Then use Ajax calls to send the survey to the server after it's complete.
- Use the standard, built-in **Request.Session** object to store the intermediate state of the survey while the user is creating it. Because the Tailspin web role will run on several node instances, Tailspin cannot use the default, in-memory session state provider, and would have to use another storage provider such as the **TableStorageSessionStateProvider** from the Windows Azure samples.
- Use an approach similar to ViewState that serializes and deserializes the workflow state and passes it between the two pages.

Note: You can download the Windows Azure samples that include the **TableStorageSessionStateProvider** from <http://code.msdn.microsoft.com/windowsazuresamples>.

The Solution

You can compare the three options using several different criteria. Which criteria are most significant will depend on the specific requirements of your application.

Simplicity

Something that is simple to implement is also usually easy to maintain. The first option is the most complex of the three, requiring JavaScript skills and good knowledge of an Ajax library. It is also difficult to unit test. The second option is the easiest to implement because it uses the standard ASP.NET **Session** object. Using the session state provider is simply a matter of “plugging-in” the **TableStorageSessionStateProvider** in the Web.config file. However, after the **TableStorageSessionStateProvider** finishes using the state data, it does not automatically clean it up, so you would have to implement this functionality. The third option is moderately complex, but you can simplify the implementation by using some of the features in the ASP.NET MVC 2 Futures package. Unlike the second option, it doesn’t require any server side setup or configuration other than including the MVC Futures code in the solution.

Note: You can download the ASP.NET MVC 2 Futures code from <http://aspnet.codeplex.com/releases/view/41742>.

Cost

The first option has the lowest costs because it uses a single POST message to send the completed survey to the server. The second option has moderate costs that arise from the table storage transaction costs incurred whenever the application writes session state to or reads session state from table storage. The amount of storage required is not likely to be significant. Tailspin could estimate these costs based on the expected number of questions created per day and the average size of the questions. The third option has higher costs than the second option because its costs arise from bandwidth usage. Again, Tailspin can estimate the costs based on the expected number of questions created per day and the average size of the questions.



With both the second and third options, the data is encoded as Base64, so any estimate of the average question size must consider this.

Performance

The first option offers the best performance because the client performs almost all the work with no roundtrips to the server until the final POST message containing the complete survey. The second option will introduce some latency into the application; the amount of latency will depend on the number of concurrent sessions, the amount of data in the session objects, and the latency between the web role and Windows Azure table storage. The third option will also introduce some latency because each question will require a round-trip to the server and each HTTP request and response message will include all the current state data.

Scalability

All three options scale well. The first option scales well because it doesn't require any session state data outside the browser, the second and third options scale well because they are "web-farm friendly" solutions that you can deploy on multiple web roles.

Robustness

The first option is the least robust, relying on "fragile" JavaScript code. The second option uses sample code that is not production quality, although you could enhance it. The third option is the most robust, using easily testable server-side code.

User Experience

The first option provides the best user experience because there are no postbacks during the survey creation process. The other two options require a postback for each question.

Security

The first two options offer good security. With the first option, the browser holds all the survey in memory until the survey creation is complete, and with the second option, the browser just holds a cookie with a session ID, while Windows Azure table storage holds the survey data. The third option is not so secure because it simply serializes the data to Base64 without encrypting it. It's possible that sensitive data could "leak" during the flow between pages.

Tailspin decided to use the third option that passes the survey design data between the two pages as a serialized object. Instead of using cookies, the application stores the data in a hidden field in the form on the pages involved.

Inside the Implementation

Now is a good time to walk through the session data storage implementation that Tailspin selected in more detail. As you go through this section, you may want to download the Visual Studio solution for the Tailspin Surveys application from <http://wag.codeplex.com/>.

The following code example shows how the Action methods in the **SurveysController** controller class in the TailSpin.Web project deserialize the data sent from the browser. Notice how the *hiddenSurvey* parameter has the **Deserialize** attribute from the ASP.NET MVC 2 Futures package applied.

```
[HttpGet]
public ActionResult New([Deserialize]Survey hiddenSurvey)
{
    if (hiddenSurvey == null)
    {
        hiddenSurvey = (Survey)this.TempData["hiddenSurvey"];
    }

    if (hiddenSurvey == null)
    {
        hiddenSurvey = new Survey(); // First time to the page
    }

    var model = new TenantPageViewData(hiddenSurvey);
    model.Title = "New Survey";
    return this.View(model);
}

[HttpPost]
public ActionResult New(Survey contentModel, [Deserialize]Survey
hiddenSurvey)
{
    contentModel.Questions = hiddenSurvey.Questions;

    if (!this.ModelState.IsValid)
    {
        var model = new TenantPageViewData<Survey>(contentModel);
        model.Title = "New Survey";
        return this.View(model);
    }
}
```

```

contentModel.Tenant = this.TenantName;
try
{
    this.surveyStore.SaveSurvey(contentModel);
}
catch (DataServiceRequestException ex)
{
    ...
}

return this.RedirectToAction("Index");
}

```

The following code example from the `NewQuestion.aspx` view shows how the application causes the form data from the client to be serialized by using the **Html.Serialize** element.

```

<% using (Html.BeginForm("AddQuestion", "Surveys")) {%>
    <%= Html.ValidationSummary(true) %>
    <%= Html.Serialize("hiddenSurvey")%>
    <%= Html.Hidden("referrer", "addQuestion") %>

    <dl>
        <dt>
            <%= Html.LabelFor(model => model.ContentModel.Text) %>
        </dt>
        <dd>
            <%= Html.TextBoxFor(model => model.ContentModel.Text,
                new { size = "60" })%>
            ...
        </dd>
    </dl>
<% } %>

```

DISPLAYING QUESTIONS

The application stores the definition of a survey and its questions in table storage. To render the questions in a page in the browser, the application uses the MVC **EditorExtensions** class.

When the **Display** action method in the **SurveysController** class in the `TailSpin.Web.Survey.Public` project builds the view to display the survey, it retrieves the survey definition from storage, populates a model, and passes the model to the view. The following code example shows this action method.



Tailspin chose this mechanism to render the questions because it makes it easier to include additional question types at a later date.

```
[HttpGet]
public ActionResult Display(string tenant, string surveySlug)
{
    var surveyAnswer = CallGetSurveyAndCreateSurveyAnswer(
        this.surveyStore, tenant, surveySlug);

    var model =
        new TenantPageViewData<SurveyAnswer>(surveyAnswer);
    model.Title = surveyAnswer.Title;
    return this.View(model);
}
```

The view uses the **EditorExtensions** class to render the questions. The following code example shows how the Display.aspx page uses the **Html.EditorFor** element that is defined in the **System.Web.Mvc.EditorExtensions** class.

```
<% for (int i = 0;
    i < this.Model.ContentModel.QuestionAnswers.Count; i++ ) { %>
...
    <%= Html.EditorFor(m=>m.ContentModel.QuestionAnswers[i],
        QuestionTemplateFactory.Create(
            Model.ContentModel.QuestionAnswers[i].QuestionType)) %>
...
<% } %>
```

This element iterates over all the questions that the controller retrieved from storage and uses the **QuestionTemplateFactory** utility class to determine which user control (.ascx files) to use to render the question. The user controls FiveStar.ascx, MultipleChoice.ascx, and SimpleText.ascx are in the EditorTemplates folder in the project.

DISPLAYING THE SUMMARY STATISTICS

The asynchronous task (described in Chapter 4, “Building a Scalable, Multi-Tenant Application for Windows Azure”) that generates the summary statistics from surveys stores the summaries in BLOB storage, using a separate BLOB for each survey. The Surveys application displays these summary statistics in the same way that it displays questions. The following code example shows the **Analyze** action method in the **SurveysController** class in the TailSpin.Web project that reads the results from BLOB storage and populates a model.

```
public ActionResult Analyze(string tenant, string surveySlug)
{
    var surveyAnswersSummary =
        this.surveyAnswersSummaryStore
```

```
        .GetSurveyAnswersSummary(tenant, surveySlug);

    var model =
        this.CreateTenantPageViewData(surveyAnswersSummary);
    model.Title = surveySlug;
    return this.View(model);
}
```

The view again uses the **Html.EditorFor** element to render the questions. The following code example shows a part of the `Analyze.aspx` file.

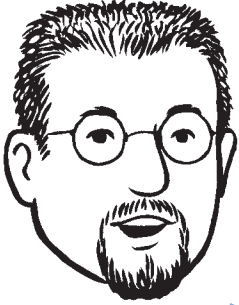
```
<% for (int i = 0;
    i < this.Model.ContentModel.QuestionAnswersSummaries.Count;
    i++ ) { %>
<li>
    <%: Html.DisplayFor(m => m.ContentModel
        .QuestionAnswersSummaries[i],
        "Summary-" + TailSpin.Web.Survey.Public.Utility
        .QuestionTemplateFactory.Create
        (Model.ContentModel.QuestionAnswersSummaries[i]
        .QuestionType))%>
</li>
<% } %>
```

The user control templates for rendering the summary statistics are named `Summary-FiveStar.ascx`, which displays an average for numeric range questions; `Summary-MultipleChoice.ascx`, which displays a histogram; and `Summary-SimpleText.ascx`, which displays a word cloud. You can find these templates in the `DisplayTemplates` folder in the `TailSpin.Web` project. To support additional question types, you must add additional user control templates to this folder.

Using SQL Azure

The Surveys application uses Windows Azure storage to store survey definitions and survey responses. Tailspin chose to use Windows Azure storage because of its lower costs and because those costs depend on the amount of usage, both in terms of capacity used and the number of storage transactions per month. However, to control the costs associated with storage, the Surveys application does not offer a great deal of flexibility in the way that subscribers can analyze the survey responses. A subscriber can browse through the responses to a survey in the order that users submitted their responses, and a subscriber can view a set of “pre-canned” summary statistical data for each survey.

SQL Azure allows subscribers to perform complex analysis on their survey results.



Giving each subscriber a separate instance of SQL Azure enables them to customize the data, and it simplifies the security model.

To extend the analysis capabilities of the Surveys application, Tailspin allows subscribers to dump their survey responses into a SQL Azure database. They can then perform whatever detailed statistical analysis they want, or they can use this as a mechanism to download their survey results to an on-premise application by exporting the data from SQL Azure.

This feature is included in the monthly fee for a Premium subscription. Subscribers at other levels can purchase this feature as an add-on to their existing subscription.

GOALS AND REQUIREMENTS

The application must be able to export all survey data to SQL Azure, including the question definitions in addition to the survey responses.

Subscribers who choose to use this feature have their own, private instance of SQL Azure to ensure that they are free to analyze and process the data in any way that they see fit. For example, they may choose to create new tables of summary data or design complex data-analysis queries. A private instance of SQL Azure also helps to ensure that their data remains confidential.

THE SOLUTION

During the on-boarding process, the application will provision a new SQL Azure instance for those subscribers who have access to this feature. The provisioning process will create the necessary tables in the database. As part of the on-boarding process, the Surveys application saves the information that the application and the subscriber require to access the SQL Azure instance in BLOB storage as part of the subscriber's details.

A task in a worker role monitors a queue for messages that instruct it to dump a subscriber's survey results to tables in SQL Azure. Figure 7 shows the table structure in SQL Azure.

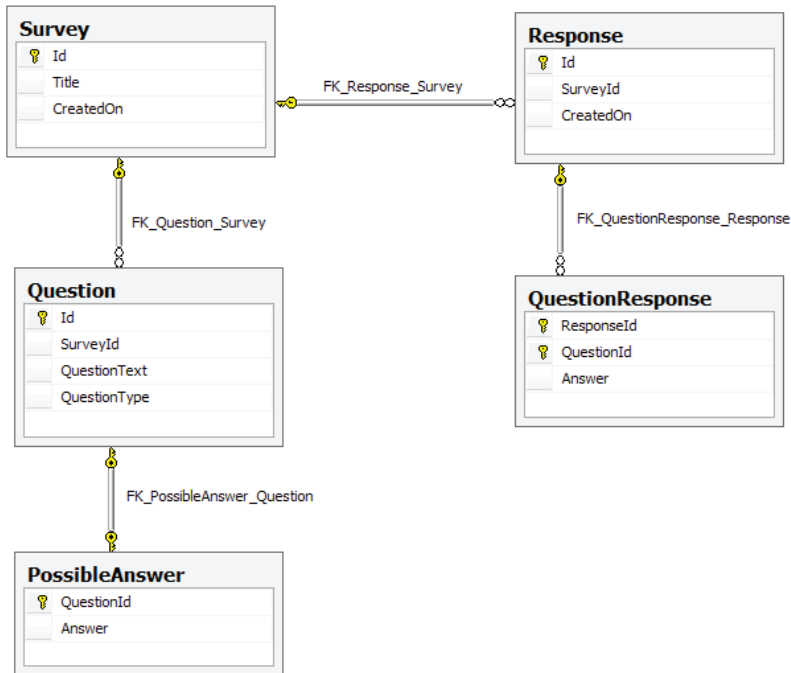


FIGURE 7
Surveys table structure in SQL Azure

INSIDE THE IMPLEMENTATION

Now is a good time to walk through the code that dumps the survey responses to SQL Azure in more detail. As you go through this section, you may want to download the Visual Studio solution for the Tailspin Surveys application from <http://wag.codeplex.com/>.

The following code example shows the task in the worker role that executes when it is triggered by a message in a queue. The **Run** method is in the **TransferSurveysToSqlAzureCommand** class.

```

public void Run(SurveyTransferMessage message)
{
    Tenant tenant =
        this.tenantStore.GetTenant(message.Tenant);
    this.surveySqlStore.Reset(
        tenant.SqlAzureConnectionString, message.Tenant,
        message.SlugName);

    Survey surveyWithQuestions = this.surveyRespository
        .GetSurveyByTenantAndSlugName(message.Tenant,
        message.SlugName, true);
}
    
```

```
IEnumerable<string> answerIds = this.surveyAnswerStore
    .GetSurveyAnswerIds(message.Tenant,
        surveyWithQuestions.SlugName);

SurveyData surveyData = surveyWithQuestions.ToDataModel();

foreach (var answerId in answerIds)
{
    SurveyAnswer surveyAnswer = this.surveyAnswerStore
        .GetSurveyAnswer(surveyWithQuestions.Tenant,
            surveyWithQuestions.SlugName, answerId);

    var responseData = new ResponseData
    {
        Id = Guid.NewGuid().ToString(),
        CreatedOn = surveyAnswer.CreatedOn
    };

    foreach (var answer in surveyAnswer.QuestionAnswers)
    {
        var questionResponseData = new QuestionResponseData
        {
            QuestionId = (from question in
                surveyData.QuestionDatas
                where question.QuestionText ==
                    answer.QuestionText
                select question.Id).FirstOrDefault(),
            Answer = answer.Answer
        };

        responseData.QuestionResponseDatas
            .Add(questionResponseData);
    }
    if (responseData.QuestionResponseDatas.Count > 0)
    {
        surveyData.ResponseDatas.Add(responseData);
    }
}
```

```
    }  
  }  
  
  this.surveySqlStore  
    .SaveSurvey(tenant.SqlAzureConnectionString, surveyData);  
}
```

The *message* parameter to this method identifies the survey to export. The method first resets the survey data in SQL Azure before it iterates over all the responses to the survey and saves the most recent data to SQL Azure. The application does not attempt to parallelize this operation; for subscribers who have large volumes of data, the dump operation may run for some time.

The application uses LINQ to SQL to manage the interaction with SQL Azure. The following code from the **SurveySqlStore** class shows how the application uses the **SurveyData** and **SurveySqlDataContext** classes. The SurveySql.dbml designer creates these classes.

```
public void SaveSurvey(string connectionString,  
    SurveyData surveyData)  
{  
    using (var dataContext =  
        new SurveySqlDataContext(connectionString))  
    {  
        dataContext.SurveyDatas.InsertOnSubmit(surveyData);  
        try  
        {  
            dataContext.SubmitChanges();  
        }  
        catch (SqlException ex)  
        {  
            Trace.TraceError(ex.TraceInformation());  
            throw;  
        }  
    }  
}
```



This task is part of the worker role described in Chapter 4, “Building a Scalable, Multi-Tenant Application for Windows Azure.” It is triggered by a message in a queue instead of by a schedule.

References and Resources

For more information about Windows Azure storage services, see “Using the Windows Azure Storage Services” on MSDN and the Windows Azure Storage Team Blog:

<http://msdn.microsoft.com/en-us/library/ee924681.aspx>

<http://blogs.msdn.com/b/windowsazurestorage/>

For more information about SQL Azure, see the paper, *SQL Azure Considerations Guide*, available for download at:

<http://wag.codeplex.com>

For further information about continuation tokens and Windows Azure table storage, see the section, “Implementing Paging with Windows Azure Table Storage” in Chapter 8 of the book “*Windows Azure Architecture Guide, Part 1, Moving Applications to the Cloud*,” available at:

<http://msdn.microsoft.com/en-us/library/ff728592.aspx>.

This appendix focuses on the scenarios for updating a Windows Azure™ technology platform service that you have previously deployed. It contains the following topics:

- “Options for Updating a Windows Azure Service”
- “Limitations When Updating a Windows Azure Service”
- “Design Considerations for Maximizing Upgradability”
- “About Upgrade Domains and Fault Domains”
- “Using Upgrade Domains for In-Place Upgrades”
- “Using the Windows Azure Service Management API”

OPTIONS FOR UPDATING A WINDOWS AZURE SERVICE

You have three options for updating a deployed Windows Azure service:

- **Redeploy the service.** Suspend and then delete the service, and then deploy the new version.
- **Perform a staged deployment.** Upload the new package and swap it with the existing production version. This is referred to as a VIP swap because it simply swaps the visible IP addresses of the services.
- **Perform an in-place upgrade.** The new package is uploaded and applied to the running instances of the service. There are options that allow you to update individual instances and individual roles within each instance.

The first of these three options, redeployment, requires downtime for your service, while the second two options allow you to perform updates without incurring service downtime. However, there are limitations on the changes you can make to a deployed service when using the second two options. For more information, see the follow-

ing section, “Limitations When Updating a Windows Azure Service”.

For information about how you can use in-place upgrades, see the sections, “About Upgrade Domains and Fault Domains” and “Using Upgrade Domains for In-Place Upgrades,” later in this guidance.

For more information about updating a deployed Windows Azure Service, see “Upgrading a Service” on MSDN® (<http://msdn.microsoft.com/en-us/library/ee517254.aspx>).

LIMITATIONS WHEN UPDATING A WINDOWS AZURE SERVICE

The following are the general rules for choosing the appropriate type of update to perform for a deployed service:

- If you need to change the number or type of endpoints for existing roles, you must delete and redeploy the service.
- If you need to change the service definition, you must perform a staged (VIP swap) deployment (or you can redeploy the service).
- If there is no change to the service definition (.csdef) file, you can perform an in-place update.

The following table shows in more detail the types of changes you can make to a deployed service using the three update options.

Changes permitted to hosting, services, and roles	In-place upgrade	Staged (VIP swap)	Delete and re-deploy
Operating system version	No	Yes	Yes
.NET trust level	No	Yes	Yes
Virtual machine size	No	Yes	Yes
Local storage settings	No	Yes	Yes
Number of roles for a service	No	Yes	Yes
Number of instances of a particular role	Yes	Yes	Yes
Number or type of endpoints for a service	No	No	Yes
Names and values of configuration settings	No	Yes	Yes
Values (but not names) of configuration settings	Yes	Yes	Yes
Add new certificates	No	Yes	Yes
Change existing certificates	Yes	Yes	Yes
Deploy new code	Yes	Yes	Yes

Note: *When you redeploy a service, you will be allocated new visible IP addresses for the endpoints, which can interrupt access to the service. To reduce the possibility of the need to redeploy your service, ensure that you define all the endpoints you will require for the service, including both HTTP and HTTPS endpoint types for each one.*

Updating Only the Configuration of a Service

If you need to edit only the configuration information for the service, and you do not need to deploy new code, you can do this using the Windows Azure Developer Portal or the Windows Azure Service Management API. You can edit the service configuration (.cscfg) file on the portal or upload a new .cscfg file through the portal. However, you must ensure that the uploaded file schema matches the existing service definition. The service will detect and apply the new configuration settings after a preset period.

Redeploying a Service Without Incurring Downtime

If you need to redeploy your service (typically, this occurs when you need to change the number of roles or endpoints it contains), you can change the Domain Name System (DNS) entries at your domain registrar or in your own DNS server that contains the DNS records for your service domain. This allows you to redirect requests to the new version as the DNS change propagates through the world's DNS servers. After one or two days, you can remove the original service.

For example, if your service DNS name is `myservice.contoso.com` and you map `www.contoso.com` to this in your DNS, you could create a new service at `newservice.contoso.com` and then map `www.contoso.com` to it using a CNAME record. After the DNS change propagates, all requests will go to the new service and you can remove the old service.

DESIGN CONSIDERATIONS FOR MAXIMIZING UPGRADABILITY

Consider the following when you are designing your application to maximize the capabilities for upgrading it, and to make it easier to deploy updated versions:

- If you intend to use in-place upgrades, ensure that state held within the service is replicated or stored outside the roles, so that it is not lost when roles restart after the upgrade process completes. Roles must correctly re-initialize when they restart, and they should not assume that locally held state is available.

- Avoid creating tables within the run-time operations or business logic of a service. Use a separate initialization routine to generate any tables required by the service roles. This avoids performance degradation through exceptions thrown when a table already exists. For sample or demo applications, you can perform the initialization in the **OnStart** method in the role. However, instead of multiple role instances attempting to initialize the storage at the same time in a production environment, it is better to have a separate console application or script to perform the initialization. When deleting and recreating tables, ensure you wait for approximately one minute for the existing table to be fully removed or an exception will be thrown when creating a new table with the same name.
- Consider setting the **IgnoreMissingProperties** property to **true** for the **DataContext** you use to access tables. By default, this property is set to **false**, which means that, when you add new properties to an existing table, existing clients that are not aware of the new property will receive an ADO.NET Data Services exception. When you set **IgnoreMissingProperties** to **true**, ADO.NET Data Services will not throw an exception when the client is not aware of the added properties. Although existing clients will not be able to use the added properties, this does allow you to extend the schema of a table and then upgrade clients later. This is particularly important in two specific scenarios:
 - For public services where you cannot be sure that all users will have upgraded versions of the client software
 - When you perform in-place upgrades, so that the roles in upgrade domains that have not yet been upgraded can still communicate with the upgraded roles and tables
- Properties in a table are identified by the combination of the name and the data type, so it is possible to have more than one property with the same name. If you need to change the data type of a property, you must add a new property with the required name and data type, copy the data from the old property to the new property for each row (converting it to the appropriate new type if required), and then delete the old property from the table. Ensure that you set the **IgnoreMissingProperties** property to **true** for the **DataContext** when performing these operations.
- Consider storing a table version number in each row of your tables. This allows newer clients to detect the schema version and take appropriate action, such as storing a default value in

an added property when they update or add a row. After all clients are upgraded to understand the new schema, they can be updated again to make full use of the new schema and the added properties; and the row version number can be updated.

ABOUT UPGRADE DOMAINS AND FAULT DOMAINS

When you deploy a service to Windows Azure, you specify the number of instances of each role you want to run. The Windows Azure Service Level Agreement guarantees the level of connectivity uptime for your service only if you deploy two or more instances of each role, which allows the service to continue to be available even if a server that hosts one instance of a role fails.

Fault Domains

The infrastructure of each Windows Azure data center is notionally divided into multiple sections known as fault domains. These sections of the infrastructure (which are not necessarily individual servers or server racks) are designed in such a way that a failure of one fault domain is extremely unlikely to affect any other fault domain. When you deploy a service, the Windows Azure Fabric Controller automatically locates the roles in at least two different fault domains so that a failure in one domain will not affect all instances of your service.

Note: *There is no guarantee that your role instances will be located in more than two fault domains, so there is no point in deploying more than two instances of a role unless this is necessary to meet the load anticipated for your service or for other operational reasons.*

Upgrade Domains

Whereas a fault domain is a physical location within a data center, an upgrade domain is a logical unit of deployment, although it also affects physical role location. An upgrade domain is primarily a way of logically subdividing your role instances into groups to allow you to more closely control how updates to the instances occur when you perform an in-place upgrade to your service. However, as long as you define more than one upgrade domain (and you have more than one instance of each role), the Windows Azure Fabric Controller will ensure that the role instances within each domain are located in at least two different fault domains.

The main advantage of using upgrade domains and in-place upgrades to a service is that the Windows Azure Fabric Controller will update all the roles within the same upgrade domain as one

transacted operation. Only the roles in that upgrade domain will be stopped, upgraded, and restarted. This means that the roles in other upgrade domains will continue to be available and can service requests. After the upgrade to one domain completes and the roles restart, the upgrade process automatically moves to the next upgrade domain.

If you are concerned that the performance of your application will degrade during the upgrade process when the roles in an upgrade domain are stopped, you should consider configuring additional role instances in an additional upgrade domain to maintain capacity.

You can also specify that updates be applied to only a single upgrade domain, and to only specific types of roles within that domain. This allows you, for example, to upgrade just the web roles without stopping and restarting the worker roles (which would result in the loss of any local state they may hold). If you specify only a single upgrade domain for a service, you will not benefit from the capability to maintain availability when performing an in-place upgrade.

You specify the number of upgrade domains for your service using the **upgradeDomainCount** attribute of the root **ServiceDefinition** element in the service definition (.csdef) file. The default is five upgrade domains for each service. For information about the **ServiceDefinition** element, see the relevant section of “Service Definition Schema” on MSDN (<http://msdn.microsoft.com/en-us/library/ee758711.aspx#ServiceDefinition>).

Within your code, you can query the **UpdateDomain** property of the **RoleInstance** class to discover which upgrade domain a role is located within (note the inconsistent naming of this property). The **RoleInstance** class also exposes the **FaultDomain** property, but this will only return the values 1 or 2 because a service is only guaranteed to run in two fault domains, even though it may be physically located in more than two.

USING UPGRADE DOMAINS FOR IN-PLACE UPGRADES

You can perform either an automatic or a manual upgrade to a service. If you specify an automatic upgrade, the Windows Azure Fabric Controller will automatically process all the roles in each domain. It works through the upgrade domains one by one, stopping, upgrading, and restarting the roles in that upgrade domain before moving on to the next upgrade domain.

If you specify a manual upgrade, the Windows Azure Fabric Controller will process only the roles in the specified upgrade domain. You can then check that the update succeeded and that the service is performing correctly before applying the upgrade to other domains. However, because the load balancer will distribute requests between

all running instances of your service, you cannot verify correct behavior just by browsing the service. Instead, you should use instrumentation such as trace messages and performance counters within the service roles to create the appropriate information that identifies the performance of each role instance.

If you want to upgrade only a specific role within each domain, you can specify this role when performing a manual upgrade and it will apply to only that role. This means that you may be able to maintain state if it is held in a role that you do not need to upgrade.

You can initiate an in-place upgrade using the **Upgrade** button on the Hosted Service page of the Windows Azure portal. You specify the following settings before you start the upgrade:

- **Upgrade Mode.** Select **Automatic** to upgrade all domains or **Manual** to upgrade one domain at a time.
- **Package.** Select a service package to upload from your local file system or a service package located in BLOB storage.
- **Configuration Settings.** Select a configuration file to upload from your local file system or a configuration file located in BLOB storage.
- **Service Deployment Name.** Specify a name for the updated deployment.
- **Service Upgrade.** Specify if you want to upgrade all the roles within the service or a single role.

Alternatively, you can use third-party tools or the Windows Azure Service Management API to initiate an in-place upgrade. The next section describes the Windows Azure Service Management API.

USING THE WINDOWS AZURE SERVICE MANAGEMENT API

The Windows Azure Service Management API exposes operations that allow you to perform the tasks available through the Windows Azure portal by writing and executing code. You can use it to manage almost all aspects of your service except for accessing billing data, creating subscriptions, or uploading Management API certificates.

The Service Management API uses Representational State Transfer (REST) protocol and XML. You must submit requests to the core management address using HTTPS. For example, the following request returns a list of hosted services for a specified subscription ID.

XML

```
https://management.core.windows.net/subscription-id/services/  
hostedservices
```


You can also submit XML documents that contain the information required to perform operations, such as creating a deployment, as the body of a posted request. For more information about the available operations, see “Windows Azure Service Management REST API Reference” on MSDN (<http://msdn.microsoft.com/en-us/library/ee460799.aspx>).

Accessing the Windows Azure Service Management API

You can access the Windows Azure Service Management API using a range of techniques, depending on your own requirements. The following are some examples:

- Use the `csmanage.exe` command line tool, which makes it easy to generate and send requests to the service using command line parameters.
- Use a custom client library and Windows Communication Foundation (WCF) bindings to send requests to the service management API.
- Use the REST protocol and XML documents in applications that do not run on or are not built using the .NET Framework.
- Use PowerShell scripts to generate and send requests.
- Use it in MSBuild tasks to automate and configure deployments

Note: *Microsoft® Visual Studio® development system uses the Windows Azure Service Management API to deploy and manage Windows Azure services.*

For more information about the Windows Azure Service Management API, see “About the Service Management API” on MSDN (<http://msdn.microsoft.com/en-us/library/ee460807.aspx>).

The `csmanage.exe` command line tool, together with other useful resources to help you learn and use the Windows Azure Service Management API, is available from “Windows Azure Code Samples” in the MSDN Code Gallery (<http://code.msdn.microsoft.com/windowsazuresamples>).

A set of PowerShell scripts for accessing the Windows Azure Service Management API can be downloaded from “Windows Azure Service Management CmdLets” in the MSDN Code Gallery (<http://code.msdn.microsoft.com/azurecmdlets>).

Performing Service Upgrades Using the Service Management API

Unlike the Windows Azure portal, the Windows Azure Service Management API does not allow you to upload a new service package from your local file system. You must first upload the service package to the Windows Azure Blob service, and then specify the BLOB service address of the package when you initiate the upgrade. However, you can automate access to the BLOB service using the Service Management API. For more information, see “About the Storage Service API” on MSDN (<http://msdn.microsoft.com/en-us/library/dd573356.aspx>).

To upgrade a service using the Windows Azure Service Management API, you first call the Upgrade Deployment operation to initiate the upgrade; specifying the service identifier, the upgrade mode (automatic or manual), the BLOB storage address of the upgrade package, and the name of the role to upgrade (if you are upgrading a single role).

If you specify manual upgrade mode, you must also call the Walk Upgrade Domain operation on each upgrade domain in order, starting from the domain with ID zero, to complete the upgrade. If you call this operation out of order, the API will return an error.

Using the Windows Azure Service Management API to Manage Scaling

One possible use of the Windows Azure Service Management API is the opportunity to react to load on a service by automatically adding or removing role and service instances. For example, you may use a remote application and the Windows Azure Diagnostics mechanism to monitor a range of factors that indicate performance of the service, such as response time, processor load, or operational queue length. The remote application can then use the Windows Azure Service Management API to add or remove instances as the load on the service varies over time. For an example of this technique, see the post, “Auto-scaling in Azure,” on the Windows Azure - Cloud computing blog (<http://blogs.msdn.com/b/gonzalorc/archive/2010/02/07/auto-scaling-in-azure.aspx>).

Appendix B

Debugging and Troubleshooting Windows Azure Applications

This appendix describes some of the technique specific to Windows Azure™ technology platform applications that will help you to detect and resolve issues when building, deploying, and running Windows Azure applications. It contains the following topics:

- “Dependencies for Azure Applications and Projects”
- “Debugging Azure Applications Locally During Development”
- “Common Setup and Deployment Errors with Windows Azure”
- “Using IntelliTrace with Applications Deployed to Windows Azure”
- “Using Windows Azure Diagnostics”
- “Resolving Permission Issues in Partial Trust Mode”
- “More Informatide.”

DEPENDENCIES FOR AZURE APPLICATIONS AND PROJECTS

To ensure that your Windows Azure projects will build and deploy correctly, ensure that you have the most recent versions of the Windows Azure tools installed and configured. Full integration with Windows Azure is available in Microsoft® Visual Studio® 2008 development system and Visual Studio 2010. If you want to use Microsoft IntelliTrace™ software to help debug your applications, you must install Visual Studio 2010 Ultimate Edition. IntelliTrace is described in the section, “Using IntelliTrace with Applications Deployed to Windows Azure,” later in this appendix.

You must also install the Visual Studio Tools for Windows Azure and the Windows Azure SDK. For more information about the requirements for building Windows Azure applications, see “Windows Azure Platform” on MSDN® (<http://msdn.microsoft.com/en-gb/azure/default.aspx>). This page also contains a section, “Interop SDKs and Tools,” that provides links to tools for platforms other than the Microsoft Windows® operating system and tools other than Visual Studio.

DEBUGGING AZURE APPLICATIONS LOCALLY DURING DEVELOPMENT

When working in Visual Studio, the techniques used to debug a Windows Azure application are generally identical to those for non-Azure applications. In addition, you can use the IntelliTrace feature in Visual Studio 2010 Ultimate Edition to output trace and debugging information from an application deployed to Windows Azure. IntelliTrace is described in the section, “Using IntelliTrace with Applications Deployed to Windows Azure,” later in this appendix.

One point to be aware of when you are debugging applications locally is that you should not use the **Debug.WriteLine** or **Console.WriteLine** methods to output debugging information or trace messages. Instead, use **Trace.Write** statements (or other methods of the **Trace** class) to write to the Diagnostic Monitor type (**DiagnosticMonitorTraceListener**) that Visual Studio automatically adds to your configuration file for each role. The following XML shows the section of the configuration file that adds the Diagnostic Monitor Trace Listener.

XML

```
<configuration>
  <system.diagnostics>
    <trace>
      <listeners>
        <add type="Microsoft.WindowsAzure.Diagnostics
              .DiagnosticMonitorTraceListener,
              Microsoft.WindowsAzure.Diagnostics,
              Version=1.0.0.0,
              Culture=neutral,
              PublicKeyToken=31bf3856ad364e35"
              name="AzureDiagnostics">
          <filter type="" />
        </add>
      </listeners>
    </trace>
  </system.diagnostics>
  ...
```

COMMON SETUP AND DEPLOYMENT ERRORS WITH WINDOWS AZURE

The following are some of the common errors that you may encounter when running an application deployed to Windows Azure:

- Failing to set Internet Information Services (IIS) to load the user profile if the application will use the Data Protection Application Programming Interface (DPAPI). To resolve this, open Internet Information Services (IIS) Manager, click **Application Pools** in the left pane, and then click the **ASP.NET v4.0** application pool in the list of application pools. Click **Advanced Settings** in the Actions pane, and then change the value of the **Load User Profile** setting in the **Profiles** section to **True**.
- Failing to enable the HTTP Redirection modules in IIS. MVC 2 and ASP.NET 4 Web Forms applications that use URLs without an extension (such as a URL ending in an action name) might return an HTTP 404 error (see Knowledge Base article 2023146 at <http://support.microsoft.com/?kbid=2023146>). To resolve this, open Programs and Features in Control Panel, and then click **Turn Windows Features On or Off**. Expand the **Internet Information Services** node, expand **World Wide Web Services**, and then expand **Common HTTP Features**. Ensure that **HTTP Error Redirection** option is selected, and then allow the corresponding feature to be installed.
- Omitting a required assembly or other type of resource from your deployment because you did not set the **Copy Local** property for the assembly or resource. This will be reported as a **FileNotFoundException**.
- Using an incorrect Windows Azure storage connection string. This will show up in the method calls that use the connection string.
- Attempting to use a native code library that you forgot to deploy, or using a 32-bit .NET code library, will usually throw a **BadImageException**.
- Attempting to perform a task in your code for which it does not have permission. This will usually throw an **UnauthorizedAccessException**. For information about the permissions available in a Windows Azure application, see the section, “Resolving Permission Issues in Partial Trust Mode,” later in this appendix.
- Using a SQL Server connection string that points to the local SQL Server® or SQL Server Express instance. This will be shown as an **HttpException** with the error message, “Unable to connect to SQL Server database.”

- Using HTTP endpoints or the default local storage account in a diagnostics connection string for a deployed account. This will throw an **ArgumentException**.
- Attempting to read from a queue or table that is not created or initialized.
- Using a certificate that does not have an exported private key. If you use Windows Certificate Manager to create a certificate, you must set the option to export the private key when you export the certificate.
- Including a return statement or code in a worker role that terminates execution of the role. Windows Azure will attempt to restart the role.

You can detect many of the deployment errors using IntelliTrace, as described in the next section, “Using IntelliTrace with Applications Deployed to Windows Azure.”

USING INTELLITRACE WITH APPLICATIONS DEPLOYED TO WINDOWS AZURE

IntelliTrace is a feature in Visual Studio 2010 Ultimate Edition that makes debugging code easier by showing events that occurred in your application as it ran, and the context in which they occurred. It maintains a list of breakpoints hit by your code and the full run-time context of the application at each one. You can review the complete state of the application at each breakpoint to determine where issues arise. You can configure the events that are recorded for each breakpoint; also, you can switch on collection of additional data, such as the parameter values and return values of methods that were called by the code.

IntelliTrace is a useful tool when debugging applications locally, but it is also useful for discovering errors in applications that you deploy to Windows Azure. For example, if a deployed application fails to execute, you can use IntelliTrace to provide information about the cause of the failure. When you use IntelliSense with a deployed Windows Azure application, it automatically collects information about the parameter values and return values of methods that were called by the code.

To use IntelliTrace in a deployed Windows Azure application

1. After you finish creating and testing your application within the local development fabric, right-click the Windows Azure service node in Solution Explorer, and then click **Publish**.

2. In the **Publish Cloud Service** dialog box, enter the deployment information required to deploy your application to Windows Azure.
3. At the bottom of the **Publish Cloud Service** dialog box, select the **Enable IntelliTrace for .NET 4 roles** check box, and then click **OK** to publish your application.
4. After deployment is complete and the application is executing, open Server Explorer and expand the **Windows Azure Compute** node to show a list of attached services.
5. If the service you have deployed is not shown in the list, right-click the **Windows Azure Compute** node, point to **Add Slot**, and then click **New**. Enter the credentials to connect to the service.
6. In the list of services under the **Windows Azure Compute** node, you will see the word **IntelliTrace** in brackets after the service name and deployment state for services that have IntelliTrace enabled. Expand the enabled service node to show a list of roles and expand the role you want to view to show the role instances. The icon for the instance indicates if it is executing, paused, or stopped.
7. Right-click a role instance in a service that has IntelliTrace enabled, and then click **View IntelliTrace Logs** to start downloading the data. After the download completes, the IntelliTrace data opens in a Visual Studio window.
8. To examine details of the IntelliTrace data, select an exception on the Exception Data page, and then click **Start Debugging** or double-click a thread in the list. This opens the IntelliTrace window where you can browse and filter, and view the call stack.
9. To step through your local code using the downloaded IntelliTrace data, open the code, right-click a code line, and then click **Search For This Line In IntelliTrace**. Select a role instance from the list that displays. You can then step through the code and view the values of variables, the call stack, and other useful debugging information.

Note: *You should enable IntelliTrace only while debugging an application or role. When IntelliTrace is enabled, the application and roles will not automatically restart after a failure. This allows Windows Azure to persist the IntelliTrace data about the failure. You must manually restart the application or role.*

For more information about using IntelliTrace, see “Debugging with IntelliTrace” on MSDN (<http://msdn.microsoft.com/en-us/library/dd264915.aspx>).

USING WINDOWS AZURE DIAGNOSTICS

Windows Azure provides integrated features for monitoring, logging, and tracing within the deployed environment; generally referred to as *diagnostics*. These features are most suited to monitoring performance of applications over time, though they can also be used for debugging purposes as well.

Typically, you will use the monitoring features for tasks such as performance and resource usage measurement over time, capacity planning, traffic analysis, billing, and auditing. However, you can also use the capability to create trace messages and log entries, and to access log files and performance counters to perform debugging tasks after the application is deployed to Windows Azure.

About Windows Azure Diagnostics

Windows Azure provides the following diagnostic features:

- A logging trace listener that allows you to write Event Tracing for Windows (ETW), trace, and debug information that you can upload into table storage and access from there. During development, when running locally, the trace information is accessible within the development fabric user interface.
- A monitoring agent that allows you to upload the contents of event logs, IIS logs, crash dumps, and the output from performance counters to BLOB storage and then access it from there. You can determine the appropriate number of instances of each type of data.
- Remote configuration for logging and monitoring that allows you to issue commands from the desktop that cause data to be uploaded to table storage or BLOB storage for specified services and roles. Data can also be automatically uploaded to table storage or BLOB storage on a predefined schedule.

Figure 1 shows an overview of the processes within the Windows Azure diagnostics features. A Windows Azure role can specify configuration information for the Diagnostics Monitor and start monitoring (step 1 in the diagram). The Diagnostics Monitor collects trace and debugging messages generated by the role (2). If the Diagnostics Monitor configuration specifies the collection of data from Windows data sources such as Event logs, performance counters, or IIS logs, this data is also collected (3).

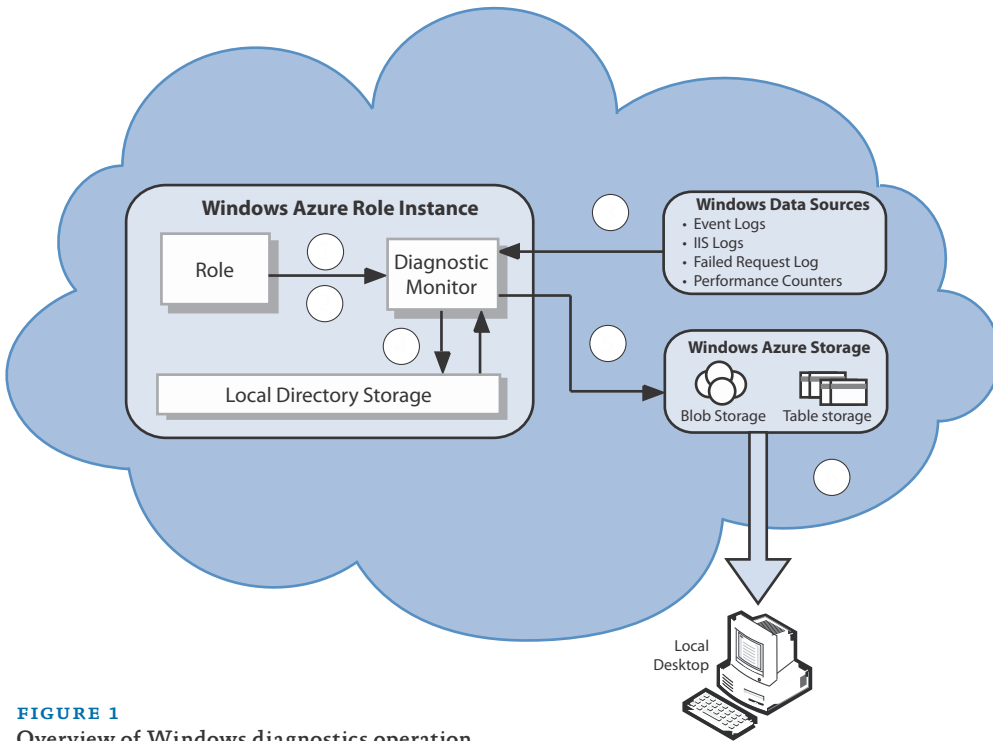


FIGURE 1
Overview of Windows diagnostics operation

The trace, counter, and log data collected from all sources is stored in the local directory storage (4). The maximum total storage size is 4 gigabytes (GB), but you can enforce quotas for each type of information. The stored data is then transferred to the appropriate type of Windows Azure storage (BLOB storage for data from event sources, and table storage for trace and debugging output), either through a scheduled upload or by issuing a command to upload specific sets of the data (5). The data can then be accessed using Windows Azure tools from a remote desktop (6).

Note: Future updates for Windows Azure monitoring will include a dashboard that makes it easy to aggregate monitoring data collected on an hourly, daily, and weekly basis, and integration with Microsoft System Center to allow remote monitoring of applications as part of an enterprise monitoring strategy.

The following table shows the default setting of the Diagnostics Monitor for each type of diagnostics data source, the ways that you initially configure it for each type of data source, and the type of storage to which the data for each type of source is uploaded.

Data source	Default configuration	How to configure	Storage
Trace logs	Enabled, stored locally	Diagnostics API, trace listener	Table
Performance counters	Disabled	Diagnostics API	Table
Windows Event logs *	Disabled	Diagnostics API	Table
Infrastructure logs	Enabled, stored locally	Diagnostics API	Table
IIS logs	Enabled, stored locally	Diagnostics API, Web.config	BLOB
IIS Failed Request logs	Disabled	Diagnostics API, Web.config	BLOB
Application crash dumps	Disabled	Diagnostics API, Crash API	BLOB
Arbitrary logs and files	Disabled	Diagnostics API	BLOB

* You cannot access the Windows Security Event log to extract events because the Windows Azure run-time account does not have permission to access this log.

You can configure a data retention policy for each type of data source, including the quota and removal of aged data. The data stored in table storage is partitioned by the high-order bits of the tick count, so queries that select data based on a time range are efficient.

Note: *The IIS Failed Request Log includes requests that are not completed within a specific interval, as well as requests that fail for other reasons. However, collecting this information incurs considerable performance overhead and can only be enabled or disabled by uploading a configuration update for the service.*

You can also use the Diagnostics Monitor with some application logging frameworks to generate trace and debugging information. For example, you can use it with the Enterprise Library Logging Application Block. The Logging Application Block makes it easy to centralize logging by providing an easy to use and highly configurable mechanism to generating log entries. By configuring the **Microsoft.WindowsAzure.Diagnostics.DiagnosticMonitorTraceListener** class as a trace listener within the configuration of the Logging Application Block, you can use the Enterprise Library **LogWriter** and **TraceManager** objects to generate log trace messages within the local directory storage that you then upload into table storage. The aExpense application that is part of the patterns & practices Windows Azure Guidance project uses this approach.

Limitations of Windows Azure Diagnostics

Although the Windows Azure diagnostics mechanism provides comprehensive information, it is not an ideal solution for debugging and troubleshooting applications during development and deployment. The main issue is the effort required to enable and disable tracing and debugging features, and the delay until the resulting data is available.

To change the configuration of the Diagnostics Monitor, you must upload a new configuration and wait for the application to read and enable this configuration. To change the tracing and debugging code, you must redeploy the application to Windows Azure and wait until the service upgrade process completes before data becomes available. In addition, the delays in obtaining data mean that long-term performance counter polling rates of less than approximately 15 minutes are not practical, which reduces the capability to easily detect run-time issues.

In addition, when a service fails completely, it will generally remove all locally buffered data, including any monitoring and trace information that has not yet been transferred to table or BLOB storage.

However, the diagnostics features are useful for troubleshooting after a failure within a role, because you can download a wide range of information and study it to ascertain the circumstances of the failure and the likely contributing causes.

Using a Custom Tracing, and Debugging Agent

To resolve the two main issues encountered when debugging a deployed application, you can use a custom agent to collect the monitoring, debugging, and trace information and send it back immediately to a client application running on your local computer, or you can write it directly to BLOB or table storage—instead of holding it in the local directory buffer and then transferring it to BLOB or table storage. This helps to overcome the delay encountered while uploading data to storage, and it means that data in the local buffer is not lost if the application fails completely.

An example of a custom monitoring agent is included in the samples provided with the Windows Azure platform AppFabric SDK. Version 1.0 (April 2010) of the Windows Azure platform AppFabric SDK is available from the Microsoft Download Center (<http://www.microsoft.com/downloads/details.aspx?FamilyID=39856a03-1490-4283-908f-c8bfobfad8a5&displaylang=en>). The sample named CloudTrace (in the ServiceBus\Scenarios subfolder) implements a class named **CloudTraceListener** that extends **TraceListener** to send trace events over the service bus to a remote client using one-way multicast events. The project also contains a client console that listens for and displays these events in real time.

You can use this sample as it is, or you can modify it to suit your own requirements. For example, you might adapt it to write the trace messages into table storage in your storage account and then monitor items in this storage account using a remote tool running on your local computer.

Note: *You should use debugging tools such as those described here only while testing and debugging your deployed application. The overhead incurred by diagnostics, especially when the sample rate is high or there is a large number of trace messages, is generally not acceptable for a production application.*

Tools for Working with Windows Azure Diagnostics

There is a growing set of tools available that allow you to view the remote diagnostics information in table storage and BLOB storage, configure diagnostics, and manage Windows Azure services. Within Visual Studio, you can use the Server Manager user interface to view and interact with deployed services. Other tools are Windows Azure Storage Explorer (see <http://azurestorageexplorer.codeplex.com/>), Windows Azure Web Storage Explorer (see <http://storageexplorer.codeplex.com/>), and the Windows Azure Management Tool (MMC) at <http://code.msdn.microsoft.com/windowsazuremmc>.

There is also a set of PowerShell cmdlets (single-function command-line tools) that allow you to manage diagnostics on a remote server. You can use them to script deployments, manage upgrades, and configure scaling of a Windows Azure application in addition to managing the diagnostics configuration. For more information, see “Windows Azure Service Management CmdLets” in the MSDN Code Gallery (<http://code.msdn.microsoft.com/azurecmdlets>).

As an example of the capabilities of the remote tools, Figure 2 shows the Diagnostics section of the Windows Azure Management Tool (MMC). You can use this section to change the configuration of the Windows Azure Diagnostics Monitor for a deployed service, to schedule uploads of the data to Windows Azure storage, and to perform on-demand data transfers to storage. It also provides features for downloading and analyzing the diagnostic data held in BLOB and table storage. You can even create your own custom extensions for the tool to implement other data analysis options.

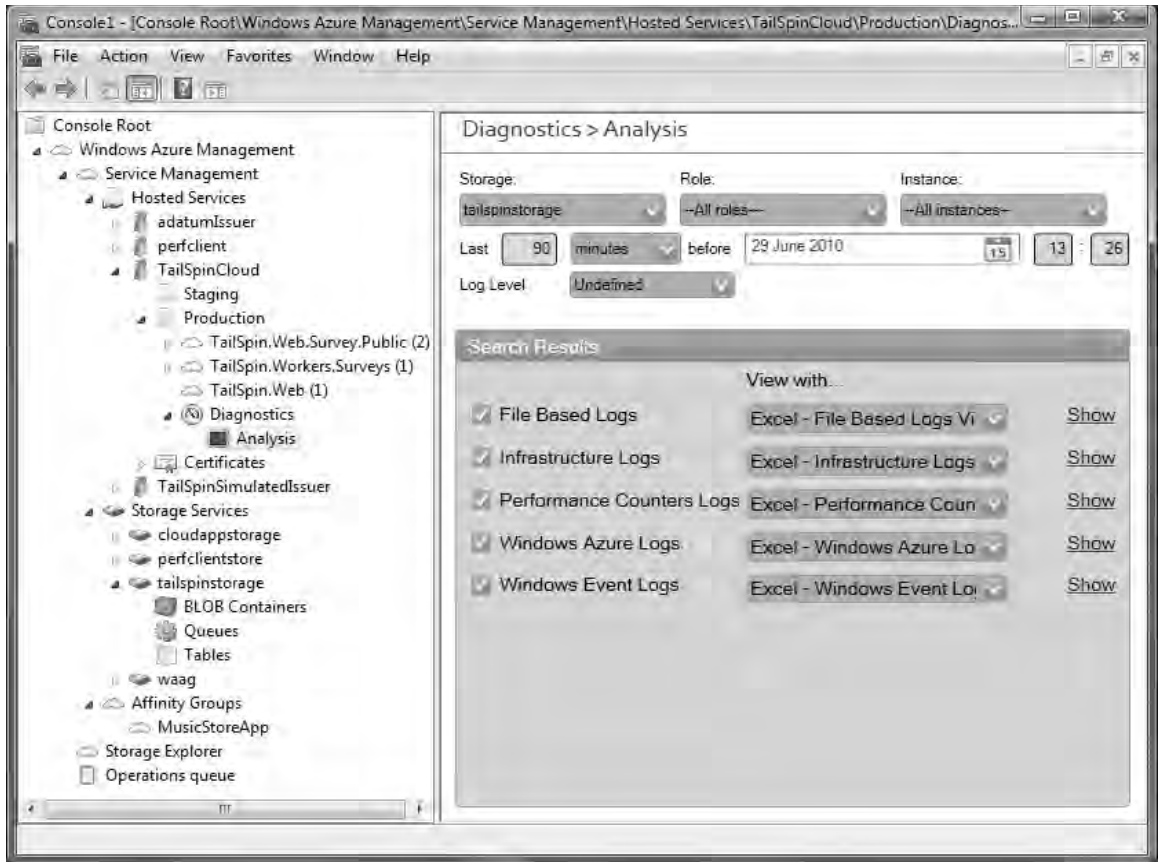


FIGURE 2
Windows Azure Management Tool (MMC) Diagnostics section

Configuring Diagnostics Using Configuration Files

When you create a Windows Azure project in Visual Studio, it automatically adds the Diagnostic Monitor type (**DiagnosticMonitorTraceListener**) to your configuration file for each role.

XML

```
<configuration>
  <system.diagnostics>
    <trace>
      <listeners>
        <add type="Microsoft.WindowsAzure.Diagnostics
              .DiagnosticMonitorTraceListener,
              Microsoft.WindowsAzure.Diagnostics,
              Version=1.0.0.0,
              Culture=neutral,
              PublicKeyToken=31bf3856ad364e35"
              name="AzureDiagnostics">
          <filter type="" />
        </add>
      </listeners>
    </trace>
  </system.diagnostics>
  ...
```

The default configuration settings in the ServiceConfiguration.cscfg file also specify the location of the storage account to use for trace and monitoring data. During development and local debugging, you can use the default value **UseDevelopmentStorage=true**, as shown here.

XML

```
<ServiceConfiguration serviceName="MyService" xmlns="...">
  <Role name="MyRole">
    <Instances count="1" />
    <ConfigurationSettings>
      <Setting name="DiagnosticsConnectionString"
              value="UseDevelopmentStorage=true" />
    </ConfigurationSettings>
  </Role>
</ServiceConfiguration>
```

To specify a cloud storage account in your configuration file, use the following syntax.

```
XML
<Setting name="DiagnosticsConnectionString"
  value="DefaultEndpointsProtocol=https;
  AccountName=...;AccountKey=..." />
```

You can also use the configuration files to specify additional shared listeners and set the levels at which they will report diagnostics information. When you upload a new configuration for a role to an Azure application, it detects the configuration change and (by default) restarts the role. However, you can vary this default behavior by editing the code in the **RoleEnvironment.Changing** and **RoleEnvironment.Changed** event handlers. For more information, see “Handling Configuration Changes” at on MSDN (<http://msdn.microsoft.com/en-us/library/ee848064.aspx>).

Using the Diagnostics API in a Windows Azure Application

In addition to being able to configure diagnostics for a deployed application using a configuration file and the tools described in the previous sections of this guidance, you can also access the diagnostics API using code within your application to configure and start diagnostic monitoring and write debugging and trace information. In your code, you can accomplish the following tasks:

- Specify any changes required from the default configuration of the Diagnostics Monitor.
- Optionally configure scheduled uploads of the diagnostic data to table or BLOB storage.
- Start the Diagnostics Monitor.
- Generate the any debugging and trace messages you require. These will be output to the local buffer ready for upload to storage, along with any other data sources that you enable.

Configuring Diagnostics Using the API

This section briefly describes the options available for configuring the Diagnostics Monitor using code. The basic approach is to obtain a reference to the initial default configuration for the Diagnostics Monitor and change it as required. Typically, you will locate this code in the **OnStart** method of your role. After you configure your monitoring requirements, you must call the **Start** method of the Diagnostics Monitor, as shown in the section, “Activating Monitoring,” later in this appendix.

An instance of the **DiagnosticMonitorConfiguration** class holds the configuration information for a Diagnostics Monitor, exposing it through a series of properties such as **DiagnosticInfrastructureLogs**, **Directories**, **Logs**, **PerformanceCounters**, and **WindowsEventLog**. The **Directories**, **PerformanceCounters**, and **WindowsEventLog** properties expose a **DataSources** collection that specifies the items to monitor. Each data source you add can be configured using properties specific to the type and common properties that set the buffer quota and transfer interval.

The following example updates the initial default configuration by setting the polling interval (the interval at which the Diagnostics Monitor will check for configuration changes which is, by default, every minute), and the buffer quota for diagnostic logs. Then it adds to the configuration a performance counter and two Windows Event logs.

```
C#
// Obtain a reference to the initial default configuration.
DiagnosticMonitorConfiguration config
    = DiagnosticMonitor.GetDefaultInitialConfiguration();

// Change the polling interval and buffer quota for logs.
config.ConfigurationChangePollInterval
    = TimeSpan.FromSeconds(30.0);
config.DiagnosticInfrastructureLogs.BufferQuotaInMB = 256;

// Set the transfer interval for all logs.
config.Logs.ScheduledTransferPeriod = TimeSpan.FromMinutes(1.0);

// Configure monitoring of a Windows performance counter
// and add it to the configuration.
PerformanceCounterConfiguration perfConfig
    = new PerformanceCounterConfiguration();
perfConfig.CounterSpecifier = @"\Processor(*)\% Processor Time";
perfConfig.SampleRate = TimeSpan.FromSeconds(15.0);
config.PerformanceCounters.DataSources.Add(perfConfig);

// Configure monitoring of Windows Application and System Event
// logs,
// including the quota and scheduled transfer interval, and add
// them
// to the configuration.
WindowsEventLogsBufferConfiguration eventsConfig
    = new WindowsEventLogsBufferConfiguration();
```

```
eventsConfig.BufferQuotaInMB = 256;
eventsConfig.ScheduledTransferLogLevelFilter = LogLevel.Warning;
eventsConfig.ScheduledTransferPeriod = TimeSpan.FromMinutes(10.0);
eventsConfig.DataSources.Add("Application!*");
eventsConfig.DataSources.Add("System!*");
config.WindowsEventLog = eventsConfig;
// Now remember to start the diagnostics monitor with this
// configuration.
// For more information, see the section "Activating Monitoring."
```

For more information about the settings available for the **DiagnosticMonitorConfiguration** class, see “DiagnosticMonitorConfiguration Members” on MSDN (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.diagnostics.diagnosticmonitorconfiguration_members.aspx).

To obtain a list of counter names, execute the command **TypePerf/q** in a Command window.

When adding a Windows Event log to the configuration, you specify it using an XPath statement of the form *channel!xpath-query*. In the preceding example, **Application!*** specifies the Windows Application event log. For more information about XPath expressions that identify events, see “Consuming Events” on MSDN ([http://msdn.microsoft.com/en-us/library/dd996910\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/dd996910(VS.85).aspx)).

If you want to capture the contents of a crash dump, you call one of the methods of the **CrashDumps** class once before starting monitoring. The simplest overload takes a Boolean parameter that specifies if you want the full dump (**true**) or a mini-dump (**false**).

```
C#
CrashDumps.EnableCollection(false);
```

You can optionally specify the directory to store the crash dump using the overload that takes a string containing the full absolute path to the required directory.

Note: *Web role requests that encounter an error and cause the role to fail are usually caught by ASP.NET, which prevents crash dump generation.*

Activating Monitoring

This section briefly describes how you activate monitoring using the diagnostics API. The **Start** method of the **DiagnosticMonitor** class starts the Diagnostic Monitor using a configuration that you specify. The **Shutdown** method stops the Diagnostic Monitor. The following code starts the Diagnostic Monitor using the configuration instance created in the previous section.

```
C#
// Start the Diagnostics Monitor.
DiagnosticMonitor.Start("DiagnosticsConnectionString", config);
```

The first parameter is the name of a setting in the ServiceConfiguration.cscfg file that specifies the storage location for uploaded diagnostics data.

Alternatively, you can create an instance of the **CloudStorageAccount** class, set the properties (such as the storage endpoints and credentials), and then pass this as the first parameter of the **Start** method. If you do not specify a configuration instance when you call the **Start** method, the Diagnostics Monitor will use the default configuration. Ensure that you use the correct connection string, because an incorrect string may cause monitoring to fail without causing a run-time error.

Avoid calling the **Start** method of the Diagnostic Monitor more than once. Changes you make to the configuration are automatically detected at intervals determined by the value of the **ConfigurationChangePollInterval** property.

Generating Diagnostics Data Within the Application

This section briefly describes how you can write information to the Diagnostics Monitor using the diagnostics API. Typically, you would add code to generate trace messages containing debugging information at any point in the code for your role where you need to generate information to assist in debugging or monitoring the application.

The following line writes a warning message to the Diagnostics Monitor to indicate that monitoring has been configured and started.

```
C#
// Write a trace message to the Diagnostic Monitor.
string roleInstance = RoleEnvironment.CurrentRoleInstance.Id;
System.Diagnostics.Trace.TraceWarning(
    "Diagnostics configured and started " +
    "for role instance: " + roleInstance);
```

You can use the other methods of the **System.Diagnostics.Trace** class to write information in the exact format you require. For more information about these methods and the other members of the class, see “Trace Members” on MSDN (http://msdn.microsoft.com/en-us/library/system.diagnostics.trace_members.aspx).

In addition to generating debugging and trace log entries, one typical requirement for generating diagnostic data within an application is to support data sources that are not directly available in the Windows Azure diagnostics framework. An example of this is Windows Management Instrumentation (WMI) events, which are not

available through the diagnostics API. Instead, you may consider capturing WMI events in your code and then generating log entries for an arbitrary log. You configure arbitrary logs using the **DataSources** collection of the **Logs** property of the configuration instance.

Transferring Diagnostics Data to Storage

This section briefly describes how you can transfer diagnostics data from the local buffer to table and BLOB storage using the diagnostics API. If you have configured scheduled upload transfers using the diagnostics configuration API or a remote tool, you do not need to perform any additional tasks. However, you can execute an on-demand upload of the monitoring data in the local directory buffer to BLOB and table storage, irrespective of whether you have configured scheduled uploads.

When you initiate a transfer of data to storage, consider filtering it on the verbosity to minimize storage requirements and transaction costs. For example, transfer only warning and critical event data.

The following code example shows a method that you can call to initiate an on-demand transfer of diagnostics data to storage. It contains hard-coded specifications of values, such as the target storage account and the transfer options. However, you can adapt the method code to accept any of the values as parameters if you want to create a more generic version.

The first stage is to create a Deployment Diagnostic Manager for the target storage account. You can specify the account using the name and key or as a connection string in your ServiceConfiguration.cscfg file.

```
C#
public Guid StartOnDemandTransfer()
{
    // Create a CloudStorageAccount for a specified name and key.
    StorageCredentialsAccountAndKey accountAndKey =
        new StorageCredentialsAccountAndKey(
            "account-name",
            "account-key");
    CloudStorageAccount storageAccount
        = new CloudStorageAccount(
            accountAndKey,
            false);

    // Specify the deployment ID of the target storage account.
    string deploymentId = "target-storage-account-deployment-id";
```

```
// Create a Deployment Diagnostic Manager for this target
// storage account.
DeploymentDiagnosticManager ddm =
    new DeploymentDiagnosticManager(
        storageAccount,
        deploymentId);
...

```

Alternatively, you can use the name of a storage account connection setting defined in the ServiceConfiguration.cscfg file.

```
C#
...
DeploymentDiagnosticManager ddm =
    new DeploymentDiagnosticManager(
        "StorageAccountConnectionString",
        deploymentId);
...

```

For more information about the Deployment Diagnostic Manager, see “DeploymentDiagnosticManager Members” on MSDN (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.diagnostics.management.deploymentdiagnosticmanager_members.aspx).

Next, you create a Role Instance Diagnostic Manager for the role containing the diagnostics data you want to transfer. This example assumes you want to transfer data from the current role in which the code is executing. You can, if required, specify the ID and name of a different role.

```
C#
...
string roleId = RoleEnvironment.CurrentRoleInstance.Id;
string roleName =
    RoleEnvironment.CurrentRoleInstance.Role.Name;
RoleInstanceDiagnosticManager rdm =
    ddm.GetRoleInstanceDiagnosticManager(
        roleName,
        roleId);
...

```

Another approach is to iterate through the roles and instances in your application using the methods of the Role Instance Diagnostic Manager. The following shows two examples of iterating through all the available instances for all roles in order to demonstrate some of the methods available to you.

```
C#
// Alternative approaches for accessing roles and instances.
foreach (string roleName in ddm.GetRoleNames())
{
    foreach (string instanceID in ddm.GetRoleInstanceIdsForRole(
        roleName))
    {
        RoleInstanceDiagnosticManager rdm
            = ddm.GetRoleInstanceDiagnosticManager(
                roleName,
                instanceID);
        // Use the Role Instance Diagnostic Manager here ...
    }
}

foreach (string roleName in ddm.GetRoleNames())
{
    IEnumerable<RoleInstanceDiagnosticManager> rdmList
        = ddm.GetRoleInstanceDiagnosticManagersForRole(roleName);
    foreach (RoleInstanceDiagnosticManager rdm in rdmList)
    {
        // Use the Role Instance Diagnostic Manager here ...
    }
}
```

After you obtain a Role Instance Diagnostic Manager, you specify the options for the transfer of the diagnostics data from the buffer to storage using an instance of the **OnDemandTransferOptions** class, as shown here.

```
C#
...
// Create a configuration options instance for the transfer
// and set the required property values.
OnDemandTransferOptions options =
    new OnDemandTransferOptions();
options.NotificationQueueName = "transfer-queue-name";
options.From = new DateTime(2010, 6, 10, 12, 0, 0);
options.To = DateTime.UtcNow;
options.LogLevelFilter = LogLevel.Critical;
...
```

The **NotificationQueueName** is the name of a queue to which the Role Instance Diagnostic Manager will send a notification that the transfer has completed. If you are not interested in receiving a notification, you can ignore this property.

Finally, you start the transfer by calling the **BeginOnDemandTransfer** method of the Role Instance Diagnostic Manager, specifying the type of data source for which you want to transfer data and the options instance you created. The method returns a GUID that identifies the notification message that will be sent to the queue you specified as the value of the **NotificationQueueName** property.

```
C#  
...  
// Start the transfer and return the GUID that identifies it.  
return rdm.BeginOnDemandTransfer(  
    DataBufferName.WindowsEventLogs,  
    options);  
}
```

Other methods of the Role Instance Diagnostic Manager allow you to cancel a specific transfer, cancel all transfers, iterate over the list of active transfers, and modify the configuration settings for the transfer. For more information, see “RoleInstanceDiagnosticManager Members” on MSDN (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.diagnostics.management.roleinstancediagnosticmanager_members.aspx).

RESOLVING PERMISSION ISSUES IN PARTIAL TRUST MODE

By default, code within all Windows Azure application roles runs in full-trust mode that grants your application permissions to perform operations such as invoking non-.NET code, using .NET Framework libraries that require full trust or inter-process communication using named pipes. However, even in full-trust mode, Windows Azure applications cannot perform some operations, such as modifying the registry or writing to the system directory.

You can configure code within Windows Azure application roles to run in partial-trust mode that is approximately equivalent to ASP.NET medium-trust level. This means that some operations your code may execute, such as attempting to use sockets for network access, will cause a run-time error because of insufficient permissions. For a full list of permission settings, see “Windows Azure Partial Trust Policy Reference” on MSDN ([http://msdn.microsoft.com/en-us/library/dd573355\(v=MSDN.10\).aspx](http://msdn.microsoft.com/en-us/library/dd573355(v=MSDN.10).aspx)).

To enable partial-trust mode, add the **enableNativeCodeExecution="false"** attribute to the role configuration in the Service Definition file, as shown here.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceDefinition name="MyService"
  xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/
  ServiceDefinition">
  <WebRole name="MyWebRole" enableNativeCodeExecution="false">
    <InputEndpoints>
      ...
    </InputEndpoints>
  </WebRole>
  ...
</ServiceDefinition>
```

You can also enable partial-trust mode through the configuration of the role in your cloud project.

MORE INFORMATION

For more information about debugging and troubleshooting Windows Azure applications, see the following resources:

- “Windows Azure Troubleshooting Guide” on MSDN:
[http://msdn.microsoft.com/en-us/library/ff431736\(v=MSDN.10\).aspx](http://msdn.microsoft.com/en-us/library/ff431736(v=MSDN.10).aspx)
- “Implementing Windows Azure Diagnostics” on MSDN:
[http://msdn.microsoft.com/en-us/library/ee758705\(v=MSDN.10\).aspx](http://msdn.microsoft.com/en-us/library/ee758705(v=MSDN.10).aspx)
- “Exercise 3 - Monitoring Applications in Windows Azure” on Channel 9:
<http://channel9.msdn.com/learn/courses/Azure/Deployment/DeployingApplicationsinWindowsAzure/Exercise-3-Monitoring-Applications-in-Windows-Azure/>
- “Take Control of Logging and Tracing in Windows Azure” in *MSDN Magazine*:
<http://msdn.microsoft.com/en-gb/magazine/ff714589.aspx>

Index

A

Access Control Services (ACS), 26
acknowledgments, xix
Analyze method, 104-105
AppendSurveyAnswerIdToAnswerList
method, 97
applications
 customizing by tenant, 13-14
 life cycle management, 11
 managing costs, 17-18
 partitioning, 41-45
AppRoutes class, 43-44
ASP.NET
 routing, 73
 security, 40
 sites, 40
audience, xiv
AuthenticateAndAuthorizeAttribute
class, 30-31
AuthenticateAndAuthorizeAttribute
filter, 33-34
AuthenticateUser method, 30-31
authentication and authorization, 11,
25-35, 46
 for basic subscribers, 47
 for individual subscribers, 48
AzureTable class, 82-84

B

BatchProcessingQueueHandler class,
69-71
BatchProcessingQueueHandlerFixture
class, 72-73

Bharath *see* cloud specialist role
“Bharath”
BLOBs
 containing survey response data,
 55
 setting access control for contain-
 ers, 36-37
 temporary, 92-94
BLOB storage, 95
 writing directly to, 89-91
BrowseResponses method, 97-98

C

caching policy, 39
claims-based authentication and
 authorization model, 39
CloudBlobContainer class, 36
cloud specialist role “Bharath”, xvii
CNAME entries in DNS, 39
code base maintenance, 11-12
ContainerBootstrapper class, 85
Content Delivery Network (CDN),
35-40
 configuration, 37
continuation tokens, 110
cost comparison for alternative
 approaches to calculating summary
 statistics, 62
customer billing, 16-17, 48-49

D

data

- display, 94-105
- protecting from other tenants, 15-18
- storage in the Surveys application, 76

data architecture

- extensibility, 15
- scalability, 15-16

database information, 48

data in the Surveys application, 75-110

- AppendSurveyAnswerIdToAnswerList method, 97
- AzureTable class, 82-84
- BrowseResponses method, 97-98
- comparing solutions, 91-92, 95
- ContainerBootstrapper class, 85
- continuation tokens, 110
- cost, 100
- creating a survey with two questions, 99
- data model for a multi-tenant application, 75-81
- data storage in the Surveys application, 76
- Delayed Write pattern, 88-89
- displaying data, 94-105
- displaying questions, 103-104
- displaying the summary statistics, 104-105
- Display method, 103-104
- EditorExtensions class, 104
- IAzureTable interface, 82-84
- implementing the paging, 97-98
- maintaining the ordered list of survey responses, 96-97
- paging through survey results, 94
- paging with BLOB storage, 95
- paging with table storage, 94-95
- performance, 101
- Questions table, 77
- robustness, 101
- SaveSurveyAnswer method, 93-94
- saving survey response data, 87-94
- saving the survey response data to a temporary BLOB, 92-94
- scalability, 101

security, 101

- session data storage, 98, 101
- simplicity, 100
- SQL Azure, 105-109
- SQL Azure Considerations Guide, 110
- storage services, 110
- store classes, 80-81
- storing survey answers, 78-79
- storing survey answer summaries, 79-80
- storing tenant data, 77-78
- SurveyAnswersSummaryStore class, 80
- SurveyAnswerStore class, 80, 96
- SurveysController class, 92-93, 102-103
- SurveySqlStore class, 81, 109
- Surveys table structure in SQL Azure, 107
- SurveyStore class, 80, 84-85
- SurveyStore constructor, 86
- SurveyTransferStore class, 81
- TenantStore class, 81
- testing and Windows Azure storage, 81-86
- TransferSurveysToSqlAzureCommand class, 107-109
- writing directly to BLOB storage, 89-91

debugging and troubleshooting

- Windows Azure applications, 121-141
 - activate monitoring, 135-136
 - common setup and deployment errors with Windows Azure, 123-124
 - configuring diagnostics with files, 132-133
 - custom tracing, and debugging agent, 129-130
 - dependencies for Azure applications and projects, 121
 - DiagnosticMonitorConfigurationMembers, 135
 - Diagnostic Monitor Trace Listener, 122
 - Diagnostic Monitor type, 132
 - diagnostics, 126-129

- diagnostics API, 133-140
 - diagnostic tools, 130-131
 - generating diagnostics data
 - within the application, 136-137
 - Implementing Windows Azure Diagnostics, 141
 - IntelliTrace, 124-126
 - NotificationQueueName property, 140
 - OnDemandTransferOptions class, 139
 - resolving permission issues in
 - partial trust mode, 140-141
 - Take Control of Logging and Tracing in Windows Azure, 141
 - transferring diagnostics data to storage, 137-140
 - UseDevelopmentStorage=true, 132
 - Windows Azure Troubleshooting Guide, 141
- E**
- Delayed Write pattern, 88-89
 - diagnostic data, 136-137
 - DiagnosticMonitorConfigurationMembers, 135
 - Diagnostic Monitor Trace Listener, 122
 - Diagnostic Monitor type, 132
 - diagnostics API, 133-140
 - diagnostics data, 137-140
 - diagnostic tools, 130-131
 - Display method, 43, 103-104
 - DNS CNAME, 13-14, 39
 - DNS names, certificates, and SSL, 19-23
 - domains, 115-116
- F**
- fault domains, 115
 - FederationResult method, 32-33
 - FederationSecurityTokenService class, 31-32
 - federation with multiple partners
 - sequence, 29
 - financial considerations, 15-18
 - fluent APIs, 73
 - ForCreateHandlerForGivenQueue unit test, 73
 - foreward, xi
- G**
- GenericQueueHandler class, 71
 - geo location, 23-25
 - information, 48
- H**
- handling multiple background task types, 53
 - Html.EditorFor element, 105
 - https://tailspin.cloudapp.net, 21
 - http://tailspin.cloudapp.net, 21
- I**
- IAzureTable interface, 82-84
 - Implementing Windows Azure Diagnostics, 141
 - IntelliTrace, 124-126
 - IT professional role “Poe”, xvii
- J**
- Jana *see* software architect role “Jana”
- K**
- key plumbing types, 68
- L**
- legal issues, 11
- M**
- MapReduce algorithm, 54-59, 74
 - Markus *see* senior software developer role “Markus”
 - Microsoft, 1
 - monitoring, 12, 135-136
 - multiple background task types, 53
 - multi-tenancy architecture in Azure, 8-9
 - multi-tenant applications, 3
 - multi-tenant applications hosted on

- Windows Azure, 7-18
 - application life cycle management, 11
 - authentication and authorization, 11
 - billing customers, 16-17
 - code base maintenance, 11-12
 - customizing the application by tenant, 13-14
 - data architecture extensibility, 15
 - data architecture scalability, 15-16
 - DNS CNAME, 13-14
 - financial considerations, 15-18
 - legal issues, 11
 - managing application costs, 17-18
 - monitoring, 12
 - multi-tenant data architecture, 14-18
 - .NET providers and third-party components, 12-13
 - new clients or free trials, 13
 - protecting data from other tenants, 15-18
 - scalability, 10
 - Service Level Agreements (SLA), 10
 - stability, 10
 - upgrades, 12
 - URLs to access the application, 13
 - multi-tenant applications scalability, 41-74
 - authentication and authorization, 46
 - basic subscription information, 45-46
 - BatchProcessingQueueHandler class, 69-71
 - cost comparison for alternative approaches to calculating summary statistics, 62
 - customer billing, 48-49
 - customizing the user interface, 50
 - database information, 48
 - example BLOBs containing survey response data, 55
 - example scenarios for worker roles, 51-52
 - execution model, 53-54
 - geo location information, 48
 - handling multiple background task types, 53
 - key plumbing types, 68
 - MapReduce algorithm, 54-59
 - multiple background task types, 53
 - on-boarding for trials and new customers, 45-48
 - partitioning the application, 41-45
 - ProcessMessages method, 71
 - provisioning a trust relationship with the subscriber's identity provider, 46-47
 - provisioning authentication and authorization for basic subscribers, 47
 - provisioning authentication and authorization for individual subscribers, 48
 - references and resources, 73-74
 - scaling applications by using worker roles, 50-59
 - scaling the surveys application, 60-73
 - slug name, 42
 - Task.Factory.StartNew method, 70
 - testing the worker role, 72-73
 - triggers for background tasks, 52
 - using a worker role to calculate the summary statistics, 63-67
 - worker role instances, 54
 - worker role "plumbing" code, 67-71
 - multi-tenant data architecture, 14-18
- N**
- .NET providers and third-party components, 12-13
 - new clients or free trials, 13
 - NotificationQueueName property, 140
- O**
- on-boarding for trials and new customers, 45-48
 - OnDemandTransferOptions class, 139
 - options for updating a deployed Windows Azure service, 110-119

- design considerations for
 - maximizing upgradability, 113-115
- fault domains, 115
- limitations when updating a Windows Azure service, 112-113
- managing scaling, 119
- options for updating a deployed Windows Azure service, 117-118
- redeploying a service without incurring downtime, 113
- service upgrades using the Service Management API, 119
- updating only the configuration of a service, 113
- upgrade domains, 115-116
- using upgrade domains for in-place upgrades, 116-117

P

- paging implementation, 97-98
- performance, 101
- permission issues in partial trust mode, 140-141
- Poe *see* IT professional role “Poe”
- preface, xiii-xvii
- prerequisites, xvi
- ProcessMessages method, 71

Q

- QuestionAnswer class, 78-80
- questions display, 103-104
- Questions table, 77
- QuestionTemplateFactory class, 104

R

- references and resources, 73-74
- robustness, 101
- roles, xvii

S

- SaveSurveyAnswer method, 93-94
- scalability, 10, 101
- scaling, managing, 119
- scenarios *see* Tailspin scenario
- security, 101

- senior software developer role
 - “Markus”, xvii
- ServiceDefinition.csdef file, 22
- Service Level Agreements (SLA), 10
- service upgrades using the Service Management API, 119
- session data storage, 98, 101
- session tokens, protecting in Windows Azure, 34-35
- SetPermissions method, 36
- simplicity, 100
- single-tenant vs. multi-tenant applications, 7-18
- slug name, 42, 76
- software architect role “Jana”, xvii
- SQL Azure, 105-109
- SQL Azure Considerations Guide, 110
- stability, 10
- storage
 - services, 110
 - Windows Azure storage, 81-86
- store classes, 80-81
- subscription information, 45-46
- summary statistics, 104-105
- SurveyAnswer class, 78-80
- survey answers, 78-79
- SurveyAnswersSummaryStore class, 80
- SurveyAnswerStore class, 80, 96-97
- survey answer summaries, 79-80
- SurveyAreaRegistration class, 44
- survey response data, 87-94
- survey responses, 96-97
- survey results, 94
- Surveys application, 60-73
- Surveys application access, 19-40
 - Access Control Services (ACS), 26
 - authentication and authorization, 25-35
 - caching policy, 39
 - CDN configuration, 37
 - Content Delivery Network (CDN), 35-39
 - DNS names, certificates, and SSL, 19-23
 - federation with multiple partners
 - sequence, 29
 - further information, 39-40
 - geo location, 23-25
 - goals and requirements, 20

- https://tailspin.cloudapp.net, 21
- http://tailspin.cloudapp.net, 21
- inside the implementation, 21-23
- overview of the solution, 20-23
- protecting session tokens in
 - Windows Azure, 34-35
- setting the access control for the
 - BLOB containers, 36-37
- URL configuration, 37-38
- web roles in the surveys
 - application, 19-23
- SurveysController class, 92-93, 102-103
- SurveySqlStore class, 81, 109
- SurveyStore class, 84-86
- SurveyStore constructor, 86
- SurveyTransferStore class, 81
- survey with two questions, 99

T

- table storage, 94-95
- Tailspin scenario, 1-6
 - goals and concerns, 3-4
 - Surveys application, 2-3
 - Surveys application architecture, 5-6
- Take Control of Logging and Tracing in
 - Windows Azure, 141
- Task.Factory.StartNew method, 70
- Task Parallel Library, 74
- Tenant class, 77-78
- tenant data, 77-78
- tenants, 3
- TenantStore class, 81
- TransferSurveysToSqlAzureCommand
 - class, 107-109
- trust relationship with the subscriber's
 - identity provider, 46-47

U

- UpdatingSurveyResultsSummaryCommand class, 63-67
- upgrade domains, 115-116
 - for in-place upgrades, 116-117
- upgrades, 12
- URL Rewrite Module for IIS, 73
- URLs
 - to access the application, 13
 - configuration, 37-38
- UseDevelopmentStorage=true, 132
- user interface, customizing, 50

W

- web roles in the surveys application, 19-23
- Windows Azure storage, 81-86
- Windows Azure Troubleshooting
 - Guide, 141
- worker role, 72-73
 - to calculate the summary
 - statistics, 63-67
 - instances, 54
 - "plumbing" code, 67-71