Moving Applications to the Cloud, 2nd Edition

patterns & practices

Guide

Microsoft

Moving Applications to the Cloud, 2nd Edition

patterns & practices

Summary: This book demonstrates how you can adapt an existing, on-premises ASP.NET application to one that operates in the cloud. The book is intended for any architect, developer, or information technology (IT) professional who designs, builds, or operates applications and services that are appropriate for 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, ASP.NET, and Microsoft Visual C#.

Category: Guide
Applies to: Windows Azure SDK for .NET (includes the Visual Studio Tools for Windows Azure), Windows Azure SQL Database, Microsoft SQL Server or SQL Server Express 2008, Windows Identity Foundation, Enterprise Library 5, WatiN 2.0, Microsoft Anti-Cross Site Scripting Library V4, Microsoft .NET Framework version 4.0, Microsoft Visual Studio 2010
Source: MSDN Library(patterns & practices) (link to source content)
E-book publication date:June 2012

Contents

Foreword – Yousef Khalidi

The Windows Azure platform, an operating environment for developing, hosting, and managing cloud-based services, establishes a foundation that allows customers to easily move their applications from on-premises locations to the cloud. With Windows Azure, customers benefit from increased agility, a very scalable platform, and reduced costs. The Microsoft cloud strategy has three broad tenets:

- Flexibility of choice, based on business needs, for deploying services
- Enterprise-class services with no compromises on availability, reliability, or security
- Consistent, connected experiences across devices and platforms

Windows Azure is a key component of the Microsoft cloud strategy.

Windows Azure builds on the many years of experience Microsoft has running online services for millions of users and on our long history of building platforms for developers. We focused on making the transition from on-premises to the cloud easy for both programmers and IT professionals.Their existing skills and experience are exactly what they need to start using the Windows Azure platform.

Microsoft is committed to Windows Azure, and will continue to expand it as we learn how all our customers around the globe, from the largest enterprises to the smallest ISVs, use it. One of the advantages of an online platform is that it allows us to introduce innovations quickly.

I'm excited to introduce this guide from the Microsoft patterns & practices team, proof of our commitment to help customers be successful with the Windows Azure platform. Whether you're new to Windows Azure, or if you're already using it, you'll find this guide a great source of ideasto consider. I encourage you to get started exploring the Microsoft public cloud and to stay tuned for further guidance from the patterns & practices team.

Sincerely,

Yousef Khalidi

Distinguished Engineer, Windows Azure

Foreword – Amitabh Srivastava

Millions of people are using cloud services from Microsoft; as a company, we're all in! And as someone who has been involved with Windows Azure since the beginning, it's gratifying to see this work come to fruition. For customers still exploring what the cloud means for them, this guide from the Microsoft patterns & practices team will answer many of the questions they may have. Microsoft is serious about cloud computing, and this guide is one of many investments that Microsoft is making to ensure that its customers are successful as they begin developing new applications or migrating existing applications to the cloud.

Developers familiar with .NET and the rest of the Microsoft platform will be able to use their existing skills to quickly build or move existing applications to the cloud and use the power of the cloud to scale to millions of users and reach everyone in the world. Yet, Windows Azure is an open platform that works well with other technology stacks and application frameworks, providing customers with the choice and flexibility to move as much or as little business to the cloud as they want and without needing to start over.

This guide is a great starting point for those who want to embark on this journey using a pragmatic, scenario-based approach.

Sincerely,

Amitabh Srivastava

Senior Vice President, Windows Azure

Preface

How can a company's applications be scalable and have high availability? To achieve this, along with developing the applications, you must also have an infrastructure that can support them. For example, you may need to add servers or increase the capacities of existing ones, have redundant hardware, add logic to the application to handle distributed computing, and add logic for failovers. You have to do this even if an application is in high demand for only short periods of time. Life becomes even more complicated (and expensive) when you start to consider issues such as network latency and security boundaries.

The cloud offers a solution to this dilemma. The cloud is made up of interconnected servers located in various data centers. However, you see what appears to be a centralized location that someone else hosts and manages. By shifting the responsibility of maintaining an infrastructure to someone else, you're free to concentrate on what matters most: the application. If the cloud has data centers in different geographical areas, you can move your content closer to the people who are using it most. If an application is heavily used in Asia, have an instance running in a data center located there. This kind of flexibility may not be available to you if you have to own all the hardware. Another advantage to the cloud is that it's a "pay as you go" proposition. If you don't need it, you don't have to pay for it. When demand is high, you can scale up, and when demand is low, you can scale back.

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. This guide shows you how to do this.

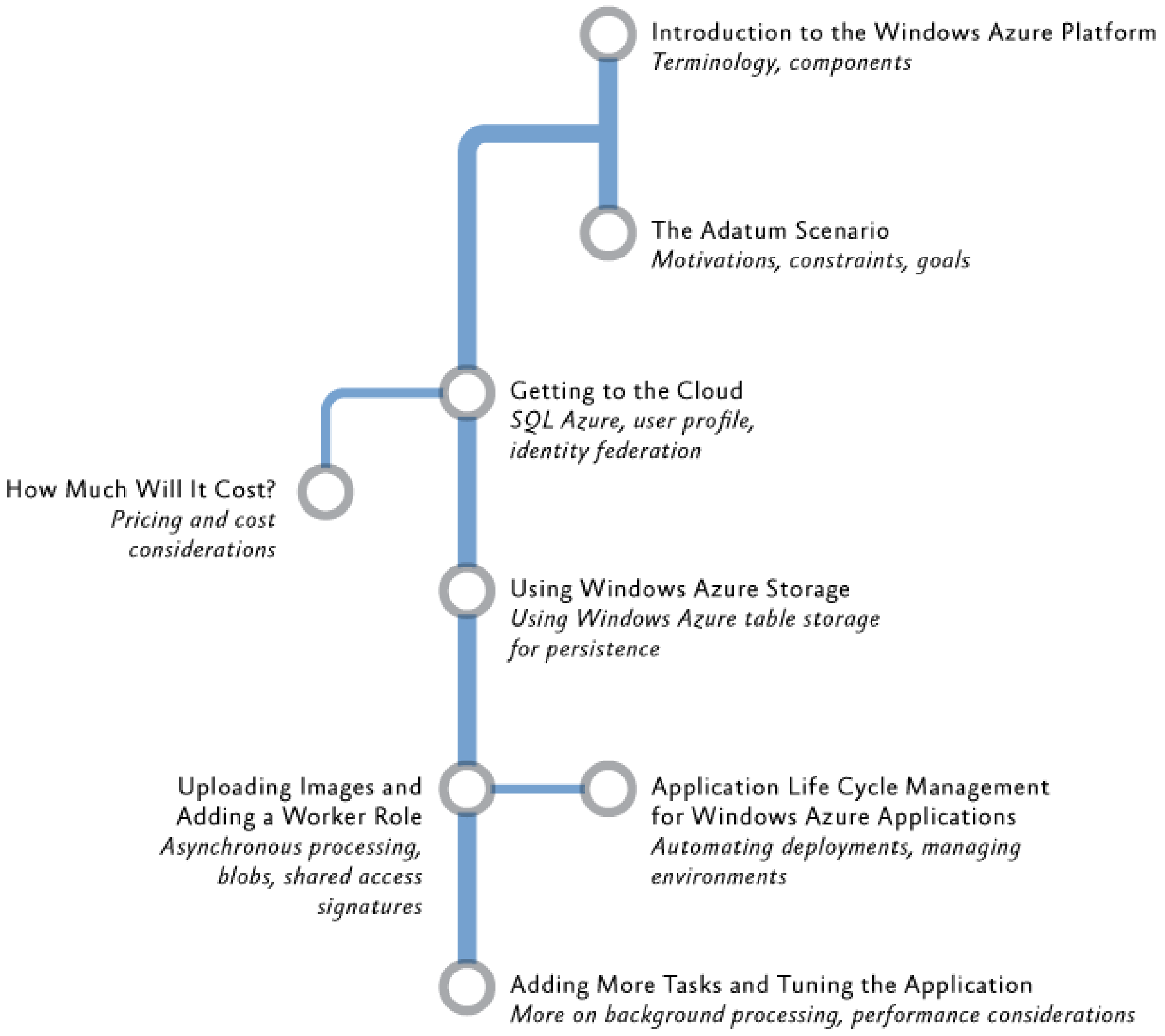
Who This Book Is For

This book is the first volume in a series about Windows® Azure™. It demonstrates how you can adapt an existing, on-premises ASP.NET application to one that operates in the cloud. The book is intended for any architect, developer, or information technology (IT) professional who designs, builds, or operates applications and services that are appropriate for 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®, ASP.NET, and Microsoft Visual C#®.

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. In addition, the Windows Azure SDK includes the *Compute Emulator* and the *Storage Emulator*. Developers can use these to write, test, and debug their applications locally 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 adapt an existing ASP.NET application and deploy it to Windows Azure.

How This Book Is Structured


Introduction to the Windows Azure Platform
Terminology, components
The Adatum Scenario
Motivations, constraints, goals
Getting to the Cloud
SQL Azure, user profile, identity federation
How Much Will It Cost?
Pricing and cost considerations
Using Windows Azure Storage
Using Windows Azure table storage for persistence
Uploading Images and Adding a Worker Role
Asynchronous processing, blobs, shared access signatures
Application Life Cycle Management for Windows Azure Applications
Automating deployments, managing environments
Adding More Tasks and Tuning the Application
More on background processing, performance considerations

"Introduction to the Windows Azure Platform" provides an overview of the platform to get you startedwith Windows Azure. It lists and provides links to resources about the features of Windows Azure such as web roles and worker roles; the services you can use such as Access Control and Caching; the different ways you can store data in Windows Azure; the development tools and practices for building Windows Azure applications; and the Windows Azure billing model. It's probably a good idea that you read this before you go to the scenarios.

"The Adatum Scenario" introduces you to the Adatumcompany and the aExpense application. The following chapters describe how Adatummigrates the aExpense application to the cloud. Reading this chapter will help you understand why Adatum wants to migrate some of its business applications to the cloud, and it describes some of its concerns.

"Phase 1: Getting to the Cloud" describes the first steps that Adatum takes in migrating the aExpenseapplication. Adatum's goal here is simply to get the application working in the cloud, but this includes "big" issues, such as security and storage.

"How Much Will It Cost?" introduces a basic cost model for the aExpense application running on Windows Azure and calculates the estimated annual running costs for the application. This chapter is optional. You don't need to read it before you go on to the following scenarios.

"Phase 2: Automating Deployment and Using Windows Azure Storage" describes how Adatum uses PowerShell scripts and the Microsoft Build Engine (MSBuild) to automate deploying aExpense to Windows Azure. It also describes how Adatum switches from using Windows Azure SQL Database to Windows Azure table storage in the aExpense application and discusses the differences between the two storage models.

"Phase 3: Uploading Images and Adding a Worker Role" describes adding a worker role to the aExpenseapplication and shows how aExpense uses Windows Azure blob storage for storing scanned images.

"Application Life Cycle Management for Windows Azure Applications" discusses how to manage developing, testing, and deploying Windows Azure applications.This chapter is optional. You don't need to read it before you go on to the last scenario.

"Phase 4: Adding More Tasks and Tuning the Application" shows how Adatum adds more tasks to the worker role in the aExpense application. In this phase, Adatum also evaluates the results of performance testing the application and makes some changes based on the results.

What You Need to Use the Code

These are the system requirements for running the scenarios:

- Microsoft Windows Vista with Service Pack 2, Windows 7 with Service Pack 1, or Windows Server 2008 R2 with Service Pack 1 (32 bit or 64 bit editions)
- Microsoft .NET Framework version 4.0
- Microsoft Visual Studio 2010 Ultimate, Premium, or Professional edition with Service Pack 1 installed
- Windows Azure Tools for Visual Studio (includes the Windows Azure SDK)
- Microsoft SQL Server or SQL Server Express 2008
- Windows Identity Foundation. This is required for claims-based authorization.
- Enterprise Library 5 (required assemblies are included in the source code folders)
- WatiN 2.0. Open the Properties dialog and unblock the zip file after you download it and before you extract the contents. Place the contents in the Lib folder of the examples.
- Microsoft Anti-Cross Site Scripting Library V4. Place this in the Lib folder of the examples.

Who's Who

As mentioned earlier, this book uses a set of scenarios that demonstrates how to move applications to 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 scenarios can be considered from each of these points of view. The following table lists the experts for these scenarios.

	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. *"Moving a single application to the cloud is easy. Realizing the benefits that a cloud-based solution can offer 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. *"I don't care what platform you want to use for the application, I'll make it work."*
	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. *"Migrating to the cloud involves a big change in the way we manage our applications. I want to make sure our cloud apps are as reliable and secure as our on-premise apps."*

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

Acknowledgements

On March 4th, I saw an email from our CEO, Steve Ballmer, in my inbox. I don't normally receive much email from him, so I gave it my full attention. The subject line of the email was: "We are all in," and it summarized the commitment of Microsoft® to cloud computing. If I needed another confirmation of what I already knew, that Microsoft is serious about the cloud, there it was.

My first contact with what eventually became Windows® Azure™ was about three years ago. I was in the Developer & Platform Evangelism (DPE) team, and my job was to explore the world of software delivered as a service. Some of you might even remember a very early mockup I developed in late 2007, called Northwind Hosting. It demonstrated many of the capabilities that Windows Azure offers today. (Watching an initiative I've been involved with since the early days become a reality makes me very, very happy.)

In February 2009, I left DPE and joined the patterns & practices team. My mission was to lead the "cloud program": a collection of projects that examined the design challenges of building applications for the cloud. When Windows Azure was announced, demand for guidance about it skyrocketed.

As we examined different application development scenarios, it became quite clear that identity management is something you must get right before you can consider anything else. It's especially important if you are a company with a large portfolio of on-premises investments, and you want to move some of those assets to the cloud. This describes many of our customers.

In December 2009, we released *A Guide to Claims-Based identity and Access Control*. This was patterns &practices's first deliverable, and an important milestone, in our cloud program.

Windows Azure is special in many ways. One is the rate of innovation. The various teams that deliver all of the platform's systems proved that they could rapidly ship new functionality. To keep up with them, I felt we had to develop content very quickly. We decided to run our projects in two-months sprints, each one focused on a specific set of considerations.

This guide mainly covers a migration scenario: how to move an existing application to Windows Azure. As in the claims guide, we've developed a fictitious case study that explains, step by step, the challenges our customers are likely to encounter.

I want to start by thanking the following subject matter experts and contributors to this guide: Dominic Betts, Scott Densmore, Ryan Dunn, Steve Marx, and MatiasWoloski. Dominic has the unusual skill of knowing a subject in great detail and of finding a way to explain it to the rest of us that is precise, complete, and yet simple to understand. Scott brought us a wealth of knowledge about how to build scalable Windows Azure applications, which is what he did before he joined my team. He also brings years of experience about how to build frameworks and tools for developers. I've had the privilege of working with Ryan in previous projects, and I've always benefited from his acuity, insights, and experience. As a Windows Azure evangelist, he's been able to show us what customers with very real requirements need. Steve is a technical strategist for Windows Azure. He's been instrumental in shaping this guide. We rely on him to show us not just what the platform can do today but how it will evolve.

This is important because we want to provide guidance today that is aligned with longer-term goals. Last but not least, Matias is a veteran of many projects with me. He's been involved with Windows Azure since the very first day, and his efforts have been invaluable in creating this guide.

As it happens with all our written content, we have sample code for most of the chapters. They demonstrate what we talk about in the guide. Many thanks to the project's development and test teams for providing a good balance of technically sound, focused and simple-to-understand code: Masashi Narumoto, Scott Densmore, Federico Boerr (Southworks), AdriánMenegatti (Southworks), Hanz Zhang,RavindraMahendravarman (Infosys Ltd.), Rathi Velusamy (Infosys Ltd.).

Our guides must not only be technically accurate but also entertaining and interesting to read. This is no simple task, and I want to thank Dominic Betts, RoAnn Corbisier, Alex Homer, and Tina Burden from the writing and editing team for excelling at this.

The visual design concept used for this guide was originally developed by Roberta Leibovitz and Colin Campbell (Modeled Computation LLC) for *A Guide toClaims-Based Identity and Access Control.* Based on the excellent responses we received, we decided to reuse it for this book. The book design was created by John Hubbard (eson). The cartoon faces were drawn by the award-winning Seattle-based cartoonist Ellen Forney.The technical illustrations were adapted from my Tablet PC mockups by Rob Nanceand Katie Niemer.

All of our guides are reviewed, commented upon, scrutinized, and criticized by a large number of customers, partners, and colleagues. We also received feedback from the larger community through our CodePlex website. The Windows Azure platform is broad and spans many disciplines. We were very fortunate to have the intellectual power of a very diverse and skillful group of readers available to us.

I also want to thank all of these people who volunteered their time and expertise on our early content and drafts. Among them, I want to mention the exceptional contributions of David Aiken, Graham Astor (Avanade), Edward Bakker (Inter Access), VivekBhatnagar, Patrick Butler Monterde (Microsoft), Shy Cohen, James Conard, Brian Davis (Longscale), AashishDhamdhere (Windows Azure, Microsoft), Andreas Erben (DAENET), Giles Frith , Eric L. Golpe (Microsoft), Johnny Halife (Southworks), Alex Homer, Simon Ince, Joshy Joseph, Andrew Kimball, MilindaKotelawele (Longscale), Mark Kottke (Microsoft), Chris Lowndes (Avanade), Dianne O'Brien (Windows Azure, Microsoft), Steffen Vorein (Avanade), Michael Wood (Strategic Data Systems).

I hope you find this guide useful!

Eugenio Pace

Eugenio Pace
Senior Program Manager – *patterns & practices*
Microsoft Corporation
Redmond

1 – Introduction to the Windows Azure Platform

This chapter provides a brief description of the Microsoft® Windows® Azure™ technology platform, the services it provides, and the opportunities it offers for on-demand, cloud-based computing; where the *cloud* is a set of interconnected computing resources located in one or more data centers. The chapter also provides links to help you find more information about the features of Windows Azure, the techniques and technologies used in this series of guides, and the sample code that accompanies them.

About Windows Azure

Developers can use the cloud to deploy and run applications and to store data. On-premises applications can still use cloud–based resources. For example, an application located on an on-premises server, a rich client that runs on a desktop computer, or one that runs on a mobile device can use storage that is located on the cloud.

Windows Azure abstracts hardware resources through virtualization. Each application that is deployed to Windows Azure runs on one or more Virtual Machines (VMs). These deployed applications behave as though they were on a dedicated computer, although they might share physical resources such as disk space, network I/O, or CPU cores with other VMs on the same physical host. A key benefit of an abstraction layer above the physical hardware is portability and scalability. Virtualizing a service allows it to be moved to any number of physical hosts in the data center. By combining virtualization technologies, commodity hardware, multi-tenancy, and aggregation of demand, Microsoft can achieve economies of scale. These generate higher data center utilization (that is, more useful work-per-dollar hardware cost) and, subsequently, savings that are passed along to you.

Bharath says:

Windows Azure can help you achieve portability and scalability for your applications, and reduce your running costs and TCO.

Virtualization also allows you to have both *vertical scalability* and *horizontal scalability*. Vertical scalability means that, as demand increases, you can increase the number of resources, such as CPU cores or memory, on a specific VM. Horizontal scalability means that you can add more instances of VMs that are copies of existing services. All these instances are load balanced at the network level so that incoming requests are distributed among them.

At the time of this writing, the Windows Azure platform includes two main components: Windows Azure itself and Windows Azure SQL Database.

Windows Azure provides a Microsoft Windows® Server-based computing environment for applications and persistent storage for both structured and unstructured data, as well as asynchronous messaging. It also provides a range of services that help you to connect users and on-premises applications to cloud-hosted applications, manage authentication, and implement data management and related features such as caching.

Windows Azure SQL Databaseis essentially SQL Server® provided as a service in the cloud.

Windows Azure also includes a range of management services that allow you to control all these resources, either through a web-baseduser interface (a web portal) or programmatically. In most cases there is a REST-based API that can be used to define how your services will work. Most management tasks that can be performed through the web portal can also be achieved using the API.

Finally, there is a comprehensive set of tools and software development kits (SDKs) that allow you to develop, test, and deploy your applications. For example, you can develop and test your applications in a simulated local environment, provided by the Compute Emulator and the Storage Emulator. Most tools are also integrated into development environments such as Microsoft Visual Studio®. In addition, there are third-party management tools available.

Windows Azure Services and Features

The range of services and features available in Windows Azure and SQL Database target specific requirements for your applications. When you subscribe to Windows Azure you can choose which of the features you require, and you pay only for the features you use. You can add and remove features from your subscription whenever you wish. The billing mechanism for each service depends on the type of features the service provides. For more information on the billing model, see "*Windows Azure Subscription and Billing Model*" later in this chapter.

The services and features available change as Windows Azure continues to evolve. This series of guides, the accompanying sample code, and the associated Hands-on-Labs demonstrate many of the features and services available in Windows Azure and SQL Database. The followingfour sections of this chapter briefly describe the main services and features available at the time of writing, subdivided into the categories of Execution Environment, Data Management, Networking Services, and Other Services.

Bharath says:

Windows Azure includes a range of services that can simplify development, increase reliability, and make it easier to manage your cloud-hosted applications.

For more information about all of the Windows Azure services and features, see "*Windows Azure Features*" on the Windows Azure Portal at http://www.windowsazure.com/en-us/home/features/overview/. For specific development and usage guidance on each feature or service, see the resources referenced in the following sections.

To use any of these features and services you must have a subscription to Windows Azure. A valid Windows Live ID is required when signing up for a Windows Azure account. For more information, see http://www.windowsazure.com/en-us/pricing/purchase-options/.

Execution Environment

The Windows Azure execution environment consists of a runtime for applications and services hosted within one or more roles. The types of roles you can implement in Windows Azure are:

- **Azure Compute (web and worker roles)**. A Windows Azure application consists of one or more hosted roles running within the Azure data centers. Typically there will be at least one web role that is exposed for access by users of the application. The application may contain additional roles, including worker roles that are typically used to perform background processing and support tasks for Web roles. For more detailed information see "*Overview of Creating a Hosted Service for Windows Azure*" at http://technet.microsoft.com/en-au/library/gg432976.aspx and "*Building an Application that Runs in a Hosted Service*" at http://technet.microsoft.com/en-au/library/hh180152.aspx.
- **Virtual Machine (VM role)**. This role allows you to host your own custom instance of the Windows Server 2008 R2 Enterprise or Windows Server 2008 R2 Standard operating system within a Windows Azure data center.For more detailed information see "*Creating Applications by Using a VM Role in Windows Azure*" at http://technet.microsoft.com/en-au/library/gg465398.aspx.

Most of the examples in this guide and the associated guide "*Developing Applications for the Cloud*" (see http://wag.codeplex.com/), and the examples in the Hands-on-Labs, use a web role to perform the required processing.

The use of a worker role is also described and demonstrated in many places throughout the guides and examples. This includesChapter 6 of this guide and the associated sample application, Lab 4 in the Hands-on-Labs for this guide, Chapter 5 of the guide "*Developing Applications for the Cloud*" (see http://wag.codeplex.com/) and the associated sample application, and Lab 3 in the Hands-on-Labs for the guide "*Developing Applications for the Cloud*".

Data Management

Windows Azure, SQL Database, and the associated services provide opportunities for storing and managing data in a range of ways. The following data management services and features are available:

- **Azure Storage**. This provides four core services for persistent and durable data storage in the cloud. The services support a REST interface that can be accessed from within Azure-hosted or on-premises (remote) applications. For information about the REST API, see "*Windows Azure Storage Services REST API Reference*" at http://msdn.microsoft.com/en-us/library/dd179355.aspx. The four storage services are:

    - **The Azure Table Service** provides a table-structured storage mechanism based on the familiar rows and columns format, and supports queries for managing the data. It is primarily aimed at scenarios where large volumes of data must be stored, while being easy to access and update. For more detailed information see "*Table Service Concepts*" at http://msdn.microsoft.com/en-us/library/dd179463.aspxand "*Table Service REST API*" at http://msdn.microsoft.com/en-us/library/dd179423.aspx.
    - **The Binary Large Object (BLOB) Service** provides a series of containers aimed at storing text or binary data. It provides both block blob containers for streaming data, and page blob containers for random read/write operations. For more detailed information see "*Understanding Block Blobs and Page Blobs*" at http://msdn.microsoft.com/en-us/library/ee691964.aspx and "*Blob Service REST API*" at http://msdn.microsoft.com/en-us/library/dd135733.aspx.
    - **The Queue Service** provides a mechanism for reliable, persistent messaging between role instances, such as between a web role and a worker role. For more detailed information see "*Queue Service Concepts*" at http://msdn.microsoft.com/en-us/library/dd179353.aspxand "*Queue REST Service API*" at http://msdn.microsoft.com/en-us/library/dd179363.aspx.
    - **Windows Azure Drives** provide a mechanism for applications to mount a single volume NTFS VHD as a page blob, and upload and download VHDs via the blob. For more detailed information see "*Windows Azure Drive*" (PDF) at http://go.microsoft.com/?linkid=9710117.
- **Windows Azure SQL Database**.This is a highly available and scalable cloud database service built on SQL Server technologies, and supports the familiar T-SQLbased relational database model. It can be used with applications hosted in Windows Azure, and with other applications running on-premises or hosted elsewhere.For more detailed information see "*Windows Azure SQL Database*" at http://msdn.microsoft.com/en-us/library/ee336279.aspx.
  - **Data Synchronization**. SQL Data Sync is a cloud-based data synchronization service built on Microsoft Sync Framework technologies. It provides bi-directional data synchronization and data management capabilities allowing data to be easily shared between multiple SQL Databaseinstancesand between on-premises and Windows Azure SQL Databaseinstances.For more detailed information see "*SQL Data Sync*" at http://msdn.microsoft.com/en-us/library/windowsazure/hh456371.aspx.
- **Caching**.This service provides a distributed, in-memory, low latency and high throughput application cache service that requires no installation or management, and dynamically increases and decreases the cache size automatically as required. It can be used to cache application data, ASP.NET session state information, and for ASP.NET page output caching.For more detailed information see "*Caching in Windows Azure*" at http://msdn.microsoft.com/en-us/library/gg278356.aspx.

Chapter 5 of this guide and the associated sample application, and Lab 2 in the Hands-on-Labs, showhow you can use table storage.

Chapter 6 of this guide and the associated sample application, and Labs 3 and 4 in the Hands-on-Labs forthis guide, show how you can use blob and queue storage.

Chapters 3 and 5 of the guide "*Developing Applications for the Cloud*" (see http://wag.codeplex.com/) explain the concepts and implementation of multi-tenant architectures for data storage.

Chapter 5 of the guide "*Developing Applications for the Cloud*" and the associated sample application describe the use of queue storage.

Chapter 6 of the guide "*Developing Applications for the Cloud*" and the associated sample application, and Lab 4 in the Hands-on-Labs for that guide, describe the use of table storage (including data paging) and blob storage in multi-tenant applications.

Chapter 2 of this guideand Chapter 6 of the guide "*Developing Applications for the Cloud*", the associated example applications, and Lab 5 in the Hands-on-Labs for the guide "*Developing Applications for the Cloud*" use SQL Database for data storage.

Chapter 6 of the guide "*Developing Applications for the Cloud*", the associated example application, and Lab 4 in the Hands-on-Labs for the guide "*Developing Applications for the Cloud*" demonstrate use of the caching service.

Networking Services

Windows Azure provides several networking services that you can take advantage of to maximize performance, implement authentication, and improve manageability of your hosted applications. These services include the following:

- **Content Delivery Network** (CDN).The CDN allows you to cachepublicly available static data for applications at strategic locations that are closer (in network delivery terms) to end users. The CDN uses a number of data centers at many locations around the world, which store the data in blob storage that has anonymous access. These do not need to be locations where the application is actually running.For more detailed information see "*Content Delivery Network*" at http://msdn.microsoft.com/en-us/library/ee795176.aspx.
- **Virtual Network Connect**.This service allows you to configure roles of an application running in Windows Azure and computers on your on-premises network so that they appear to be on the same network. It uses a software agent running on the on-premises computer to establish an IPsec-protected connection to the Windows Azure roles in the cloud, and provides the capability to administer, manage, monitor, and debug the roles directly.For more detailed information see "*Windows Azure Connect*" at http://msdn.microsoft.com/en-us/library/gg433122.aspx.
- **Virtual Network Traffic Manager**.This is a service that allows you to set up request redirection and load balancing based on three different methods. Typically you will use Traffic Manager to maximize performance by redirecting requests from users to the instance in the closest data

center using the Performance method. Alternative load balancing methods available are Failover and Round Robin.For more detailed information see "*How to Configure Traffic Manager Settings*" at http://www.windowsazure.com/en-us/manage/services/networking/traffic-manager/.

- **Access Control**.This is a standards-based service for identity and access control that makes use of a range of identity providers (IdPs) that can authenticate users. ACS acts as a Security Token Service (STS), or token issuer, and makes it easier to take advantage of federation authentication techniques where user identity is validated in a realm or domain other than that in which the application resides. An example is controlling user access based on an identity verified by an identity provider such as Windows Live ID or Google.For more detailed information see "*Access Control Service 2.0*" at http://msdn.microsoft.com/en-us/library/gg429786.aspx and "*Claims Based Identity & Access Control Guide*" at http://claimsid.codeplex.com/.
- **Service Bus**.This provides a secure messaging and data flow capability for distributed and hybrid applications, such as communication between Windows Azure hosted applications and on-premises applications and services,without requiring complex firewall and security infrastructures. It can use a range of communication and messaging protocols and patterns to provide delivery assurance, reliable messaging; can scale to accommodate varying loads; and can be integrated with on-premises BizTalk Server artifacts.For more detailed information see "*Service Bus*" at http://msdn.microsoft.com/en-us/library/ee732537.aspx.

Chapter 4 of the guide "*Developing Applications for the Cloud*" (see http://wag.codeplex.com/) and the associated example application demonstrate how you can use the Content Delivery Network (CDN).

Detailed guidance on using Access Control can be found in the associated guide "*Claims Based Identity & Access Control Guide*" (see http://claimsid.codeplex.com/) and in Lab 3 in the Hands-on-Labs for that guide.

Other Services

Windows Azure provides the following additional services:

- **Business Intelligence Reporting**.This service allows you to develop and deploy business operational reports generated from data stored in a SQL Database instanceto the cloud. It is built upon the same technologies as SQL Server Reporting Services, and lets you uses familiar tools to generate reports. Reports can be easily accessed through the Windows Azure Management Portal, through a web browser, or directly from within your Windows Azure and on-premises applications.For more detailed information see "Windows Azure SQL *Reporting*" at http://msdn.microsoft.com/en-us/library/gg430130.aspx.
- **Marketplace**.This is an online facility where developers can share, find, buy, and sell building block components, training, service templates, premium data sets, and finished services and applications needed to build Windows Azure platform applications.For more detailed

information see "*Windows Azure Marketplace*" at http://msdn.microsoft.com/en-us/library/gg315539.aspx.

Developing Windows Azure Applications

This section describes the development tools and resources that you will find useful when building applications for Windows Azure and SQL Database.

Typically, on the Windows platform, you will use Visual Studio 2010 with the Windows Azure Tools for Microsoft Visual Studio. The Windows Azure SDKprovide everything you need to create Windows Azure applications, including local compute and storage emulators that run on the development computer. This means that you can write, test, and debug applications before deploying them to the cloud. The tools also include features to help you deploy applications to Windows Azure and manage them after deployment.

Markus says:

You can build and test Windows Azure applications using the compute and storage emulators on your development computer.

You can download the Windows Azure SDK, which includes the toolsfor Microsoft Visual Studio, and development tools for other platforms and languages such as iOS, Eclipse, Java, and PHP from "*Windows Azure Platform Tools*" at http://www.windowsazure.com/en-us/develop/downloads/.

For a useful selection of videos, Quick Start examples, and Hands-On-Labs that cover a range of topics to help you get started building Windows Azure applications, see "*Developer Center*" at http://www.windowsazure.com/en-us/develop/overview/.

The MSDN "*Developing Applications for Windows Azure*" section at http://msdn.microsoft.com/en-us/library/gg433098.aspx includes specific examples and guidance for creating hosted services, using the Windows Azure SDK to package and deploy applications, and a useful Quick Start example.

The Windows Azure Training Kit contains hands-on labs to get you quickly started. You can download it at http://www.microsoft.com/downloads/details.aspx?FamilyID=413E88F8-5966-4A83-B309-53B7B77EDF78&displaylang=en.

To understand how a Windows Azure role operates, and the execution lifecycle, see "*Real World: Startup Lifecycle of a Windows Azure Role*" athttp://msdn.microsoft.com/en-us/library/hh127476.aspx.

For a list of useful resources for developing and deploying databases in SQL Database, see "*Development (*Windows Azure SQL Database*)*" at http://msdn.microsoft.com/en-us/library/ee336225.aspx.

Upgrading Windows Azure Applications

After you deploy an application to Windows Azure, you will need to update it as you change the role services in response to new requirements, code improvements, or to fix bugs. You can simply redeploy a service by suspending and then deleting it, and then deploy the new version. However, you can avoid application downtime by performing staged deployments (uploading a new package and swapping it with the existing production version), or by performing an in-place upgrade (uploading a new package and applying it to the running instances of the service).

For information about how you can perform service upgradesby uploading a new package and swapping it with the existing production version, see "*How to Deploy a Service Upgrade to Production by Swapping VIPs in Windows Azure*" at http://msdn.microsoft.com/en-us/library/ee517253.aspx.

For information about how you can perform in-place upgrades, including details of how services are deployed into upgrade and fault domains and how this affects your upgrade options, see "*How to Perform In-Place Upgrades on a Hosted Service in Windows Azure*" at http://msdn.microsoft.com/en-us/library/ee517255.aspx.

If you need only to change the configuration information for a service without deploying new code you can use theweb portal or the management API to edit the service configuration file orto upload a new configuration file.

Managing, Monitoring, and Debugging Windows Azure Applications

This section describes the management tools and resources that you will find useful when deploying, managing, monitoring, and debugging applications in Windows Azure and SQL Database.

All storage and management subsystems in Windows Azure use REST-based interfaces. They are not dependent on any .NET Framework or Microsoft Windows® operating system technology. Any technology that can issue HTTP or HTTPS requests can access Windows Azure's facilities.

To learn about the Windows Azure managed and native Library APIs, and the storage services REST API, see "*API References for Windows Azure*" at http://msdn.microsoft.com/en-us/library/ff800682.aspx.

The REST-based service management API can be used as an alternative to the Windows Azure web management portal. The API includes features to work with storage accounts, hosted services, certificates, affinity groups, locations, and subscription information. For more information, see "*Windows Azure Service Management REST API Reference*" at http://msdn.microsoft.com/en-us/library/ee460799.aspx.

In addition to these components, the Windows Azure platform provides diagnostics services for activities such as monitoring an application's health. You can use the Windows Azure Management Pack and Operations Manager 2007 R2 to discover Windows Azure applications, get the status of each role instance, collect and monitor performance information, collect and monitor Windows Azure events, and collect and monitor the .NET Framework trace messages from each role instance. For more information, see"*Monitoring Windows Azure Applications*" at http://msdn.microsoft.com/en-us/library/gg676009.aspx.

Markus says:

Windows Azure includes components that allow you to monitor and debug cloud-hosted services.

For information about using the Windows Azure built-in trace objects to configure diagnostics and instrumentation without usingOperations Manager, and downloading the results, see "*Collecting Logging Data by Using Windows Azure Diagnostics*" at http://msdn.microsoft.com/en-us/library/gg433048.aspx.

For information about debugging Windows Azure applications, see "*Troubleshooting and Debugging in Windows Azure*" at http://msdn.microsoft.com/en-us/library/gg465380.aspx and "*Debugging Applications in Windows Azure*" at http://msdn.microsoft.com/en-us/windowsazure/WAZPlatformTrainingCourse_WindowsAzureDebugging.

Chapter 7, "Application Life Cycle Management for Windows Azure Applications" of this guide contains information about managing Windows Azure applications.

Managing SQL DatabaseInstances

Applications access SQL Database instances in exactly the same way as with a locally installed SQL Server using the managed ADO.NET data access classes, native ODBC, PHP, or JDBC data access technologies.

Windows Azure SQL Database instances can be managed through the web portal, SQL Server Management Studio, Visual Studio 2010 database tools, and a range of other tools for activities such as moving and migrating data, as well as command line tools for deployment and administration.

A database manager is also available to make it easier to work with Windows Azure SQL Database instances. For more information see http://msdn.microsoft.com/en-us/library/gg442309.aspx. For a list of other tools, see "*Developer Center*" at http://www.windowsazure.com/en-us/develop/overview/.

Windows Azure SQL Database supports a management API as well as management through the web portal. For information about the Windows Azure SQL Database management API see "*Management REST API Reference*" at http://msdn.microsoft.com/en-us/library/gg715283.aspx.

Windows Azure Subscription and Billing Model

This section describes the billing model for Windows Azure and SQL Database subscriptions and usage. To use Windows Azure you first create a billing account by signing up for Microsoft Online Services at https://mocp.microsoftonline.com/ or through the Windows Azure portal at https://windows.azure.com/. The Microsoft Online Services customer portal manages subscriptions to all Microsoft services. Windows Azure is one of these, but there are others such as Business Productivity Online, Windows Office Live Meeting, and Windows Intune.

Every billing account has a single account owner who is identified with a Windows Live® ID. The account owner can create and manage subscriptions, view billing information and usage data, and specify the service administrator for each subscription. A Windows Azure subscription is just one of these subscriptions.

Poe says:

The account owner and the service administrator for a subscription can be (and in many cases should be) different Live IDs.

Administrators manage the individual hosted services for a Windows Azure subscription using the Windows Azure portal at https://windows.azure.com/. A Windows Azure subscription can include one or more of the following:

- Hosted services, consisting of hosted roles and the instances within each role. Roles and instances may be stopped, in production, or in staging mode.
- Storage accounts, consisting of table, blob, and queue storage instances.
- Content Delivery Network instances.
- Windows Azure SQL Database instances.
- Windows Azure SQL Reporting instances.
- Access Control, Service Bus, and Caching instances.
- Windows Azure Connect and Traffic Manager instances.

Figure 1 illustrates the Windows Azure billing configuration for a standard subscription.

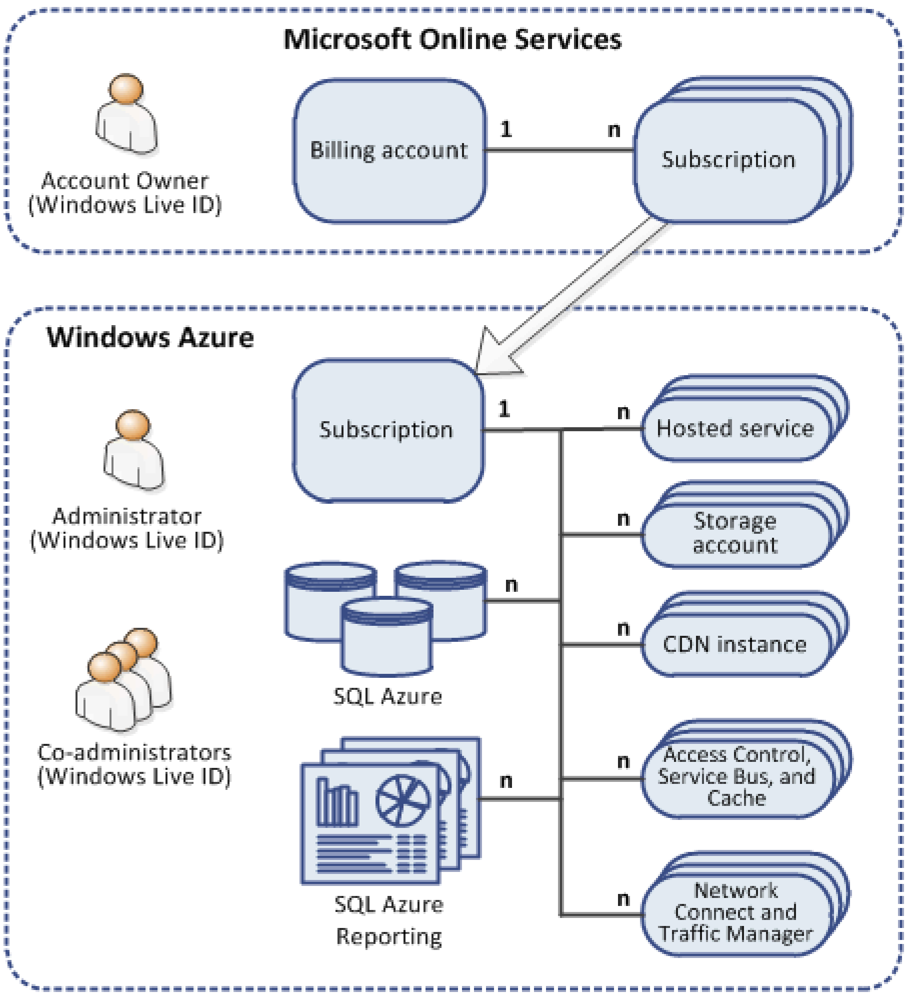

Microsoft Online Services
Account Owner (Windows Live ID)
Billing account
1
n
Subscription
Windows Azure
Administrator (Windows Live ID)
Co-administrators (Windows Live ID)
Subscription
1
n
Hosted service
n
Storage account
n
SQL Azure
n
CDN instance
n
SQL Azure Reporting
n
Access Control, Service Bus, and Cache
n
Network Connect and Traffic Manager

Figure 1
Windows Azure billing configuration for a standard subscription

For more information about Windows Azure billing, see "*Pricing Details*" at http://www.windowsazure.com/en-us/pricing/details/.

Estimating Your Costs

Windows Azure charges for how you consume services such as compute time, storage, and bandwidth. Compute time charges are calculated by an hourly rate as well as a rate for the instance size. Storage charges are based on the number of gigabytes and the number of transactions. Prices for data transfer vary according to the region you are in and generally apply to transfers between the Microsoft data centers and your premises, but not on transfers within the same data center.

To estimate the likely costs of a Windows Azure subscription, see the following resources:

- Subscription overviewfor the various purchasing models such as the pay-as-you-go and subscription model, including a tool for measuring consumption, at http://www.windowsazure.com/en-us/pricing/purchase-options/.
- Pricing calculator at http://www.windowsazure.com/en-us/pricing/calculator/.

Poe says:

You are billed for role resources that are used by a deployed service, even if the roles on those services are not running. If you don't want to get charged for a service, delete the deployments associated with the service.

Chapter 4 of this guideprovides additional information about estimating the costs of hosting applications in Windows Azure.

More Information

There is a great deal of information available about the Windows Azure platform in the form of documentation, training videos, and white papers. Here are some web sites you can visit to learn more:

- The website for this series of guides at http://wag.codeplex.com/ provides links to online resources, sample code, Hands-on-Labs, feedback, and more.
- The portal to information about Microsoft Windows Azure is at http://www.windowsazure.com/. It has links to white papers, tools such as the Windows Azure SDK, and many other resources. You can also sign up for a Windows Azure account here.
- Ryan Dunn and Steve Marx have a series of Channel 9 discussions about Azure at Cloud Cover, located at http://channel9.msdn.com/shows/Cloud+Cover/.
- Find answers to your questions on the Windows Azure Forum at http://social.msdn.microsoft.com/Forums/en-US/category/windowsazureplatform.
- Steve Marx blog is at http://blog.smarx.com/. It is a great source of news and information on Windows Azure.
- Ryan Dunn has a blog that covers Windows Azure topics at http://dunnry.com/blog.
- Eugenio Pace, a program manager in the Microsoft patterns & practices group, is creating a series of guides on Windows Azure, to which this documentation belongs. To learn more about the series, see his blog at http://blogs.msdn.com/eugeniop.
- Scott Densmore, lead developer in the Microsoft patterns & practices group, writes about developing applications for Windows Azure on his blog at http://scottdensmore.typepad.com/.
- Code and documentation for the patterns & practice Windows Azure Guidance project is available on the CodePlexWindows Azure Guidance site at http://wag.codeplex.com/.
- Comprehensive guidance and examples on Windows Azure Access Control Service is available in the patterns & practices book "*A Guide to Claims–based Identity and Access Control*", also available online at http://claimsid.codeplex.com/ .

2 – The Adatum Scenario

This chapter introduces a fictitious company named Adatum. The chapter describes Adatum's current infrastructure, its software portfolio, and why Adatum wants to move some of its applications to the Windows® Azure™technology platform. As with any company considering this process, there are many issues to take into account and challenges to be met, particularly because Adatum has not used the cloud before. The chapters that follow this one show, step-by-step, how Adatum modifies its expense tracking and reimbursement system, aExpense, so that it can be deployed to Windows Azure.

The Adatum Company

Adatum is a manufacturing company of 5,000 employees that mostly uses Microsoft® technologies and tools. It also has some legacy systems built on other platforms, such as AS400 and UNIX. As you would expect, Adatum developers are knowledgeable about various Microsoft products, including .NET Framework, ASP.NET, SQL Server® database software, Windows Server® operating system, and Microsoft Visual Studio® development system. Employees in Adatum's IT department are proficient at tasks such as setting up and maintaining Microsoft Active Directory® directory service and using System Center.

Adatum uses many different applications. Some are externally facing, while others are used exclusively by its employees. The importance of these applications ranges from "peripheral" to "critical," with many lying between the two extremes. A significant portion of Adatum's IT budget is allocated to maintaining applications that are either of mid-level or peripheral importance.

Adatum wants to change this allocation. Its aim is to spend more money on the services that differentiate it from its competitors and less on those that don't. Adatum's competitive edge results from assets, such as its efficient supply chain and excellent quality controls, and not from how effectively it handles its internal e-mail. Adatum wants efficient e-mail, but it's looking for more economical ways to provide this so that it can spend most of its budget on the systems that directly affect its customers. Adatum believes that one way to achieve this optimization is to selectively deploy applications to the cloud.

Adatum's Challenges

Adatum faces several challenges. Currently, deploying new on-premises applications takes too long, considering how quickly its business changes and how efficient its competitors are. The timeframe for acquiring, provisioning, and deploying even a simple application can be at least several weeks. No matter the application's complexity, requirements must be analyzed, procurement processes must be initiated, requests for proposals may need to be sent to vendors, networks must be configured, and so on. Adatum must be able to respond to its customers' demands more rapidly than the current procedures allow.

Another issue is that much of Adatum's infrastructure is used inefficiently. The majority of its servers are underutilized, and it's difficult to deploy new applications with the requisite service-level agreements (SLA) to the existing hardware. Virtual machines are appropriate in some cases, but they are not appropriate in all cases. This inefficiency means that Adatum's capital is committed to an underutilized infrastructure when it could be better used elsewhere in the business.

A final issue is that less critical applications typically get less attention from the IT staff. It is only when the application fails or cannot keep up with demand that anyone takes notice. By this time, the problem is expensive to fix, both in terms of IT time and in inefficient use of the users' time.

Adatum believes that by deploying some of its applications to a public cloud such as Windows Azure it can take advantage of economies of scale, promote standardization of its applications, and have automated processes for managing them. Most importantly, Adatum believes that this will make it more effective at addressing its customers' needs, a more effective competitor, and a better investment for its shareholders.

Adatum's Goals and Concerns

One of Adatum's goals is to improve the experience of all users of its applications. At a minimum, applications in the cloud should perform as well as their on-premises counterparts. The hope, though, is that they will perform better. Many of its applications are used more at some times than at others. For example, employees use the salary tool once every two weeks but rarely at other times. They would benefit if the applications had increased responsiveness during peak periods. This sensitivity to demand is known as *dynamic scalability*. However, on-premises applications that are associated with specific servers don't provide this flexibility. Adatum can't afford to run as many servers as are needed during peak times because this hardware is dormant the rest of the time. If these applications were located in the cloud, it would be easy to scale them depending on the demand.

Another goal is to expand the ways that users can access Adatum's applications. Currently, applications are only accessible from the intranet. Publishing them to the Internet is difficult and requires increased security. It also requires a virtual private network (VPN), which users often don't want to use because of the additional complexity that a VPN can introduce. Applications that are located in the public cloud are, by definition, available on the Internet. However, the public cloud also raises questions about security. In addition, many of Adatum's applications use Windows authentication so that users aren't required to enter application-specific credentials. Adatum is concerned that its users would need special credentials for each application in the public cloud.

A third goal is that at least some of Adatum's applications should be *portable*. Portability means that the application can be moved back and forth between a hosted data center to an on-premises data center without any modifications to the application's code or its operations. If both options are available, the risks that Adatum incurs if it does use the cloud are reduced.

In addition to its concerns about security, Adatum has two other issues. First, it would like to avoid a massive retraining program for its IT staff. Second, very few of Adatum's applications are truly isolated from other systems. Most have various dependencies. Adatum has put a great of deal effort into integrating its systems, even if not all of them operate on the same platform. It is unsure how these dependencies affect operations if some systems are moved to the public cloud.

Adatum's Strategy

Adatum is an innovative company and open to new technologies, but it takes carefully considered steps when it implements them. Adatum's plan is to evaluate the viability of moving to the cloud by starting with some of its simpler applications. It hopes to gain some initial experience, and then expand on what it has learned. This strategy can be described as "try, learn, fail fast, and then optimize." Adatum has decided to start with its aExpense application.

The aExpense Application

The aExpense application allows Adatum's employees to submit, track, and process business expenses. Everyone in Adatum uses this application to request reimbursements. Although aExpense is not a critical application, it is important. Employees can tolerate occasional hours of downtime, but prolonged unavailability isn't acceptable.

Adatum's policy is that employees must submit their expenses before the end of each month. The majority of employees don't submit their expenses until the last two business days. This causes relatively high demands during a short time period. The infrastructure that supports the aExpense application is scaled for average use across the month instead of for this peak demand. As a result, when the majority of employees try to submit their expenses during the last two business days, the system is slow and the employees complain.

The application is deployed in Adatum's data center and is available to users on the intranet. While traveling, employees access it through a VPN. There have been requests for publishing aExpense directly to the Internet, but it's never happened.

The application stores a great deal of information because most expense receipts must be scanned and then stored for seven years. For this reason, the data stores used by aExpense are frequently backed up.

The application is representative of many other applications in Adatum's portfolio so it's a good test case for using the cloud. Moving the aExpense application to Windows Azure will expose many of the challenges Adatum is likely to encounter as it expands the number of applications that it relocates to the cloud.

The aExpense Architecture

Figure 1 illustrates the aExpense architecture.

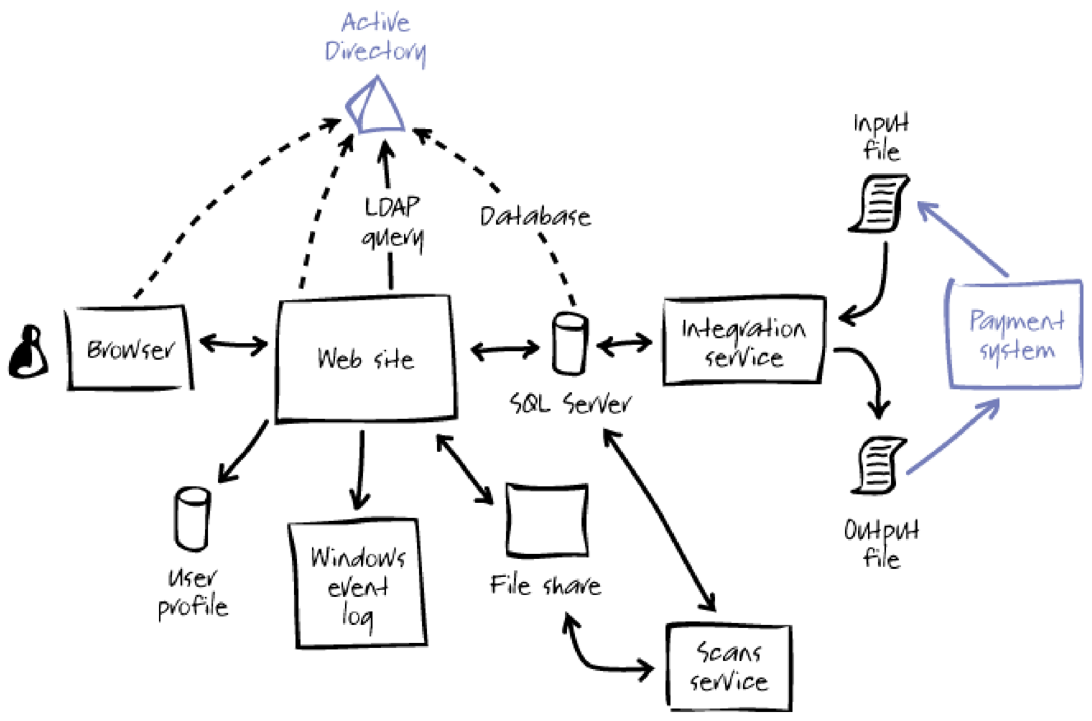


Figure 1
a-Expense architecture

The architecture is straightforward and one that many other applications use. aExpense is an ASP.NET application and employees use a browser to interact with it. The application uses Windows authentication for security. To store user preferences, it relies on ASP.NET membership and profile providers. Exceptions and logs are implemented with Enterprise Library's Exception Handling Application Block and Logging Application Block. The website uses Directory Services APIs to query for employee data stored in Active Directory, such as the employee's manager. The manager is the person who can approve the expenses.

The aExpense application implements the trusted subsystem to connect to SQL Server. It authenticates with a Windows domain account. The SQL database uses SQL Server authentication mode. The aExpense application stores its information on SQL Server. Scans of receipts are stored on a file share.

There are two background services, both implemented as Windows services. One periodically runs and generates thumbprints of the scanned receipts. It also compresses large images for increased storage efficiency. The other background service periodically queries the database for expenses that need to be reimbursed. It then generates a flat file that the payment system can process. This service also imports the payment results and sends them back to aExpense after the payments are made.

3 – Phase1: Getting to the Cloud

This chapter walks you through the first steps of migrating an application to the Windows® Azure™ technology platform. You'll see an example of how to take an existing business application, developed using ASP.NET, and move it to the cloud. This first stage is only concerned with getting the application to work in the cloud without losing any functionality. It does address some "big" issues, such as security and data storage that are relevant to almost any cloud-based application.

This first stage doesn't explore how to improve the application by exploiting the features available in Windows Azure. In addition, the on-premises version of the application that you'll see is not complete; it contains just enough basic functionality to get started. The following chapters discuss how to improve the application by using some of the features available in Windows Azure, and you'll see more features added to the application. For now, you'll see how to take your first steps into the cloud.

The Premise

The existing aExpense application is a business expense submission and reimbursement system used by Adatum employees. The application is built with ASP.NET 4.0, deployed in Adatum's datacenter, and is accessible from the Adatum intranet. The application relies on Microsoft Active Directory® to authenticate employees. It also uses Active Directory to access some of the user profile data that the application requires, for example, an employee's cost center and manager. Because aExpense uses Windows authentication, it recognizes the credentials used when employees log on to the corporate network and doesn't need to prompt them again for their user names and passwords.

Poe says:

Integration with Active Directoryreally simplifies the task of managing this application. The aExpense application leverages Active Directory's access management facilities, and the cost center and manager information that Adatum store in Active Directory.

The aExpense access control rules use application-specific roles such as "Employee" and "Manager." Access control is intermixed with the application's business logic.

The aExpense application uses a simple SQL Server® database for storing application data, and the application uses LINQ to SQL as its data access mechanism. The application is configured to connect to SQL Server by using integrated security, and the website uses a service account to log on to the database.

The aExpense application uses the Enterprise Library Logging Application Block and the Exception Handling Application Block for logging diagnostic information from the application.

Figure 1 shows a whiteboard diagram of the structure of the on-premises aExpense application.

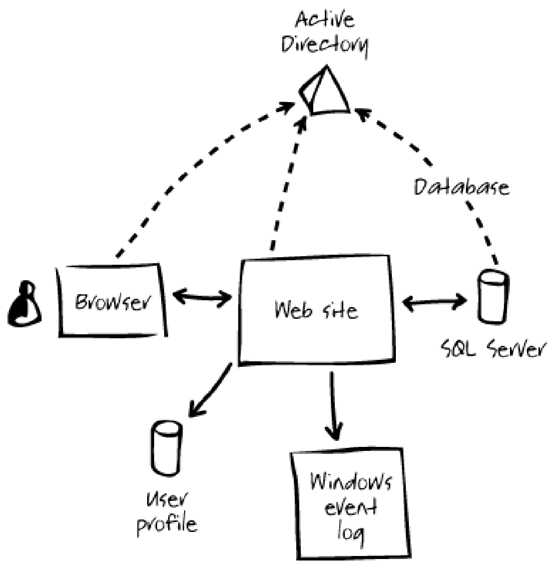


Figure 1

aExpense as an on-premises application

Goals and Requirements

In this first phase, Adatum has a number of goals for the migration of the aExpense application to the cloud that the team summarizes as "Getting it to work in the cloud." Optimizing the application for the cloud and exploiting the features of Windows Azure will come later.

Your decision to move an application to the cloud should be based on clear goals and requirements.

Adatum identified some specific goals to focus on in this first phase. The aExpense application in the cloud must be able to access all the same data that the on-premises version of the application can access. This includes the business expense data that the application processes and the user profile data, such as a user's cost center and manager, that it needs to enforce the business rules in the application. However, Adatum would like to remove any dependency on Active Directory from aExpense and avoid having the application call back into Adatum from the cloud.

Bharath says:

We want to avoid having to make any calls back into Adatum from the cloud application. This would add significantly to the complexity of the solution.

A second goal is to make sure that operations staff have access to the same diagnostic information from the cloud-based version of aExpense as they have from the existing on-premises version of the application.

A significant concern that Adatum has about a cloud-based solution is security, so a third goal is to continue to control access to the aExpense application based on identities administered from within

Adatum, and to enable users to access the application by using their existing credentials. Adatum does not want the overhead of managing additional security systems for its cloud-based applications.

Overall, the goals of this phase are to migrateaExpense to the cloud while preserving the user experience and the manageability of the application, and to make as few changes as possible to the existing application.

Overview of the Solution

The first step was to analyze the existing application to determine which pieces would need to change when it was migrated to the cloud. Remember that the goal at this stage is to make the application work in the cloud while making as few changes as possible to the application.

At this stage, Adatum wants to make as few changes as possible to the application.

The migration project team determined that they could replace SQL Server with Windows Azure SQL Database to meet the application's data storage requirements. They could easily copy the existing database structure and contents to Windows Azure SQL Database.

You can use the Migration Wizard at http://sqlazuremw.codeplex.com/ to help you to migrate your local SQL Server databases to Windows Azure SQL Databaseinstances.

They also determined that the application could continue to use the Enterprise Library application blocks in Windows Azure, and that the cloud-based application could continue to generate the same diagnostic information as the on-premises version.

Markus says:

It would be great if we could continue to use tried and tested code in the cloud version of the application.

You can download a white paper that explains the capabilities and limitations of Enterprise Library 5.0when it is used by .NET applications designed to run on the Windows Azure platform from http://wag.codeplex.com/.

The on-premises aExpense application stores users' preferred reimbursement methods by using the ASP.NET profiles feature. The default ASP.NET profile provider uses SQL Server as its storage mechanism. Because Windows Azure SQL Database is a relatively expensive storage mechanism (compared to Windows Azure table storage), and because the profile data is very simple, the team decided to use a profile provider implementation that used Windows Azure table storage. Switching to a different profile provider should have no impact on any existing code in the application.

The biggest changes to the application that the team identified were in the authentication and authorization functionality. The Adatum team decided to modify the application to use a claims-based system. Adatum will configure an on-premises Active Directory Federation Services (ADFS) claims issuer in their datacenter. When a user tries to access the aExpense application in the cloud, that user will be redirected to this claims issuer. If the user has not already logged on to the Adatum domain, the user will provide his or her Windows credentials, and the claims issuer will generate a token that contains a set of claims obtained from Active Directory. These claims will include the user's role membership, cost center, and manager. This will minimize the changes needed in the application and remove the direct dependency that the current version of the application has on Active Directory because the application will obtain the required user data from the claims issuer (the claims issuer still has to get the data from Active Directory on behalf of the aExpense application). The external claims issuer can integrate with Active Directory, so that application users will continue to have the same single sign-on experience.

Jana says:

Using claims can simplify the application by delegating responsibilities to the claims issuer.

Figure 2 shows the whiteboard drawing that the team used to explain the architecture of aExpense would look like after the migration to Windows Azure.

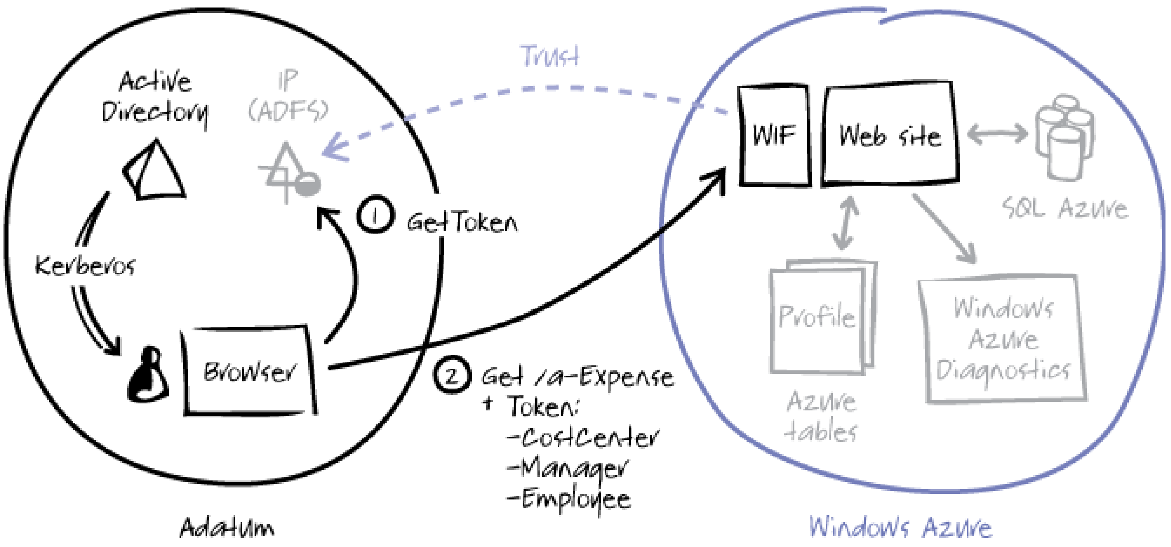


Figure 2
aExpense as an application hosted in Windows Azure

Inside the Implementation

Now is a good time to walk through the process of migrating aExpense into a cloud-based application in more detail. As you go through this section, you may want to download the Microsoft Visual Studio® development system solution from http://wag.codeplex.com/. This solution contains implementations of the aExpense application, before (in the BeforeAzure folder) and after the migration (in the Azure-SQLAzure folder). If you are not interested in the mechanics, you should skip to the next section.

Use the Visual Studio Windows Azure Project template from the Cloud section to get started with your cloud project.

Creating a Web Role

The developers at Adatum created the Visual Studio solution for the cloud-based version of aExpense by using the Windows Azure Project template. This template generates the required service configuration and service definition files, and the files for the web and worker roles that the application will need.

For more information about how to create a Windows Azure Project in Visual Studio, see the list of resources in the section "*Developing Windows Azure Applications*" in Chapter 1, "Introduction to the Windows Azure Platform."

This first cloud-based version of aExpense has a single web role that contains all the code from the original on-premises version of the application.

The service definition file defines the endpoint for the web role. The aExpense application only has a single HTTPS endpoint, which requires a certificate. In this case, it is known as "localhost." When you deploy the application to Windows Azure, you'll also have to upload the certificate.

```
<ServiceDefinition name="aExpense.Azure" xmlns="…">
<WebRole name="aExpense">
<Sites>
<Site name="Web">
<Bindings>
<Binding name="Endpoint1" endpointName="Endpoint1" />
</Bindings>
</Site>
</Sites>
<Endpoints>
<InputEndpoint name="Endpoint1" protocol="https" port="443"
        certificate="localhost" />
</Endpoints>
<Certificates>
<Certificate name="localhost" storeLocation="LocalMachine"
        storeName="My" />
</Certificates>
<ConfigurationSettings>
<Setting name="DataConnectionString" />
</ConfigurationSettings>
<Imports>
<Import moduleName="Diagnostics" />
</Imports>
<LocalResources>
<LocalStorage name="DiagnosticStore"
        cleanOnRoleRecycle="false" sizeInMB="5120" />
</LocalResources>
</WebRole>
</ServiceDefinition>
```

The "localhost" certificate is only used for testing your application.

The service configuration file defines the aExpenseweb role. It contains the connection strings that the role will use to access storage and details of the certificates used by the application. The application uses the **DataConnectionString** to connect to the Windows Azure storage holding the profile data, and uses the **DiagnosticsConnectionString** to connect to the Windows Azure storage for saving logging and performance data. The connection strings will need to change when you deploy the application to the cloud so that the application can use Windows Azure storage.

```
<ServiceConfiguration serviceName="aExpense.Azure" xmlns="…">
<Role name="aExpense">
<Instances count="1" />
<ConfigurationSettings>
<Setting name=
    "Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
               value="DefaultEndpointsProtocol=https;
               AccountName={Azure storage account name};
               AccountKey={Azure storage shared key}" />
<Setting name="DataConnectionString"
               value="DefaultEndpointsProtocol=https;
               AccountName={Azure storage account name};
               AccountKey={Azure storage shared key}" />
</ConfigurationSettings>
<Certificates>
<Certificate name="localhost" thumbprint="…"
                   thumbprintAlgorithm="sha1" />
</Certificates>
</Role>
</ServiceConfiguration>
```

The values of "Azure storage account name" and "Azure storage shared key" are specific to your Windows Azure storage account.

Markus says:

In Chapter 5, you'll see how Adatum automated editing the configuration file and uploading the certificate as part of the automated deployment process.

Securing aExpense

Before the migration, aExpense used Windows Authentication to authenticate users. This is configured in the Web.config file of the application.

After the migration, the aExpense application delegates the process of validating credentials to an external claims issuer instead of using Windows Authentication. You make this configuration change in the Web.config file.

To find out more about claims-based Identity, the **FedUtil** tool, and Windows Identity Foundation(WIF) take a look at 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.

The first thing that you'll notice in the Web.config file is that the authentication mode is set to **None**, while the requirement for all users to be authenticated has been left in place.

```
<authorization>
<deny users="?" />
</authorization>
<authentication mode="None" />
```

The **WSFederationAutheticationModule** (FAM) and **SessionAuthenticationModule** (SAM) modules now handle the authentication process. You can see how these modules are loaded in the **system.webServer** section of the Web.config file.

Markus says:

You can make these changes to the Web.config file by running the FedUtil tool.

```
<system.webServer>
  …
<add name="WSFederationAuthenticationModule"
       type="Microsoft.IdentityModel.Web.
           WSFederationAuthenticationModule, …" />
<add name="SessionAuthenticationModule"
       type="Microsoft.IdentityModel.Web.
           SessionAuthenticationModule, …" />
</system.webServer>
```

When the modules are loaded, they're inserted into the ASP.NET processing pipeline in order to redirect the unauthenticated requests to the claims issuer, handle the reply posted by the claims issuer, and transform the security token sent by the claims issuer into a **ClaimsPrincipal** object. The modules also set the value of the **HttpContext.User** property to the **ClaimsPrincipal** object so that the application has access to it.

More specifically, the **WSFederationAuthenticationModule** redirects the user to the issuer's logon page. It also parses and validates the security token that is posted back. This module also writes an encrypted cookie to avoid repeating the logon process. The **SessionAuthenticationModule** detects the logon cookie, decrypts it, and repopulates the **ClaimsPrincipal** object.After the claim issuer authenticates the user, the aExpense application can access the authenticated user's name.

The Web.config file contains a new section for the **Microsoft.IdentityModel** that initializes the Windows Identity Foundation (WIF) environment.

```
<microsoft.identityModel>
<service>
    …
</service>
</microsoft.identityModel>
```

You can also use a standard control to handle the user logout process from the application. The following code example from the Site.Master file shows a part of the definition of the standard page header.

```
<div id="toolbar">
    Logged in as:
<i>
<%= Microsoft.Security.Application.Encoder.HtmlEncode
(this.Context.User.Identity.Name) %>
</i> |
<idfx:FederatedPassiveSignInStatus
          ID="FederatedPassiveSignInStatus1"
          runat="server"
          OnSignedOut="FederatedPassiveSignInStatus1SignedOut"
          SignOutText="Logout" FederatedPassiveSignOut="true"
          SignOutAction="FederatedPassiveSignOut" />
</div>
```

You'll also notice a small change in the way that aExpense handles authorization. Because the authentication mode is now set to **None** in the Web.config file, the authorization rules in the Web.config file now explicitly deny access to all users as well as allowing access for the designated role.

```
<location path="Approve.aspx">
<system.web>
<authorization>
<allow roles="Manager" />
<deny users="*"/>
</authorization>
</system.web>
</location>
```

The claim issuer now replaces the ASP.NET role management feature as the provider of role membership information to the application.

There is one further change to the application that potentially affects the authentication process. If you were to run the aExpense application on more than one web role instance in Windows Azure, the default cookie encryption mechanism (which uses DPAPI) is not appropriate because each instance has a different key. This would mean that a cookie created by one web role instance would not be readable by another web role instance. To solve this problem you should use a cookie encryption mechanism that uses a key shared by all the web role instances. The following code from the Global.asax file shows how to replace the default **SessionSecurityHandler** object and configure it to use the **RsaEncryptionCookieTransform** class.

Bharath says:

Although the initial deployment of aExpense to Windows Azure will only use a single web role, we need to make sure that it will continue to work correctly when we scale up the application. That is why we use RSA with a certificate to encrypt the session cookie.

```
private void OnServiceConfigurationCreated(object sender,
    ServiceConfigurationCreatedEventArgs e)
{
    // Use the <serviceCertificate> to protect the cookies that
    // are sent to the client.
    List<CookieTransform> sessionTransforms =
        new List<CookieTransform>(
            new CookieTransform[]
            {
                new DeflateCookieTransform(),
                new RsaEncryptionCookieTransform(
                    e.ServiceConfiguration.ServiceCertificate),
                new RsaSignatureCookieTransform(
                    e.ServiceConfiguration.ServiceCertificate)
            });
   SessionSecurityTokenHandler sessionHandler =
    new
     SessionSecurityTokenHandler(sessionTransforms.AsReadOnly());

    e.ServiceConfiguration.SecurityTokenHandlers.AddOrReplace(
        sessionHandler);
}
```

Managing User Data

Before the migration, aExpense used an LDAP query to retrieve Cost Center, Manager, and Display Name information from Active Directory. It used the ASP.NET Role provider to retrieve the role membership of the user, and the ASP.NET Profile Provider to retrieve the application specific data for the application—in this case, the preferred reimbursement method. The following table summarizes how aExpense accesses user data, and where the data is stored before the migration:

User Data	Access Mechanism	Storage
Role Membership	ASP.NET Role Provider	SQL Server
Cost Center	LDAP	Active Directory
Manager	LDAP	Active Directory
Display Name	LDAP	Active Directory

User Name	ASP.NET Membership Provider	SQL Server
Preferred Reimbursement Method	ASP.NET Profile Provider	SQL Server

After the migration, aExpense continues to use the same user data, but it accesses the data differently. The following table summarizes how aExpense accesses user data, and where the data is stored after the migration:

User Data	Access Mechanism	Storage
Role Membership	ADFS	Active Directory
Cost Center	ADFS	Active Directory
Manager	ADFS	Active Directory
Display Name	ADFS	Active Directory
User Name	ADFS	Active Directory
Preferred Reimbursement Method	ASP.NET Profile Provider	Windows Azure table storage

The external issuer delivers the claim data to the aExpense application after it authenticates the application user. The aExpense application uses the claim data for the duration of the session and does not need to store it.

The external issuer delivers the claim data to the aExpense application after it authenticates the application user.

The application can read the values of individual claims whenever it needs to access claim data. You can see how to do this if you look in the **ClaimHelper** class.

Profile Data

Before the migration, aExpense used the ASP.NET profile feature to store application-specific user settings. Adatum tries to avoid customizing the schema in Active Directory, so aExpense stores a user's preferred reimbursement method by using the profile feature. The default Profile Provider stores the profile properties in a SQL Server database.

Poe says:

We don't like to customize the Active Directory schema if we can possibly avoid it. Schema changes have far-reaching implications and are difficult to undo.

Using the profile feature makes it very easy for the application to store small amounts of user data. You enable the profile feature and specify the properties to store in the Web.config file.

```
<profile defaultProvider="SqlProvider">
```

```
<providers>
<clear />
<add name="SqlProvider"
         type="System.Web.Profile.SqlProfileProvider"
         connectionStringName="aExpense"
         applicationName="aExpense" />
</providers>
<properties>
<add name="PreferredReimbursementMethod" />
</properties>
</profile>
```

Then you can access a profile property value in code like this.

```
var profile = ProfileBase.Create(userName);
string prm =
    profile.GetProperty<string>("PreferredReimbursementMethod");
```

After migration, aExpense continues to use the profile system to store the preferred reimbursement method for each user. Although it is possible to use the SQL Server profile provider in Windows Azure by using the custom scripts at http://support.microsoft.com/kb/2006191/,the solution uses a sample provider that utilizes Windows Azure table storage to store profile information. You can download this provider from Windows Azure ASP.NET Providers Sample. The only change required for the application to use a different profile provider is in the Web.config file.

Markus says:

Using a profile provider to access profile data minimizes the code changes in the application.

```
<profile defaultProvider="TableStorageProfileProvider">
<providers>
<clear />
<add name="TableStorageProfileProvider"
         type="AExpense.Providers.TableStorageProfileProvider …"
         applicationName="aExpense" />
</providers>

<properties>
<add name="PreferredReimbursementMethod" />
</properties>
</profile>
```

Using the **TableStorageProfileProvider** class does raise some issues for the application:

- The TableStorageProfileProvider is unsupported sample code.
- You must migrate your existing profile data from SQL Server to Windows Azure table storage.

- You need to consider whether, in the long run, Windows Azure table storage is suitable for storing profile data.

Even with these considerations to taken into account, using the table storage profile provider enabled Adatum to keep the changes in the application to a minimum; this means that the running costs of the application will be lower than they would be using SQL Database.

Chapter 4, "How Much Will It Cost?", describes the relative costs of using Windows Azure storage and Windows Azure SQL Database.

Chapter 5, "Phase 2: Automating Deployment and Using Windows Azure Storage," provides more information about using Windows Azure table storage.

Connecting to SQL Server

Before the migration, aExpense stores application data in a SQL Server database. In this first phase, the team moved the database to SQL Database and the data access code in the application remained unchanged. The only thing that needs to change is the connection string in the Web.config file.

Connecting to Windows Azure SQL Databaseinstead of an on-premises SQL Server requires only a configuration change.

```
<add name="aExpense" connectionString=
"Data Source={Server Name};
   Initial Catalog=aExpense;
UId={User Id};
Pwd={User Password};
Encrypt=True;
TrustServerCertificate=False;"
providerName="System.Data.SqlClient" />
```

The values of **Server Name**, **User Id**, and **User Password** are specific to your Windows Azure SQL Databaseaccount.

There are two things to notice about the connection string. First, notice that, because Windows Azure SQL Databasedoes not support Windows Authentication, the credentials for your account are stored in plain text. You should consider encrypting this section of the Web.config file. This will add to the complexity of your application, but it will enhance the security of your data. If your application is likely to run on multiple role instances, you must use an encryption mechanism that uses keys shared by all the role instances.

To encrypt your SQL connection string in the Web.config file, you can use the Pkcs12 Protected Configuration Provider that you can download from http://archive.msdn.microsoft.com/pkcs12protectedconfg.

For additional background information about using this provider, read the set of four blog posts on the SQL Database Team Blog starting with this one: http://blogs.msdn.com/b/sqlazure/archive/2010/09/07/10058942.aspx.

The second thing to notice about the connection string is that it specifies that all communications with Windows Azure SQL Databaseare encrypted. Even though your application may reside on a computerin the same data center as the database, you have to treat that connection as if it was using the internet.

Any traffic within the data center is considered "internet," so it should be encrypted.

Bharath says:

You can also add protection to your Windows Azure SQL Database instance by configuring the firewall in the portal. You can use the Windows Azure SQL Database firewall to specify the IP addresses of the computers that are permitted to connect to your databseserver.

Windows Azure SQL Database Connection Timeout

When you try to connect to SQL Database, you can specify a connection timeout value. If the timeout expires before establishing a connection, an error occurs, which your application must handle. How your application handles a timeout error depends on the specific circumstances, but possible strategies include keep retrying the connection until it succeeds, report the error to the user, or log the error and move on to another task.

The default connection timeout value is 15 seconds, but because the Service Level Agreement (SLA) specifies that an SLA violation does not occur until 30 seconds have elapsed, you should set your connection timeout to 30 seconds.

To retry connections you can use the **RetryPolicy** delegate in the **Microsoft.WindowsAzure.StorageClient** namespace. The article at http://blogs.msdn.com/windowsazurestorage/archive/2010/04/22/savechangeswithretries-and-batch-option.aspx describes how to use this delegate to retry saving changes to Windows Azure table storage, but you could adapt this approach to work with a context object in LINQ to SQL or ADO.NET Entity Framework.

Another option is the Enterprise Library Transient Fault Handling Application Block. For more details, see http://msdn.microsoft.com/en-us/library/hh680934(v=PandP.50).aspx.

Handling Dropped Connections

If a connection to SQL Database drops while your application is using the connection, you should immediately try to re-establish the connection. If possible, you should then retry the operation that was in progress before the connection dropped, or in the case of a transaction, retry the transaction. It is possible for a connection to fail between sending a message to commit a transaction and receiving a message that reports the outcome of the transaction. In this circumstance, you must have some way of checking whether the transaction completed successfully in order to determine whether you must retry it.

Diagnostics

The aExpense application uses the Logging Application Block and the Exception Handling Application Block from the Enterprise Library. The cloud-based version of aExpense continues to use these application blocks to help support and operations staff at Adatum troubleshoot problems. Although there are minimal changes required in the application to use these blocks in the cloud version of aExpense, the procedure for accessing the log files is different.

Jana says:

The Logging Application Block and the Exception Handling Application Block are part of the Enterprise Library. We use them in a number of applications within Adatum.

For aExpense to write logging information to Windows Azure logs, Adatum made a change to the Web.config file to make the Logging Application Block use the Windows Azure trace listener.

Poe says:

We want to have access to the same diagnostic data when we move to the cloud.

```
<listeners>
<add listenerDataType="Microsoft.Practices.EnterpriseLibrary.
  Logging.Configuration.SystemDiagnosticsTraceListenerData,
  Microsoft.Practices.EnterpriseLibrary.Logging,
Version=5.0.414.0,
  Culture=neutral, PublicKeyToken=31bf3856ad364e35"
type="Microsoft.WindowsAzure.Diagnostics
.DiagnosticMonitorTraceListener,
  Microsoft.WindowsAzure.Diagnostics, Version=1.0.0.0,
Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  traceOutputOptions="Timestamp, ProcessId"
  name="System Diagnostics Trace Listener" />
</listeners>
```

If you create a new Windows Azure Project in Visual Studio, the Web.config file will contain the configuration for the Windows Azure Diagnostics trace listener. The following code example from the Web.config file shows the trace listener configuration you must add if you are migrating an existing ASP.NET web application.

```
<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>
```

By default in Windows Azure, diagnostic data is not automatically persisted to storage; instead, it is held in a memory buffer.In order to access the diagnostic data, you musteither add some code to your application that transfers the data to Windows Azure storage, or add a diagnostics configuration file to your project. You can either schedule Windows Azure to transfer log data to storage at timed intervals, or perform this task on-demand.Adatum decided to use a diagnostics configuration file to control how the log data is transferred to persistent storage; the advantage of using a configuration file is that it enables Adatum to collect trace data from the **Application_Start** method where the aExpense application performs its initialization routines. The following snippet shows the sample diagnostics.wadcfg file from the solution.

```
<?xml version="1.0" encoding="utf-8" ?>
<DiagnosticMonitorConfiguration
xmlns="http://schemas.microsoft.com/ServiceHosting/2010/10/Diagno
sticsConfiguration"
      configurationChangePollInterval="PT1M"
      overallQuotaInMB="5120">
<DiagnosticInfrastructureLogs bufferQuotaInMB="1024"
     scheduledTransferLogLevelFilter="Verbose"
     scheduledTransferPeriod="PT1M" />
<Logs bufferQuotaInMB="1024"
     scheduledTransferLogLevelFilter="Verbose"
     scheduledTransferPeriod="PT1M" />
<Directories bufferQuotaInMB="1024"
     scheduledTransferPeriod="PT1M">

<!-- These three elements specify the special directories
           that are set up for the log types -->
<CrashDumps container="wad-crash-dumps" directoryQuotaInMB="256"
/>
<FailedRequestLogs container="wad-frq" directoryQuotaInMB="256"
/>
<IISLogs container="wad-iis" directoryQuotaInMB="256" />

</Directories>
<PerformanceCounters bufferQuotaInMB="512"
scheduledTransferPeriod="PT1M">
<!-- The counter specifier is in the same format as the
         imperative diagnostics configuration API -->
<PerformanceCounterConfiguration
```

```
       counterSpecifier="\Processor(_Total)\% Processor Time"
sampleRate="PT5S" />
</PerformanceCounters>
<WindowsEventLog bufferQuotaInMB="512"
     scheduledTransferLogLevelFilter="Verbose"
     scheduledTransferPeriod="PT1M">
<!-- The event log name is in the same format as the
         imperative diagnostics configuration API -->
<DataSource name="System!*" />
</WindowsEventLog>
</DiagnosticMonitorConfiguration>
```

The value of the **overallQuotaInMB** must be more than the sum of the **bufferQuotaInMB** values in the diagnostics.wadcfg file, and you must configure a local storage resource in the Web role named "DiagnosticsStore" that is at least the size of the **overallQuotaInMB** value in the diagnostics configuration file. In this example, the log files are transferred to storage every minute and you can then view them with any storage browsing tool such as the Server Explorer window in Visual Studio.

For more information about the diagnostics.wadcfg file, see http://msdn.microsoft.com/en-us/library/gg604918.aspx. This blog post also contains some useful tips: http://blogs.msdn.com/b/davidhardin/archive/2011/03/29/configuring-wad-via-the-diagnostics-wadcfg-config-file.aspx.

Bharath says:

Because persisting diagnostic data to Windows Azure storage costs money, we will need to plan how long to keep the diagnostic data in Windows Azure and how we are going to download it for offline analysis.

Setup and Physical Deployment

When you're developing an application for Windows Azure, it's best to do as much development and testing as possible by using the local Compute Emulator and Storage Emulator. When the application is ready, you can deploy it to Windows Azure for further testing. You shouldn't need to make any code changes before deploying to Windows Azure, but you will need to make sure that you upload any certificates that the application requires and update the configuration files for the Windows Azure environment.

You can upload SSL certificates to Windows Azure by using the Windows Azure Management Portal at http://windows.azure.com/or by using a script. For information about how to upload certificates by using a Windows PowerShell™ command line script, see Chapter 5, "Phase 2: Automating Deployment and Using Windows Azure Storage."

Role Instances, Upgrade Domains, and Fault Domains

Deploying multiple instances of web roles and worker roles is an easy way to scale out your application to meet increased demand. It's also easy to add or remove role instances as and when you need them, either through the Windows Azure Management Portal or by using scripts, so you only pay for the services you actually need. You can also use multiple role instances to enable fault tolerant behavior in your application, and to add the ability to perform "live" upgrades of your application.

Use multiple role instances to scale out your application, add fault tolerance, and enable in-place upgrades.

If you have two or more role instances, Windows Azure organizes them into virtual groupings known as upgrade domains. When you perform an in-place upgrade of your application, Windows Azure upgrades a single domain at a time; this ensures that the application remains available throughout the process. Windows Azure stops, upgrades, and restarts all the role instances in the upgrade domain before moving on to the next one.

There are some limitations to the types of upgrade that you can perform like this. In particular, you cannot modify the service configuration or add or remove roles from the application. You can also specify how many upgrade domains your application should have in the service configuration file.

In Windows Azure, fault domains are a physical unit of failure. If you have two or more role instances, Windows Azure will allocate them to multiple fault domains, so that if one fault domain fails, there will still be running instances of your application. Windows Azure determines how many fault domains your application uses.

Windows Azure also ensures upgrade domains and fault domains are orthogonal, so that the role instances in an upgrade domain are spread across different fault domains.

Deployment Scripts

Manually modifying your application's configuration files before you deploy the application to Windows Azure is an error-prone exercise. The developers at Adatum have developed a set of deployment scripts that automatically update the configuration files, package the application files, and upload the application to Windows Azure. You'll see more of these scripts later.

Using a Mock Issuer

By default, the downloadable version of aExpense is set up to run on a standalone development workstation. This is similar to the way you might develop your own applications. It's generally easier to start with a single development computer.

Poe says:

Using a simple, developer-created claims issuer is good practice during development and unit testing.

To make this work, the developers of aExpense wrote a small stub implementation of an issuer. You can find this code in the downloadable Visual Studio solution. The project is in the Dependencies folder and is named Adatum.SimulatedIssuer.

When you first run the aExpense application, you'll find that it communicates with the stand-in issuer. The issuer issues predetermined claims.

It's not very difficult to write this type of component, and you can reuse the downloadable sample, or you can start with the template included in the Windows Identity Foundation (WIF) SDK.

You can download the WIF SDK from the Microsoft Download Center. The book, "A Guide to Claims-Based Identity and Access Control," describes several ways you can create a claims issuer. You can download a PDF copy of this book from http://msdn.microsoft.com/en-us/library/ff423674.aspx.

Converting to a Production Issuer

When you are ready to deploy to a production environment, you'll need to migrate from the simulated issuer that runs on your development workstation to a component such as ADFS 2.0.

Making this change requires two steps. First, you need to modify the Web application's Web.config file using the **FedUtil** utility such that it points to the production issuer. Next, you need to configure the issuer so that it recognizes requests from your Web application and provides the appropriate claims.

To learn more about **FedUtil** and configuring applications to issue claims, take a look at 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.

You can refer to documentation provided by your production issuer for instructions about how to add a relying party and how to add claims rules.

When you forward a request to a claim issuer, you must include a *wreply* parameter that tells the claim issuer to return the claims. If you are testing your application locally and in the cloud, you don't want to hard code this URL because it must reflect the real address of the application. The following code shows how the aExpense application generates the *wreply* value dynamically in the Global.asax.cs file.

Building the *wreply* parameter dynamically simplifies testing the application in different environments.

```
private void
  WSFederationAuthenticationModule_RedirectingToIdentityProvider
  (object sender, RedirectingToIdentityProviderEventArgs e)
{
    // 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); // e.g. "http" or "https"
    wreply.Append("://");
    wreply.Append(request.Headers["Host"] ??
        requestUrl.Authority);
    wreply.Append(request.ApplicationPath);

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

    e.SignInRequestMessage.Reply = wreply.ToString();
}
```

Isolating Active Directory

The aExpense application uses Windows Authentication. Because developers do not control the identities in their company's enterprise directory, it is sometimes useful to swap out Active Directory with a stub during the development of your application.

The on-premises aExpense application (before the migration) shows an example of this. To use this technique, you need to make a small change to the Web.config file to swap Windows Authentication for Forms Authentication and then add a simulated LDAP profile store to the application. Swap Windows Authentication for Forms Authentication with the following change to the Web.config file.

```
<authentication mode="Forms">
<forms name=".ASPXAUTH"
      loginUrl="~/SimulatedWindowsAuthentication.aspx"
      defaultUrl="~/default.aspx" requireSSL="true">
</forms>
</authentication>
```

You need to add a logon page to the application that enables you to select the user that you want to use for testing. In aExpense, this page is known as SimulatedWindowsAuthentication.aspx.

You also need to add a class that simulates an LDAP lookup for the Active Directory attributes that your application needs. In this example, the **GetAttributes** method simulates the LDAP query "&(objectCategory=person)(objectClass=user);costCenter;manager;displayName".

```
public static class SimulatedLdapProfileStore
{
    public static Dictionary<string, string> GetAttributesFor(
        string userName, string[] attributes)
    {
        Dictionary<string, string> results;

        switch (userName)
        {
            case "ADATUM\\johndoe":
                results = new Dictionary<string, string>
                {
                    { "costCenter", "31023" },
                    { "manager", "ADATUM\\mary" },
                    { "displayName", "John Doe" }
                };
                break;

            …
        }

        return results;
    }
}
```

These code samples come from the BeforeAzure solution in the downloadable solutions.

SQL Server

At this stage, the development team at Adatum is working with sample data, so the deployment script builds a sample database in SQL Database. They will need to create a script that transfers data from the on-premises version of SQL Server to Windows Azure, when they come to migrate the live application. To migrate an existing database schema to Windows Azure SQL Database, you can use SQL Server Management Studio to export the schema as a Transact-SQL script, and then run the script against the Windows Azure SQL Databaseinstance. To move data to Windows Azure SQL Database, you can use SQL Server Integration Service. However, the team is planning to investigate whether they need Windows Azure SQL Database at all, or whether the application can utilize Windows Azure table storage. Therefore, they haven't spent any time yet developing a data migration strategy.

You can get an overview of the limitations of Windows Azure SQL Database that are important to be aware of at http://msdn.microsoft.com/en-us/library/ff394102.aspx.

You can get an overview of how to migrate a database to Windows Azure SQL Database at http://msdn.microsoft.com/en-us/library/ee730904.aspx. You can also use the migration wizard at http://sqlazuremw.codeplex.com/ to help you to migrate your local SQL Server databases to Windows Azure SQL Database.

Accessing Diagnostics Log Files

The on-premises version of aExpense uses the Logging Application Block and the Exception Handling Application Block to capture information from the application and write it to the Windows event log. The Windows Azure version of the application continues to use the same application blocks, and through a configuration change, it is able to write log data to the Windows Azure logging system. However, to access the log data in Windows Azure, you have to perform a few more steps. First, you need to save the log data to persistent storage. You can do this manually by using the Windows Azure Management Portal, or you can add code or a diagnostics.wadcfg file to your application to dump the log data to storage at scheduled intervals. Second, you need to have some way of viewing the log data in Windows Azure storage.

Poe says:

To access your log files in Windows Azure, you can find a utility that lets you access your Windows Azure storage remotely or develop some scripts that download them for you on a regular basis.

.

More Information

The Windows Azure Developer Center (http://www.windowsazure.com/en-us/develop/overview/) contains links to plenty of resources to help you learn about developing applications for Windows Azure.

MSDN contains more information about Windows Azure and SQL Database. A good starting place is at http://msdn.microsoft.com/en-us/library/dd163896.aspx.

You can download the latest versions of Windows Azure tools for developing applications using .NET and other languages from the Windows Azure Developer Center Downloads page at http://www.windowsazure.com/en-us/develop/downloads/.

4 – How Much Will It Cost?

This chapter presents a basic cost model for running the aExpense application in the cloud. It makes some assumptions about the usage of the application and uses the current pricing information for Windows® Azure™ technology platform services to estimate annual operational costs.

The Premise

The existing, on-premises, version of aExpense is a typical business application. Adatum selected this application as a pilot cloud migration project because the application has features that are common to many of Adatum's other business applications, and a Adatum hopes that any lessons learned from the project can be applied elsewhere. Adatum has deployed the aExpense application in its data center, with components installed across several different servers. The web application is hosted on a Windows Server box that it shares with another application. aExpense also shares a SQL Server® database software installation with several other applications. aExpense has its own dedicated drive array for storing scanned expense receipts.

The current deployment of aExpense is sized for average use, not peak use, so during the busy two days at month-end when the majority of users submit their business expense claims, the application can be slow and unresponsive.

Goals and Requirements

It is difficult for Adatum to determine accurately how much it costs to run the current version of aExpense. The application uses several different servers, shares an installation of SQL Server with several other business applications, and is backed up as part of the overall backup strategy in the data center.

It is very difficult to estimate the operational costs of an existing on-premises application.

Although Adatum cannot determine the existing running costs of the application, Adatum wants to estimate how much it will cost to run in the cloud. One of the specific goals of the pilot project is to discover how accurately it can predict running costs for cloud based applications.

A second goal is to estimate what cost savings might be possible by configuring the application in different ways. Adatum will then be able to assign a cost to a particular level of service, which will make it much easier to perform a cost-benefit analysis on the application. A specific example of this in the aExpense application is to estimate how much it will cost to deploy for peak demand at the busy month-end period.

Bharath says:

You can manage the cost of a cloud-based application by changing its behavior through configuration changes.

Overall, Adatum would like to see greater transparency in managing the costs of its suite of business applications.

Overview of the Solution

The first step was to analyze what Adatum will be billed for every month for a cloud-based version of aExpense. Figure 1 shows the services that Microsoft will bill Adatum for each month for the cloud version of the aExpense application.

Bharath says:

The simple cost model in this chapter does not include an estimate for the cost of any worker roles that Adatum may add to the aExpense application, and also assumes a single, small, web role instance. In addition, the model doesn't include any cost estimates for testing environments. Adatum should review its cost estimates when the aExpense application design is complete and when Adatum completes stress-testing the application.

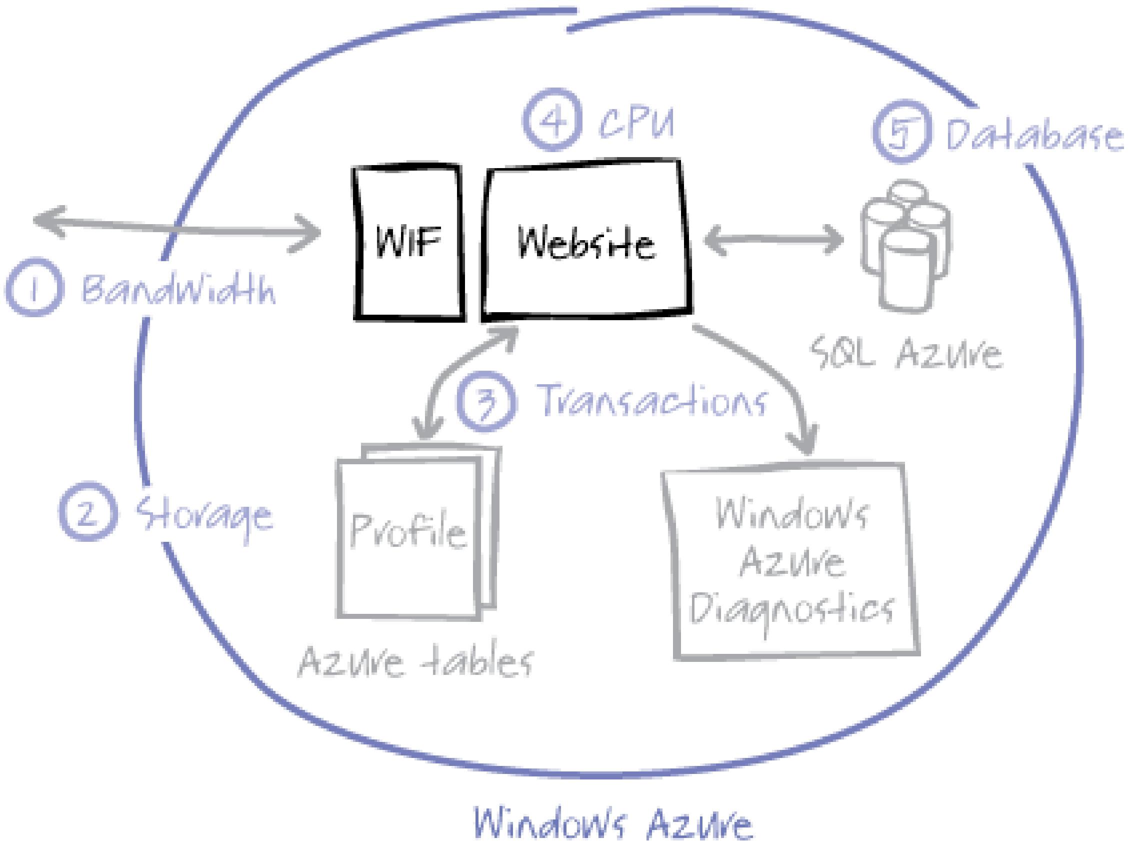


Figure 1
Billable services

The following table summarizes the current rates in U.S. dollars for these services at the time of writing.

Service	Description	Cost
1. In/Out Bandwidth	This is the web traffic between the user's browser and the aExpense site.	Inbound: Free Outbound (North America and Europe): $0.12 per GB Outbound (Asia Pacific Region): $0.19 per GB
2. Windows Azure Storage	In aExpense, this will be used to store just profile data initially. Later, it will also store scanned receipt images.	$0.125 per GB
3. Transactions	Each interaction with the storage system is billed.	$0.01 per 100,000 transactions
4. Compute	For the time the aExpense web roles are running.	Small size role $0.12 per hour
5. SQL Storage	Windows Azure SQL Databasecost per month.	UP to 100 MB: 4.995 Up to 1 GB: $9.99 Up to 10 GB: First GB $9.99, each additional GB $3.996 Up to 50 GB: First 10 GB $45.954, each additional GB $1.998 Up to 150 GB: First 50 GB $125.874, each additional GB $0.999

The prices listed here are accurate for the U.S. market as of June 2012. However, for up-to-date pricing information, see http://www.windowsazure.com/en-us/pricing/details/. You can find the pricing for other regions at the same address.

Bharath says:

After you have estimated your usage of the billable services by your application, you can use the Windows Azure Pricing calculator at http://www.windowsazure.com/en-us/pricing/calculator/ to quickly estimate your monthly costs.

Bandwidth Cost Estimate for aExpense

aExpense is not a bandwidth intensive application. Assuming that all scanned receipt images will be transferred back and forth to the application twice, and taking into account the web traffic for the application, Adatum estimated that 4 GB of data would move each way every month.

Data transfer	GB/month	$/GB/month	Total/month
Inbound	4GB	Currently free	$0.00

Outbound	4GB	$0.12	$0.48
		Total/year	**$5.76**

Windows Azure Storage Estimate for aExpense

Based on an analysis of existing usage, on average 60 percent of 5,000 Adatum employees submit 10 business expense items per month. Each scanned receipt averages 15 KB in size, and to meet regulatory requirements, the application must store 7 years of history. This gives an estimated storage requirement for the application of 36 GB.

Storage		**Cost**	**Total/month**
GB stored/month	36 GB	$0.125/ GB	$4.50
Storage transactions/month	90,000	$0.01/10 K	$0.009
		Total/year	**$54.11**

Compute Estimate for aExpense

Adatum's assumption here is that the application will run 24 hours/day, 365 days/year.

Hours	**$/hour**	**Number of instances**	**Total/year**
8760	$0.12	2	$2,102.40

To qualify for the Windows Azure SLA, you should have at least two instances of each of your roles. For more information, see http://www.windowsazure.com/en-us/support/legal/sla/.

Windows Azure SQL Database Storage Requirements Estimate

Adatum estimates that each business expense record in the database will require 2 KB of storage. So based on the analysis of existing usage (on average 60 percent of 5,000 Adatum employees submit 10 business expense items per month) and the requirement to store data for 7 years, this gives an estimated storage requirement of 4.8 GB.

SQL storage size	**$/month**	**Total/year**
Up to 10 GB	Up to $45.95	$551.40

Although Adatum could base the price calculation on a 5 GB Windows Azure SQL Database database that costs $25.97/month, Adatum is concerned that it may on occasion need more than 5 GB of storage. A compromise for the calculation might be 7 GB which would cost $33.97 per month or $407.64 per year.

This means that the costs as an approximate proportion of the total cost of running the application ($ 2713.67 per year) will be as follows:

- Compute (web and worker roles): $2102.40 (77 %)
- SQL Database: $ 551.40 (20 %)
- Windows Azure storage: $54.11 (2 %)
- Bandwidth: $5.76 (0.2 %)

Variations

One of the issues raised by users of the existing aExpense application is poor performance of the application during the two days at the end of the month when the application is most heavily used. To address this issue, Adatum then looked at the cost of doubling the compute capacity of the application for 2 days a month by adding an extra two web roles.

Poe says:

A common complaint about the existing version of aExpense is its poor performance at the time when most people want to use it.

Hours/month	Hours/year	$/hour	Role instances	$/year
48	576	$0.12	2	$138.24

Adatum also examined the cost implications of limiting access to the application to 12 hours/day for only 6 days/week.

Poe says:

It's easy to redirect users to an "Application is not currently available" page by using DNS.

Compute	Number of role instances	Hours/day	Days/year	$/hour	$/year
Standard	2	12	313	$0.12	$901.44
Month End	2	12	24	$0.12	$69.12
				Total/year	**$970.56**

This is about 50 percent of the compute cost of running the application 24 hours/day, 365 days/year.

Adatum was also interested in comparing the cost of storing the business expense data in Windows Azure table storage instead of SQL Database. The following table assumes that the storage requirement for seven years of data is the same 4.8 GB as for Windows Azure SQL Database. It also assumes that each new business expense item is accessed 5 times during the month.

Storage		Cost	Total/Month
GB stored/month	4.8 GB	$0.125/ GB	$0.60
Storage transactions/month	150,000	$0.01/10 K	$0.15
		Total/year	**$9.00**

As you can see, this is a fraction of the cost of using Windows Azure SQL Database($400.00 or more).

More Information

You can find the Windows Azure Pricing calculator at http://www.windowsazure.com/en-us/pricing/calculator/.

You can find information that will help you to understand your Windows Azure bill at http://www.windowsazure.com/en-us/pricing/details/.

5 – Phase 2: Automating Deployment and Using Windows Azure Storage

This chapter walks you through the changes Adatum made to the aExpense application during the second phase of the project. You'll see how Adatum extended the build process for aExpense to include a step that deploys the packaged application directly to Windows® Azure™ technology platform. You'll also see how Adatum changed the aExpenseapplication to use Windows Azure table storage instead of Windows Azure SQL Database and how the development team met some of the challenges they encountered along the way. The user-perceived functionality of the application didn't change from the previous phase.

The Premise

At the end of the first phase of the project, Adatum now had a version of the aExpense application that ran in the cloud. When the team at Adatum developed this version, they kept as much as possible of the original application, changing just what was necessary to make it work in Windows Azure.

The team does most of its testing using the Compute Emulator and Storage Emulator, which makes it easy for them to debug issues with the code. They also deploy the application to Windows Azure for additional testing in a staging environment in the cloud. They have found that manually deploying the application to Windows Azure through the Windows Azure Management Portal was error-prone, especially editing the configuration files with the correct connection strings.

Chapter 7, "Application Life Cycle Management for Windows Azure Applications," discusses testing applications for Windows Azure in more detail.

A simple cost analysis of the existing solution has revealed that Windows Azure SQL Database would account for about one third of the annual running costs of the application. Because the cost of using Windows Azure table storage is much lower than using Windows Azure SQL Database, Adatum is keen to investigate whether it can use Windows Azure table storage instead.

Goals and Requirements

In this phase, Adatum has two specific goals. The first is to evaluate whether the aExpense application can use Windows Azure table storage instead of SQL Database. Data integrity is critical, so Adatum wants to use transactions when a user submits multiple business expense items as a part of an expense submission.

You should evaluate whether Windows Azure table storage can replace Windows Azure SQL Databasein your application.

The second goal is to automate the deployment process to Windows Azure. As the project moves forward, Adatum wants to be able to deploy versions of aExpense to Windows Azure without needing to manually edit the configuration files, or use the Windows Azure Management Portal. This will make deploying to Windows Azure less error-prone, and easier to perform in an automated build environment.

Overview of the Solution

Moving from SQL Database to Windows Azure table storage involves some major changes to the application logic. The original version of aExpense used LINQ to SQL as the technology in the Data Access Layer (DAL) to communicate with the database. The DAL converted the data that it retrieved using LINQ to SQL to a set of domain-model objects that it passed to the user interface (UI). The new version of aExpense that uses Windows Azure table storage uses the .NET Client Library (which is a part of WCF Data Services) to interact with Windows Azure table storage.

Markus says:

WCF Data Services was previously named ADO.NET Data Services.

The Windows Azure table service only supports a subset of the functionality defined by the .NET Client Library for WCF Data Services. You can find more details at http://msdn.microsoft.com/en-us/library/dd894032.aspx.

The business expense data is now organized into two Windows Azure tables that have a parent/child relationship: an expense header record and expense item detail records.

Figure 1 shows a whiteboard diagram of the structure of the Windows Azure tables.

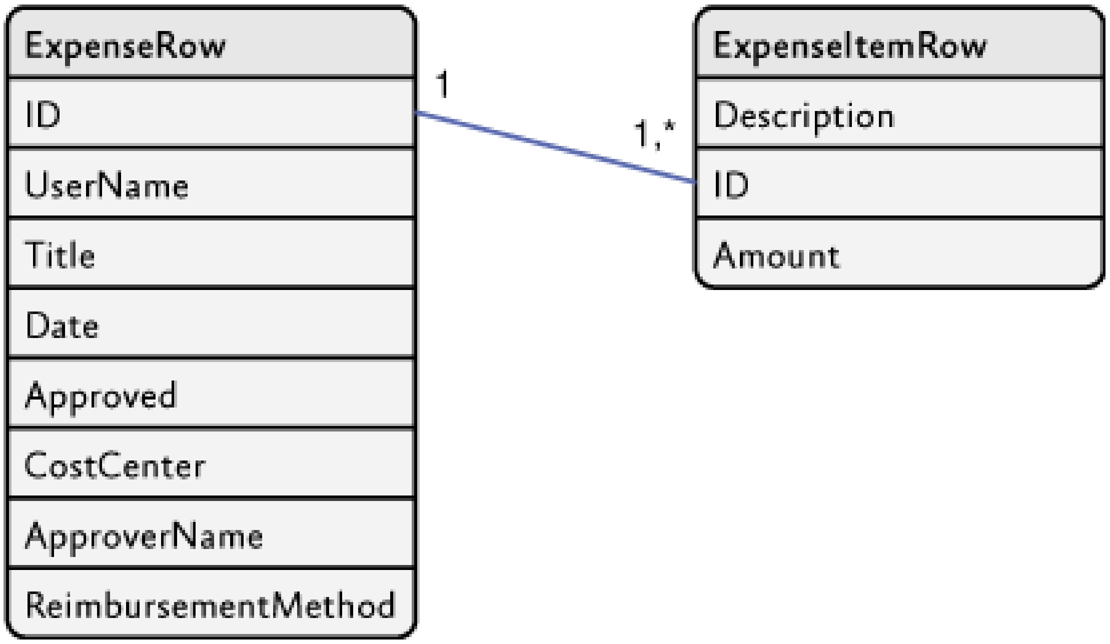


Figure 1
Windows Azuretable structure

In Chapter 8, "Phase 4: Adding More Tasks and Tuning the Application," of this guide you can find a description of an alternative solution that stores **ExpenseRows** and **ExpenseItemRows** in the same table.

The developers at Adatum have updated the web UI of the aExpense application to support adding multiple business expense items as a part of an expenses submission. The aExpense application uses Entity Group Transactions to ensure the integrity of the data in the application.

Bharath says:

You can use an Entity Group Transaction to group multiple table data modification operations on entities in the same table and partition group into a single, atomic transaction.

The automated deployment of aExpense is handled in two stages. The first stage uses an MSBuild script to compile and package the application for deployment to Windows Azure. This build script uses a custom MSBuild task to edit the configuration files for a cloud deployment, instead of a local Compute Emulator deployment. The second stage uses a Windows PowerShell script with some custom cmdlets to perform the deployment to Windows Azure.

Poe says:

Automating the deployment of applications to Windows Azure using scripts will make it much easier to manage applications running in the cloud.

Inside the Implementation

Now is a good time to walk through these changes in more detail. As you go through this section, you may want to download the Microsoft® Visual Studio® development system solution from http://wag.codeplex.com/. This solution (in the Azure-AzureStorage folder) contains the implementation of aExpense, after the changes made in this phase. If you are not interested in the mechanics, you should skip to the next section.

Automating Deployment to Windows Azure

Although you should not have to make any code changes when you deploy to Windows Azure instead of the local Compute Emulator, you will almost certainly need to make some configuration changes. The aExpense application now uses Windows Azure table storage for storing business expense data, so you must change both the **DataConnectionString** and the **Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString**in the ServiceConfiguration.csfg file to provide the information for the Windows Azure storage account that you want to use.

Deploying to the cloud will require configuration changes for the application.

This is what the relevant section of the configuration file looks like when the application is using development storage.

```
<ConfigurationSettings>
<Setting name=
    "Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
    value="UseDevelopmentStorage=true" />
<Setting name="DataConnectionString"
    value="UseDevelopmentStorage=true" />
</ConfigurationSettings>
```

This is what it looks like when the application is using cloud storage.

```
<ConfigurationSettings>
<Setting name=
    "Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
    value="DefaultEndpointsProtocol=https;
    AccountName={Azure storage account name};
    AccountKey={Azure storage shared key}" />
<Setting name="DataConnectionString"
    value="DefaultEndpointsProtocol=https;
    AccountName={Azure storage account name};
    AccountKey={Azure storage shared key}" />
</ConfigurationSettings>
```

The values of **Azure storage account name** and **Azure storage shared key** are specific to your Windows Azure storage account.

If you want to edit these settings manually, you should use the tools in Visual Studio instead of directly editing the XML.

The MSBuild script for the aExpense application uses a custom build task namedRegexReplace to make the changes during the build. The example shown here replaces the development storage connection strings with the Windows Azure storage connection strings.

Markus says:

You should also have a target that resets the development connection strings for local testing.

```
<Target Name="SetConnectionStrings"
DependsOnTargets="BuildTasks">
<RegexReplace
    Pattern='Setting name=
    "Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
      value="UseDevelopmentStorage=true"'
    Replacement='Setting name=
    "Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
      value="DefaultEndpointsProtocol=https;
      AccountName=$(StorageAccountName);
      AccountKey=$(StorageAccountKey)"'
    Files='$(AzureProjectPath)\$(ServiceConfigName)'/>
<RegexReplace
    Pattern='Setting name="DataConnectionString"
      value="UseDevelopmentStorage=true"'
    Replacement='Setting name="DataConnectionString"
      value="DefaultEndpointsProtocol=https;
      AccountName=$(StorageAccountName);
      AccountKey=$(StorageAccountKey)"'
    Files='$(AzureProjectPath)\$(ServiceConfigName)'/>
</Target>
```

The source code for the RegexReplace custom build task is available in the download for this phase.

The team at Adatum then developed a Windows PowerShell script (deploy.ps1) that would deploy the packaged application to Windows Azure. You can invoke this script from an MSBuild task. This script uses the Windows Azure PowerShell Cmdlets, a library of cmdlets that wrap the Windows Azure Service Management API and Diagnostics API. You can download this library athttps://www.windowsazure.com/en-us/manage/downloads/.

```
$buildPath = $args[0]
$packagename = $args[1]
$serviceconfig = $args[2]
$servicename = $args[3]
```

```
$thumbprint = $args[4]
$cert = Get-Item cert:\CurrentUser\My\$thumbprint
$sub = $args[5]
$slot = $args[6]
$storage = $args[7]
$package = join-path $buildPath $packageName
$config = join-path $buildPath $serviceconfig
$a = Get-Date
$buildLabel = $a.ToShortDateString() + "-" +
$a.ToShortTimeString()

#Important!  When using file based packages (non-http paths), the
#cmdlets will attempt to upload the package to blob storage for
#you automatically.  If you do not specifiy a –
#StorageServiceName option, it will attempt to upload a storage
#account with the same name as $servicename.  If that
#account does not exist, it will fail.  This only applies to
#file-based package paths.

if ((Get-PSSnapin | ?{$_.Name -eq "AzureManagementToolsSnapIn"})
   -eq $null)
{
  Add-PSSnapin AzureManagementToolsSnapIn
}

$hostedService = Get-HostedService $servicename -Certificate
  $cert -SubscriptionId $sub | Get-Deployment -Slot $slot

if ($hostedService.Status -ne $null)
{
    $hostedService |
      Set-DeploymentStatus 'Suspended' |
      Get-OperationStatus -WaitToComplete
    $hostedService |
      Remove-Deployment |
      Get-OperationStatus -WaitToComplete
}

Get-HostedService -ServiceName $servicename -Certificate $cert
    -SubscriptionId $sub | New-Deployment -Slot $slot
    -Package $package -Configuration $config -Label $buildLabel
    -ServiceName $servicename -StorageServiceName $storage |
    Get-OperationStatus -WaitToComplete

Get-HostedService -ServiceName $servicename -Certificate $cert
    -SubscriptionId $sub |
    Get-Deployment -Slot $slot |
    Set-DeploymentStatus 'Running' |
    Get-OperationStatus -WaitToComplete
```

The script needs the following two pieces of information to connect to Windows Azure, and you should replace the values of **Account API Certificate** and **Account Subscription ID** with values that are specific to your Windows Azure account:

- The thumbprint of the API certificate that is installed on your local computer. This certificate must match a certificate that you have uploaded to the Management Certificates page of the Windows Azure Management Portal.
- You can find your Subscription ID in the properties pane of the Management Certificates page in the Windows Azure Management Portal.

The API Certificate gives you access to all the Windows Azure Service Management API functionality. You may want to restrict access to this script to ensure that only authorized users have access to your Windows Azure services. This is not the same certificate as the SSL certificate used by the HTTPS endpoint.

The deployment script takes four parameters:

- *buildPath*. This parameter identifies the folder where you build your deployment package. For example: C:\AExpenseBuildPath.
- *packagename*. This parameter identifies the package to upload to Windows Azure. For example: aExpense.Azure.cspkg.
- *serviceconfig*. This parameter identifies the service configuration file for your project. For example: ServiceConfiguration.cscfg.
- *servicename*. This parameter specifies the name of your Windows Azure hosted service. For example: aExpense.

The script first verifies that the Windows Azure Service Management Cmdlets are loaded. Then, if there is an existing service, the script suspends and removes it. The script then deploys and starts the new version of the service.

Poe says:

The examples here deploy aExpense to the staging environment. You can easily modify the scripts to deploy to production. You can also script in-place upgrades when you have multiple role instances.

MSBuild can invoke the Windows PowerShell script in a task and pass all the necessary parameter values:

```
<Target Name="Deploy"
  DependsOnTargets="SetConnectionStrings;Build">
<MSBuild
    Projects="$(AzureProjectPath)\$(AzureProjectName)"
    Targets="CorePublish"
    Properties="Configuration=$(BuildType)"/>

<Exec WorkingDirectory="$(MSBuildProjectDirectory)"
    Command=
    "$(windir)\system32\WindowsPowerShell\v1.0\powershell.exe

    -NoProfile -f deploy.ps1 $(PackageLocation) $(PackageName)
    $(ServiceConfigName) $(HostedServiceName)
    $(ApiCertThumbprint) $(SubscriptionKey) $(HostSlot)
    $(StorageAccountName)" />

</Target>
```

See the release notes provided with the examples for information on using the Windows PowerShell scripts.

The aExpense application uses an HTTPS endpoint, so as part of the automatic deployment, Adatum needed to upload the necessary certificate. The following deploycert.ps1 Windows PowerShell script performs this operation.

```
$servicename = $args[0]
$certToDeploy = $args[1]
$certPassword = $args[2]
$thumbprint = $args[3]
$cert = Get-Item cert:\CurrentUser\My\$thumbprint
$sub = $args[4]
$algo = $args[5]
$certThumb = $args[6]


if ((Get-PSSnapin | ?{$_.Name -eq "AzureManagementToolsSnapIn"})
    -eq $null)
{
  Add-PSSnapin AzureManagementToolsSnapIn
}


try
{
  Remove-Certificate -ServiceName $servicename
    -ThumbprintAlgorithm $algo -Thumbprint $certThumb
    -Certificate $cert -SubscriptionId $sub
}
catch {}

Add-Certificate -ServiceName $servicename -Certificate $cert
    -SubscriptionId $sub -CertificateToDeploy $certToDeploy
    -Password $certPassword
```

The script needs the following two pieces of information to connect to Windows Azure, and you should replace the values of **Account API Certificate** and **Account Subscription ID** with values that are specific to your Windows Azure account:

- The thumbprint of the API certificate installed on your local computer. This certificate must match a certificate that you have uploaded to the Windows Azure Management Portal. For more information, see the page "About the Service Management API" at http://msdn.microsoft.com/en-us/library/ee460807.aspx.
- You can find your Subscription ID in the properties pane of the Management Certificates page in the Windows Azure Management Portal.

You must also pass the script three parameters:

- *servicename*. This parameter specifies the name of your Windows Azure service. For example: aExpense.
- *certToDeploy*. This parameter specifies the full path to the .pfx file that contains the certificate.
- *certPassword*. This parameter specifies the password that protects the private key in the .pfx file.

Windows Azure applications require you to create and install certificates for authentication and signing purposes. For information about the types of certificates required and how to create them, see "*How to Create a Certificate for a Role*" at http://msdn.microsoft.com/en-us/library/gg432987.aspx and "*How to: Manage Service Certificates*" at http://msdn.microsoft.com/en-us/library/gg551727.aspx.

The script doesn't check whether the certificate has already been deployed. If it has, the script will complete without an error.

An MSBuildfile can invoke this script and pass the necessary parameters. The following code is an example target from an MSBuildfile.

```
<Target Name="DeployCert">
<Exec WorkingDirectory="$(MSBuildProjectDirectory)"
Command=
"$(windir)\system32\WindowsPowerShell\v1.0\powershell.exe
-f deploycert.ps1 $(HostedServiceName) $(CertLocation)
$(CertPassword) $(ApiCertThumbprint) $(SubscriptionKey)
$(DeployCertAlgorithm) $(DeployCertThumbprint)" />
</Target>
```

Storing Business Expense Data in Windows Azure Table Storage

Modifying the aExpense application to use Windows Azure table storage instead of SQL Database meant that the developers at Adatum had to re-implement the Data Access Layer in the application. Because

Windows Azure table storage uses a fundamentally different approach to storage, this was not simply a case of replacing LINQ to SQL with the .NET Client Library.

Jana says:

Keeping all your data access code in a Data Access Layer restricts the scope of the code changes required if you need to change your storage solution (the code changes take place only in the DAL).

Use the .NET Client Library to access Windows Azure table storage.

How Many Tables?

The most important thing to understand when transitioning to Windows Azure table storage is that the storage model is different from what you may be used to. In the relational world, the obvious data model for aExpense would have two tables, one for expense header entities, and one for expense detail entities, with a foreign-key constraint to enforce data integrity. The best data model to use is not so obvious with Windows Azure table storage for a number of reasons:

- You can store multiple entity types in a single table in Windows Azure.
- Entity Group Transactions are limited to a single partition in a single table (partitions are discussed in more detail later in this chapter).
- Windows Azure table storage is relatively cheap, so you shouldn't be so concerned about normalizing your data and eliminating data redundancy.

Bharath says:

You have to stop thinking in relational terms when you are dealing with Windows Azure table storage.

Adatum could have used a single table to store both the expense header and expense detail entities. This approach would have enabled Adatum to use Entity Group Transactions to save an expense header entity and its related detail entities to a single partition in a single, atomic transaction. However, storing multiple entity types in the same table adds to the complexity of the application.

The sample application uses a single table solution, so the code samples in this chapter won't match exactly with the code in the downloaded solution.

You can find a description of how to store multiple entity types in the same table in Chapter 8, "Phase 4: Adding More Tasks and Tuning the Application," of this guide.

Adatum decided on a two-table solution for aExpense with tables named Expense and ExpenseItem. You can see the table definitions in the **ExpenseRow** and **ExpenseItemRow** classes in the **AExpense.DataAccessLayer** namespace. This approach avoids the complexity of multiple entity types in a table, but it does make it more difficult to guarantee the integrity of the data because the expense header entity can't be saved in the same transaction as the expense detail entities. How Adatum solved this problem is described in the section, "Transactions in aExpense," later in this chapter.

Adatum had to make a change to the data type that the application uses to store the business expense amount. In SQL Database, this field was stored as a decimal. This data type is not supported in Windows Azure table storage and the amount is now stored as a double.

Partition Keys and Row Keys

The second important decision about table storage is the selection of keys to use. Windows Azure table storage uses two keys: a partition key and a row key. Windows Azure uses the partition key to implement load balancing across storage nodes. The load balancer can identify "hot" partitions (partitions that contain data that is accessed more frequently than the data in other partitions) and run them on separate storage nodes in order to improve performance. This has deep implications for your data model design and your choice of partition keys:

Jana says:

Choosing the right partition key is the most important decision you make that affects the performance of your storage solution.

The partition key and row key together make up a tuple that uniquely identifies any entity in table storage.

- The partition key forms the first part of the tuple that uniquely identifies an entity in table storage.
- You can only use Entity Group Transactions on entities in the same table and in the same partition. You may want to choose a partition key based on the transactional requirements of your application. Don't forget that a table can store multiple entity types.

  Bharath says:

  Each entry in a table is simply a property bag. Each property bag can represent a different entity type; this means that a single partition can hold multiple entities of the same or different types.

- You can optimize queries based on your knowledge of partition keys. For example, if you know that all the entities you want to retrieve are located on the same partition, you can include the partition key in the **where** clause of the query. If the entities you want to retrieve span multiple partitions, you can split your query into multiple queries and execute them in parallel across the different partitions.

If you want to create parallel queries, you should plan to use Parallel LINQ (PLINQ) instead of creating your own threads in the web role.

The row key is a unique identifier for an entity within a partition and forms the second part of the tuple that uniquely identifies an entity in table storage.

In the aExpense application, Adatum decided to use the **UserName** property as the partition key of the Expense table. They anticipate that the vast majority of queries will filter based on the **UserName** property. For example, the website displays the expense submissions that belong to the logged on user. The ExpenseItem table uses the same partition key. This means that when a user inserts several **ExpenseItem** entities at the same time, the application can use a transaction to ensure that it inserts all the **ExpenseItem**instances (or none of them). It also means that all the data that belongs to a user is located on the same partition, so you only need to scan a single partition to retrieve a user's data.

Jana says:

Although the Expense and ExpenseItem tables use the same partition key values, you cannot have a transaction that spans tables in Windows Azure table storage.

In Chapter 8, "Phase 4: Adding More Tasks and Tuning the Application," this choice of partition key and row key is re-evaluated in the light of some results from testing the aExpense application.

For the Expense table, the application uses a GUID as the row key to guarantee a unique value. Because Windows Azure tables do not support foreign keys, for the ExpenseItem table, the row key is a concatenation of the parent Expense entity's row key and a GUID for the ExpenseItem row. This enables the application to filter **ExpenseItem** instances by ExpenseID as if there was a foreign key relationship. The following code in the **SaveChanges** method in the **ExpenseRepository** class shows how the application creates this row key value from the **Id** property of the expense header entity and the **Id** property of the expense detail entity.

```
expenseItemRow.RowKey = string.Format(
    CultureInfo.InvariantCulture,
    "{0}_{1}", expense.Id, expenseItemRow.Id);
```

The following code example shows how to query for **ExpenseItem** instances based on ExpenseID.

Markus says:

A more natural way of writing this query would be to use **StartsWith** instead of **CompareTo**. However, **StartsWith** is not supported by the Windows Azure table service. You also get performance benefits from this query because the **where** clause includes the partition key.

```
char charAfterSeparator =
    Convert.ToChar((Convert.ToInt32('_') + 1));
var nextId = expenseId.ToString() + charAfterSeparator;

var expenseItemQuery =
    (from expenseItem in context.ExpenseItem
     where
     expenseItem.RowKey.CompareTo(expenseId.ToString()) >= 0 &&
     expenseItem.RowKey.CompareTo(nextId) < 0 &&
     expenseItem.PartitionKey.CompareTo(expenseRow.PartitionKey)
       == 0
     select expenseItem).AsTableServiceQuery();
```

Windows Azure places some restrictions on the characters that you can use in partition and row keys. Generally speaking, the restricted characters are ones that are meaningful in a URL. For more information, see http://msdn.microsoft.com/en-us/library/dd179338.aspx. In the aExpense application, it's possible that these illegal characters could appear in the **UserName** used as the partition key value for the Expense table.

> If there is an illegal character in your partition key, Windows Azure will return a Bad Request (400) message.

To avoid this problem, the aExpense application encodes the **UserName** value using a base64 encoding scheme before using the **UserName** value as a row key. Implementing base64 encoding and decoding is very easy.

```
public static string EncodePartitionAndRowKey(string key)
{
    if (key == null)
    {
        return null;
    }

    return Convert.ToBase64String(
        System.Text.Encoding.UTF8.GetBytes(key));
}

public static string DecodePartitionAndRowKey(string encodedKey)
{
    if (encodedKey == null)
    {
        return null;
    }

    return System.Text.Encoding.UTF8.GetString(
        Convert.FromBase64String(encodedKey));
}
```

The team at Adatum first tried to use the **UrlEncode** method because it would have produced a more human readable encoding, but this approach failed because it does not encode the percent sign (%) character.

According to the documentation, the percent sign character is not an illegal character in a key, but Adatum's testing showed that entities with a percent sign character in the key could not be deleted.

Another approach would be to implement a custom escaping technique.

Markus says:

A custom methodto transform the user name to a legal character sequence could leave the keys human-readable, which would be useful during debugging or troubleshooting.

Query Performance

As mentioned earlier, the choice of partition key can have a big impact on the performance of the application. This is because Windows Azure tracks activity at the partition level, and can automatically migrate a busy partition to a separate storage node in order to improve data access performance for the application.

You can also use partition keys to improve the performance of individual queries. The current version of the application retrieves stored business expense submissions for a user by using this query.

```
var query = (from expense in context.Expenses
             where expense.UserName.CompareTo(userName) == 0
             select expense).AsTableServiceQuery();
```

As it stands, this query must scan all the partitions of the table to search for matching records. This is inefficient if there are a large number of records to search, and its performance may be further affected if it has to scan data across multiple storage nodes sequentially.

Jana says:

It's important to understand the impact that partitions can have on query performance.

When Adatum stress tests the application, it plans to evaluate whether modifying the query to reference the partition key in the **where** clause provides any significant performance improvements in the aExpense application.

```
var query = (from expense in context.Expenses
             where expense.PartitionKey.CompareTo(
                 EncodePartitionAndRowKey(userName)) == 0
             select expense).AsTableServiceQuery();
```

If a table query does not include the partition key in its **where** clause, to avoid the potential performance problems associated with scanning multiple partitions, you should re-evaluate your choice of row key and partition key for the table.

Transactions in aExpense

Figure 2 shows what happens when a user makes a business expense submission. The operation creates an expense master record and at least one expense detail item.

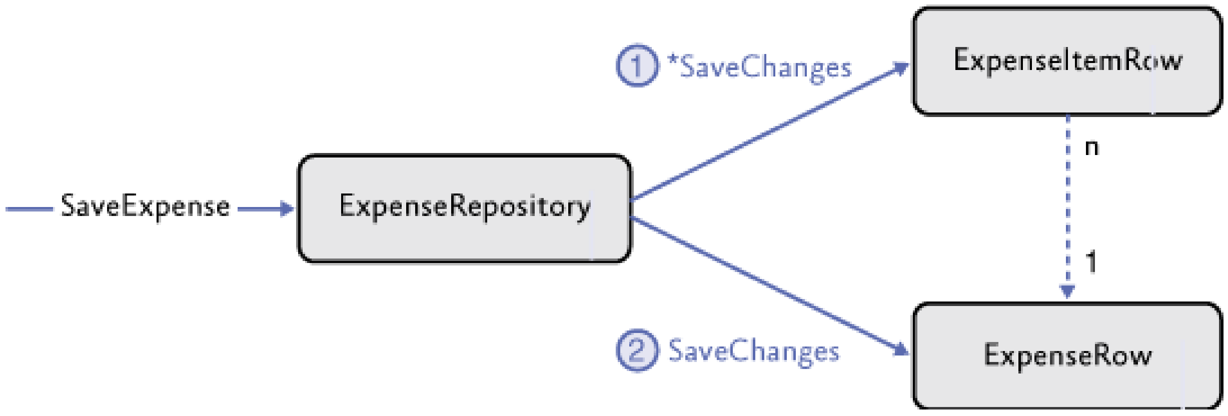


Figure 2
Saving a business expense submission

Adatum decided to store the Expense and ExpenseItem entities in two separate Windows Azure tables, but this means it cannot wrap the complete **SaveExpense** operation in a single transaction to guarantee its data integrity. The following code shows the solution Adatum adopted that uses both a transaction and compensating code to ensure data integrity.

Transactions cannot span tables in Windows Azure table storage.

You can find a description of how to store multiple entity types in the same table, and therefore simplify the way that transactions are handled in Chapter 8, "Phase 4: Adding More Tasks and Tuning the Application," of this guide.

```
public void SaveExpense(Expense expense)
{
    ExpenseDataContext context = new
        ExpenseDataContext(this.account);
    ExpenseRow expenseRow = expense.ToTableEntity();

    foreach (var expenseItem in expense.Details)
    {
        var expenseItemRow = expenseItem.ToTableEntity();
        expenseItemRow.PartitionKey = expenseRow.PartitionKey;
        expenseItemRow.RowKey =
            string.Format(CultureInfo.InvariantCulture,
            "{0}_{1}", expense.Id, expenseItemRow.Id);
        context.AddObject(ExpenseDataContext.ExpenseItemTable,
             expenseItemRow);
    }

    context.SaveChanges(SaveChangesOptions.Batch);

    context.AddObject(ExpenseDataContext.ExpenseTable,
        expenseRow);
    context.SaveChanges();
}
```

The code sample does not include any retry logic for when the call to the **SaveChanges** method fails. The article at http://blogs.msdn.com/windowsazurestorage/archive/2010/04/22/savechangeswithretries-and-batch-option.aspx summarizes how to use the **RetryPolicy** delegate to handle retry logic: when and how often to retry, which errors indicate not to retry. It also describes a workaround for a problem with retry policies and transactions.

The transaction takes place (implicitly) in the Windows Azure table storage service.In the method, the code first adds multiple **ExpenseItem** entities to the context and then calls the **SaveChanges** method. The **SaveChanges** method has a parameter **SaveChangesOptions.Batch** that tells the context to submit all the pending changes in a single request.If all the changes in the batch are to entities in the same partition of the same table,Windows Azure automatically uses an Entity Group Transaction when it persists the changes to table storage. An Entity Group Transaction provides the standard "all or nothing" transactional behavior to the operation.

Markus says:

There are some additional restrictions on performing Entity Group Transactions: each entity can appear only once in the transaction, there must be no more than 100 entities in the transaction, and the total size of the request payload must not exceed 4 megabytes (MB).

At the moment, we're assuming that no one will submit more than 100 business expense items as part of a single submission. We need to add some additional validation to this code to ensure the **SaveChanges** method can use an Entity Group Transaction.

After the **ExpenseItem** detail entities are inserted, the code saves the **ExpenseRow** header entity. The reason for saving the details first, followed by the header record, is that if there is a failure while saving the details, there will be no orphaned header record that could be displayed in the UI.

To resolve the potential issue of orphaned detail records after a failure, Adatum is planning to implement an "orphan collector" process that will regularly scan the Expense table looking for, and deleting, orphaned ExpenseItem records.

Working with Development Storage

There are some differences between development table storage and Windows Azure table storage documented at http://msdn.microsoft.com/en-us/library/gg433135.aspx. The team at Adatum encountered the error "One of the request inputs is not valid" that occurs when testing the application with empty tables in development storage. The solution that Adatum adopted was to insert, and then delete, a dummy row into the Windows Azure tables if the application is using the local Storage Emulator. During the initialization of the web role, the application calls the **CreateTableIfNotExist<T>** extension method in the **TableStorageExtensionMethods**class to check whether it is running against local development storage, and if this is the case, it adds, and then deletes, a dummy record to the application's Windows Azure tables.

Markus says:

Don't assume that local development storage will work in exactly the same way as Windows Azure storage.

You should consider adding dummy records to all tables in local development storage.

The following code from the **TableStorageExtensionMethods**class demonstrates how the aExpense application determines whether it is using development storage and how it adds and deletes a dummy record to the table.

```
public static bool CreateTableIfNotExist<T>(
    this CloudTableClient tableStorage, string entityName)
    where T : TableServiceEntity, new()
{
    bool result = tableStorage.CreateTableIfNotExist(entityName);

    // Execute conditionally for development storage only
    if (tableStorage.BaseUri.IsLoopback)
    {
        InitializeTableSchemaFromEntity(tableStorage,
            entityName, new T());
    }
    return result;
}

private static void InitializeTableSchemaFromEntity(
    CloudTableClient tableStorage, string entityName,
    TableServiceEntity entity)
{
    TableServiceContext context =
        tableStorage.GetDataServiceContext();
    DateTime now = DateTime.UtcNow;
    entity.PartitionKey = Guid.NewGuid().ToString();
    entity.RowKey = Guid.NewGuid().ToString();
    Array.ForEach(
        entity.GetType().GetProperties(BindingFlags.Public |
        BindingFlags.Instance),
        p =>
        {
            if ((p.Name != "PartitionKey") &&
                (p.Name != "RowKey") && (p.Name != "Timestamp"))
            {
               if (p.PropertyType == typeof(string))
               {
                   p.SetValue(entity, Guid.NewGuid().ToString(),
                       null);
               }
               else if (p.PropertyType == typeof(DateTime))
               {
                   p.SetValue(entity, now, null);
               }
           }
      });

    context.AddObject(entityName, entity);
    context.SaveChangesWithRetries();
    context.DeleteObject(entity);
    context.SaveChangesWithRetries();
}
```

Retrieving Data from Table Storage

The aExpense application uses LINQ to specify what data to retrieve from table storage. The following code example shows how the application retrieves expense submissions for approval by approver name.

Use the **AsTableServiceQuery** method to return data from Windows Azure table storage.

```
var query = (from expense in context.Expenses
    where expense.ApproverName.CompareTo(approverName) == 0
    select expense).AsTableServiceQuery();
```

The **AsTableServiceQuery** method converts the standard **IQueryable** result to a **CloudTableQuery** result. Using a **CloudTableQuery** object offers the following benefits to the application:

- Data can be retrieved from the table in multiple segments instead of getting it all in one go. This is useful when dealing with a large set of data.
- You can specify a retry policy for cases when the query fails.

Materializing Entities

In the aExpense application, all the methods in the **ExpenseRepository** class that return data from queries call the **ToList** method before returning the results to the caller.

Jana says:

Try to handle all data access issues within your data access layer.

```
public IEnumerable<Expense> GetExpensesForApproval(string
approverName)
{
    ExpenseDataContext context = new
        ExpenseDataContext(this.account);

    var query = (from expense in context.Expenses
                 where
expense.ApproverName.CompareTo(approverName) == 0
                 select expense).AsTableServiceQuery();

    try
    {
        return query.Execute().Select(e => e.ToModel()).ToList();
    }
    catch (InvalidOperationException)
    {
        Log.Write(EventKind.Error,
          "By calling ToList(), this exception can be handled
           inside the repository.");
        throw;
    }
}
```

The reason for this is that calling the **Execute** method does not materialize the entities. Materialization does not happen until someone calls **MoveNext** on the **IEnumerable** collection. Without **ToList**, the first call to **MoveNext** happens outside the repository. The advantage of having the first call to the **MoveNext** method inside the **ExpenseRepository** class is that you can handle any data access exceptions inside the repository.

Markus says:

Keeping data access functionality within the Data Access Layer helps separate concerns in the applications and localizes the interaction with the cloud to a small part of the application, which is great when you're debugging and troubleshooting.

More Information

The Windows Azure Table document at http://go.microsoft.com/fwlink/?LinkId=153401 contains detailed information about working with Windows Azure table storage.

6 – Phase 3: Uploading Images and Adding a Worker Role

This chapter walks you through the changes in the cloud-based version aExpense application that Adatum made when they added support for uploading, storing, and displaying scanned images of receipts. You'll see how the application uses Windows® Azure™ technology platform blob storage to store the image data, how the application uses a worker role in Windows Azure to perform background processing tasks on the images, and how the application uses shared access signatures to control access to the images. The chapter also introduces a simple set of abstractions that wrap a worker role in the expectation that the aExpense application will be given additional background tasks to perform in the future.

The Premise

During this phase of the project, the team at Adatum turned their attention to the first of the background processes in the aExpense application that performs some processing on the scanned images of business expense receipts that users upload.

The on-premises web application enables users to upload images of their business expense receipts and the application assigns a unique name to each image file before it saves the image to a file share. It then stores the path to the image in the SQL Server® database that holds the business expenses data. The application can then later retrieve the image related to an expense submission and display it through the user interface (UI).

The completed on-premises application also has a background process that processes the images,which is implemented as a Windows service. This process performs two tasks: first, it compresses the images to preserve disk space, and second, it generates a thumbnail image. By default, the application's UI displays the thumbnail, but if a user wants to see a more detailed image, it enables viewing the full-sized version of the image.

Goals and Requirements

Adatum has a number of goals for the implementation of the image-processing component of the application. Firstly, it wants to minimize the storage requirements for the images while maintaining the legibility of the information on the receipts.

Storage for scanned receipt images will be one of the monthly costs for the application.

It also wants to maintain the responsiveness of the application and minimize the bandwidth required by the UI. A user shouldn't have to wait after uploading an image while any necessary processing takes place, and the application should display image thumbnails with an option to display a full-sized version.

Finally, Adatum wants to maintain the privacy of its employees by making receipts visible only to the employee who submitted them and to the people responsible for approving the expense submission.

Jana says:

We need to protect the privacy of the users of this application.

Overview of the Solution

The team at Adatum made several significant changes to the implementation of the aExpense application for the Windows Azure based version in this phase. The first decision to make was to select a storage mechanism for the scanned receipt images. A simple approach to the application's storage requirements would be to use the Windows Azure drive. This would have required minimal changes to the code from the on-premises version, because the Windows Azure drive is a simple NTFS volume that you can access by using the standard .NET I/O classes. The big drawback of this approach is that you can only write the Windows Azure drive from one role instance at a time. Adatum expect to deploy multiple instances of the aExpense web role to ensure high availability and scalability for the application. The approach adopted by Adatum was to store the image data in Windows Azure block blob storage; although this approach required more changes to the code, it was compatible with using multiple role instances.

Adatum decided to implement the image processing service by using a worker role in the cloud-based version. Most of the code from the existing on-premises version of the application that compressed images and generated thumbnails was simply repackaged in the worker role. Using a worker role to process the images offloads work from the application's web role and helps to improve the responsiveness of the UI by performing the image processing on a background thread. What did change was the way that the image processing service identified when it had a new image to process. Figure 1 shows how, in the final on-premises version, the Windows Service uses the **FileSystemWatcher** class to generate an event whenever the application saves a new image to the file share. The aExpense application then invokes the image processing logic from the **OnCreated** event handler.

A worker role is the natural way to implement a background process in Windows Azure.

Bharath says:

Adatum should review the cost estimates for the aExpense application now that the application includes a worker role.

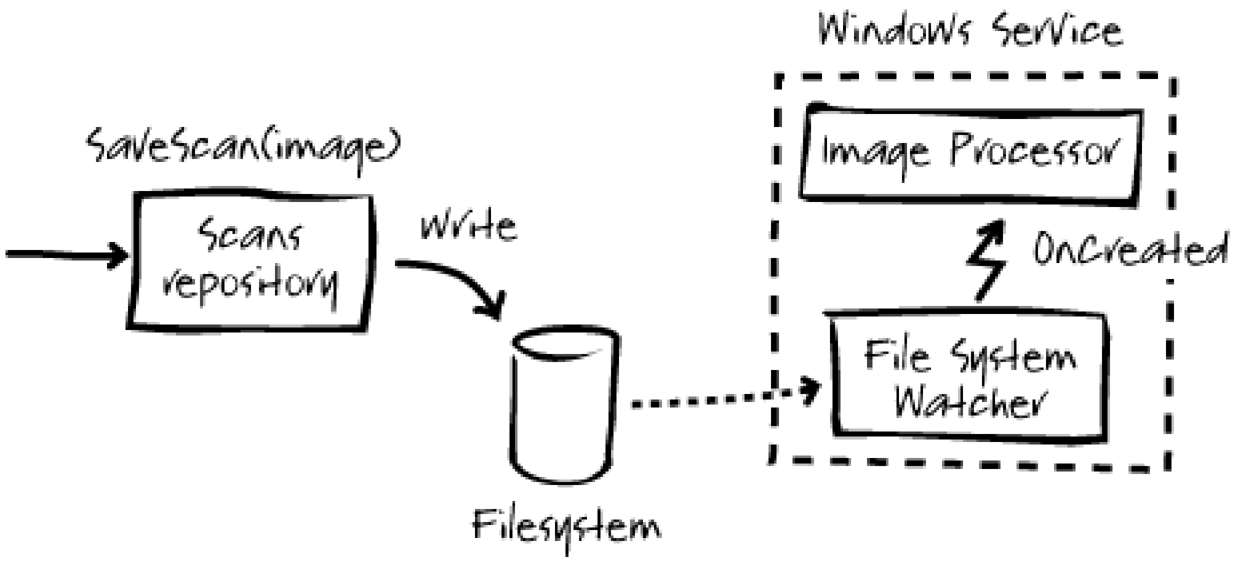


Figure 1

On-premises image processing

For the cloud-based version of the application using blob storage, this approach won't work because there is no Windows Azure equivalent of the **FileSystemWatcher** class for blob storage. Instead, as Figure 2 shows, Adatum decided to use a Windows Azure queue.

Markus says:

We can use a Windows Azure queue to communicate from the web role to the worker role.

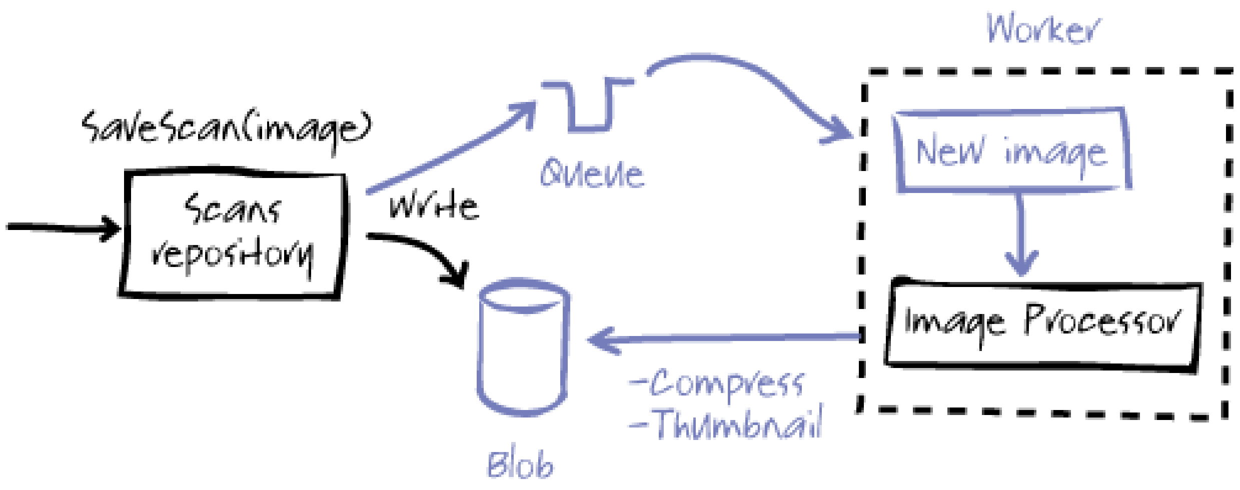


Figure 2

Cloud-based image processing

Whenever a user uploads a new image to aExpense as part of an expense submission, the application writes a message to the queue and saves the image to blob storage. The worker role will pick up messages from the queue, compress the image, and then generate a thumbnail. The worker role saves the new image and thumbnail in blob storage. After the worker role completes the image processing, it updates the expense item entity in Windows Azure table storage to include references to the image and the thumbnail and deletes the original image.

To display images in the UI, the application locates the images in blob storage from the information maintained in the expense item entity.

With Windows Azure queues, it is possible that a message could be read twice, either by two different worker roles, or by the same worker role. Although this would mean some unnecessary processing if it did occur, provided that the operation processing the message is idempotent, this would not affect the integrity of the application's data. In the aExpense application, a duplicate message causes the worker role to resize the original image and to generate the thumbnail a second time, overwriting the saved compressed image and thumbnail.

With Windows Azure queues, it is possible that a message could be read twice, either by two different worker roles, or by the same worker role.

If your message processing method is not idempotent, there are several strategies that you can adopt to stop the message recipient processing a message multiple times:

- When you read a message from a queue, you can use the *visibilitytimeout* parameter to set how long the messages should be hidden from other readers (the default value is 30 seconds). This gives you time to make sure that you can process and delete the message from the queue before another client reads it. Getting a message from a queue does not automatically delete it from the queue. It is still possible for the *visibilitytimeout* period to expire before you delete the message, for example, if the method processing the message fails.
- Each message has a **DequeueCount** property that records how many times the message has been retrieved from the queue. However, if you rely on this property, and only process messages that have a **DequeueCount** value of **0**, your application must guard against the possibility that a message has been dequeued but not processed.
- You could also add a unique transaction identifier to the message and then save the identifier in the blob's metadata properties. If, when you retrieve a message from a queue, the unique identifier in the message matches the unique identifier in the blob's metadata, you know that the message has already been processed once.

The application also allows users to view only images of receipts that they previously uploaded or images that belong to business expense submissions that they can approve. The application keeps other images hidden to protect other users' privacy. To achieve this, Adatum decided to use the Shared Access Signatures (SAS) feature in Windows Azure after they had evaluated a several other approaches.

Jana says:

We evaluated three alternative approaches to making business expense receipt images browsable before deciding on shared access signatures.

In Windows Azure, all storage mechanisms can be configured to allow data to be read from anywhere by using anonymous requests, which makes the model shown in Figure 3 very easy to implement:

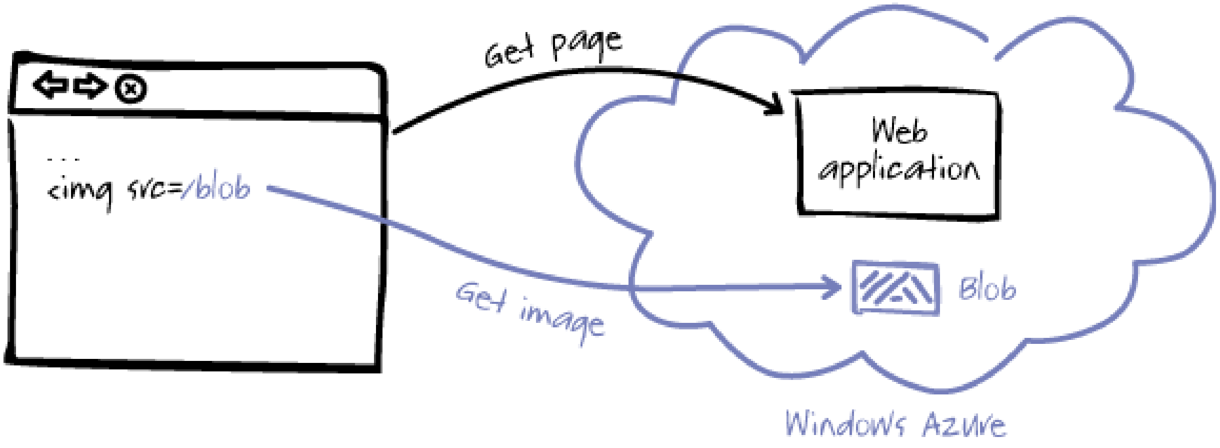


Figure 3
Directly addressable storage

In this scenario, you can access blob content directly through a URL such as **https://<application>.blob.core.windows.net/<containername>/<blobname>**. In the aExpense application, you could save the URLs as part of the expense entities in Windows Azure table storage. This is not an appropriate approach for the aExpense application because it would make it easy for someone to guess the address of a stored image, although this approach would work well for data that you did want to make publicly available, such as logos, branding, or downloadable brochures. The advantages of this approach are its simplicity, the fact that data is cacheable, that it offloads work from the web server, and that it could leverage the Content Delivery Network (CDN) infrastructure in Windows Azure. The disadvantage is the lack of any security.

Using deliberately obscure and complex URLs is a possible option, but this approach offers only weak protection and is not recommended.

Bharath says:

CDN enables you to cache blob data at strategically placed locations for delivering the content to users with the maximum available bandwidth.

The second approach considered by Adatum for accessing receipt images in blob storage was to route the request through the web site in much the same way that a "traditional" tiered application routes requests for data through the middle tier. Figure 4 shows this model.

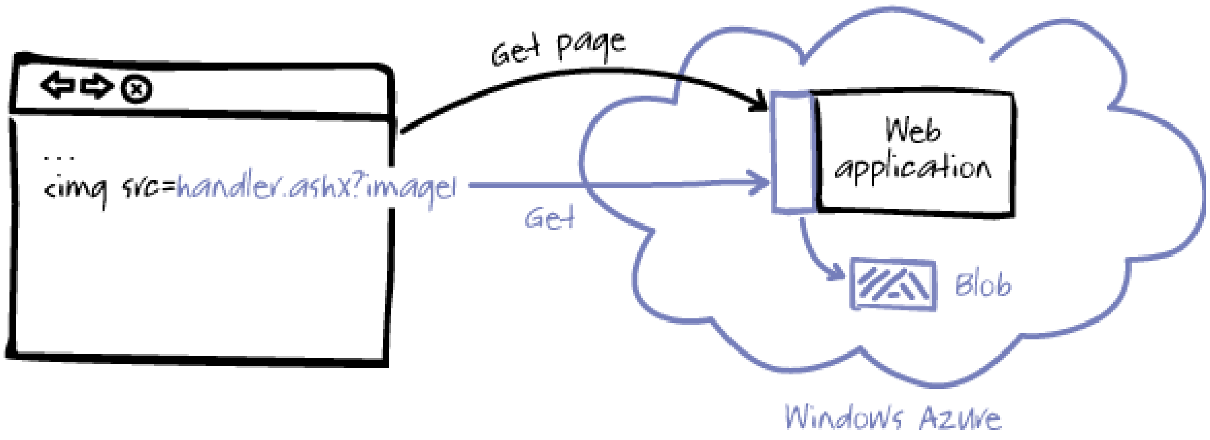


Figure 4

Routing image requests through the web server

In this scenario, there is no public access to the blob container, and the web application holds the access keys for blob storage. This is how aExpense writes the image data to blob storage. Although this approach would enable Adatum to control access to the images, it would add to the complexity of the web application and to the workload of the web server. A possible implementation of this scenario would be to use an HTTP handler to intercept image requests, check the access rules, and return the data. In addition, you couldn't use this approach if you wanted to use the Windows Azure CDN feature.

The approach that Adatum decided on for the aExpense application was to use the Windows Azure SAS feature. SAS enables you to control access to individual blobs, even though access is set at the container level, by generating blob access URLs that are only valid for a limited period of time. In the aExpense application, the web role generates these special URLs and embeds them in the page served to users. These special URLs then allow direct access to the blob for a limited period of time. There is some additional work for the web server, because it must generate the SAS URLs, but Windows Azure blob storage handles most of the work. The approach is reasonably secure because the SAS URLs, in addition to having a limited lifetime, also contain a uniquely generated signature, which makes it very difficult for anyone to guess a URL before it expires.

Jana says:

SAS works well from a security perspective; the URL is only valid for a limited period of time, and would be difficult to guess.

Inside the Implementation

Now is a good time to walk through these changes in more detail. As you go through this section, you may want to download the Microsoft® Visual Studio® development system solution from http://wag.codeplex.com/. This solution (in the Azure-WithWorkers folder) contains the implementation of aExpense after the changes in this phase are made. If you are not interested in the mechanics, you should skip to the next section.

Uploading and Saving Images

In the aExpense application, the web role is responsible for uploading the image from the user's workstation and saving the initial, uncompressed version of the image. The following code in the **SaveExpense** method in the **ExpenseRepository** class saves the original, uncompressed image to blob storage.

```
this.receiptStorage.AddReceipt(expenseItem.Id.ToString(),
    expenseItem.Receipt, string.Empty);
```

The following code from the **ExpenseReceiptStorage** class shows how the application saves the image data to blob storage.

```
public string AddReceipt(string receiptId, byte[] receipt,
    string contentType)
{
    CloudBlob blob = this.container.GetBlobReference(receiptId);
    blob.Properties.ContentType = contentType;
    blob.UploadByteArray(receipt);

    return blob.Uri.ToString();
}
```

Abstracting the Worker Role

Figure 5 summarizes the common pattern for the interaction between web roles and worker roles in Windows Azure.

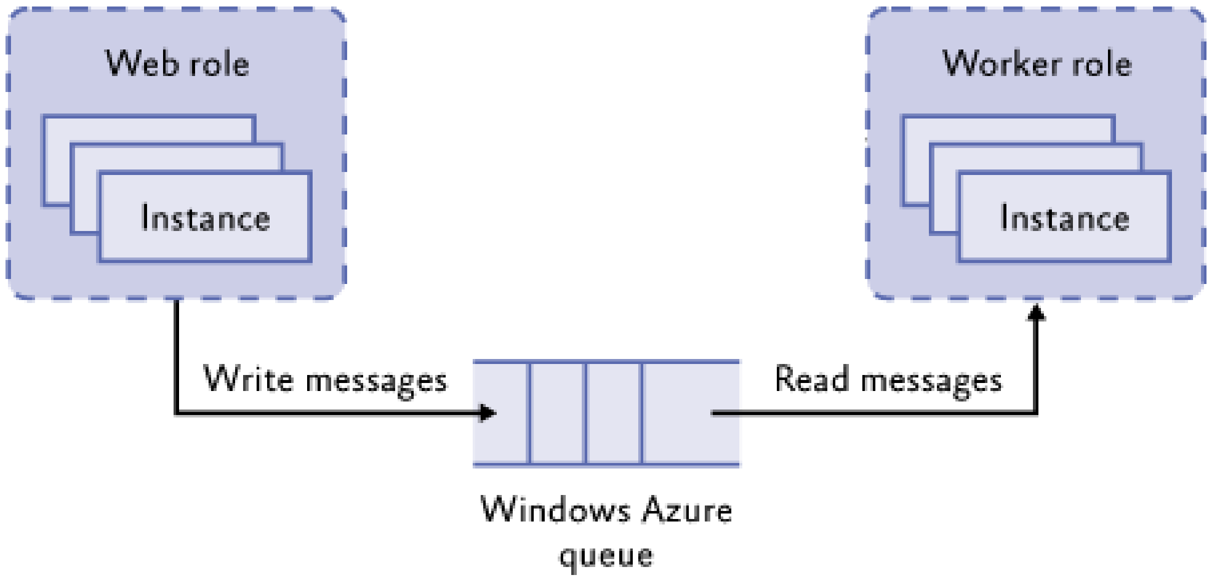


Figure 5

Web-to-worker role communication with a Windows Azure queue

Jana says:

This is a very common pattern for communicating between the web role and the worker role.

In this pattern, to communicate with the worker role, a web role instance places messages on to a queue, and a worker role instance polls the queue for new messages, retrieves them, and processes them. There are a couple of important things to know about the way the queue service works in Windows Azure. First, you reference a queue by name, and multiple role instances can share a single queue. Second, there is no concept of a typed message; you construct a message from either a string or a byte array. An individual message can be no more than 8 kilobytes (KB) in size.

If the size of your messages could be close to the maximum, be aware that Windows Azure converts all messages to Base64 before it adds them to the queue.

In addition, Windows Azure implements an "at-least-once" delivery mechanism; thus, it does not guarantee to deliver messages on a first-in, first-out basis, or to deliver only a single copy of a message, so your application should handle these possibilities.

Windows Azure does not guarantee to deliver messages on a first-in, first-out basis, or to deliver only a single copy of a message.

Although in the current phase of the migration of aExpense to Windows Azure, the worker role only performs a single task, Adatum expects the worker role to take on additional responsibilities in later phases. Therefore, the team at Adatum developed some simple "plumbing" code to hide some of the complexities of Windows Azure worker roles and Windows Azure queues and to make them easier to work with in the future. Figure 6 is a high-level description of these abstractions and shows where to plug in your custom worker role functionality.

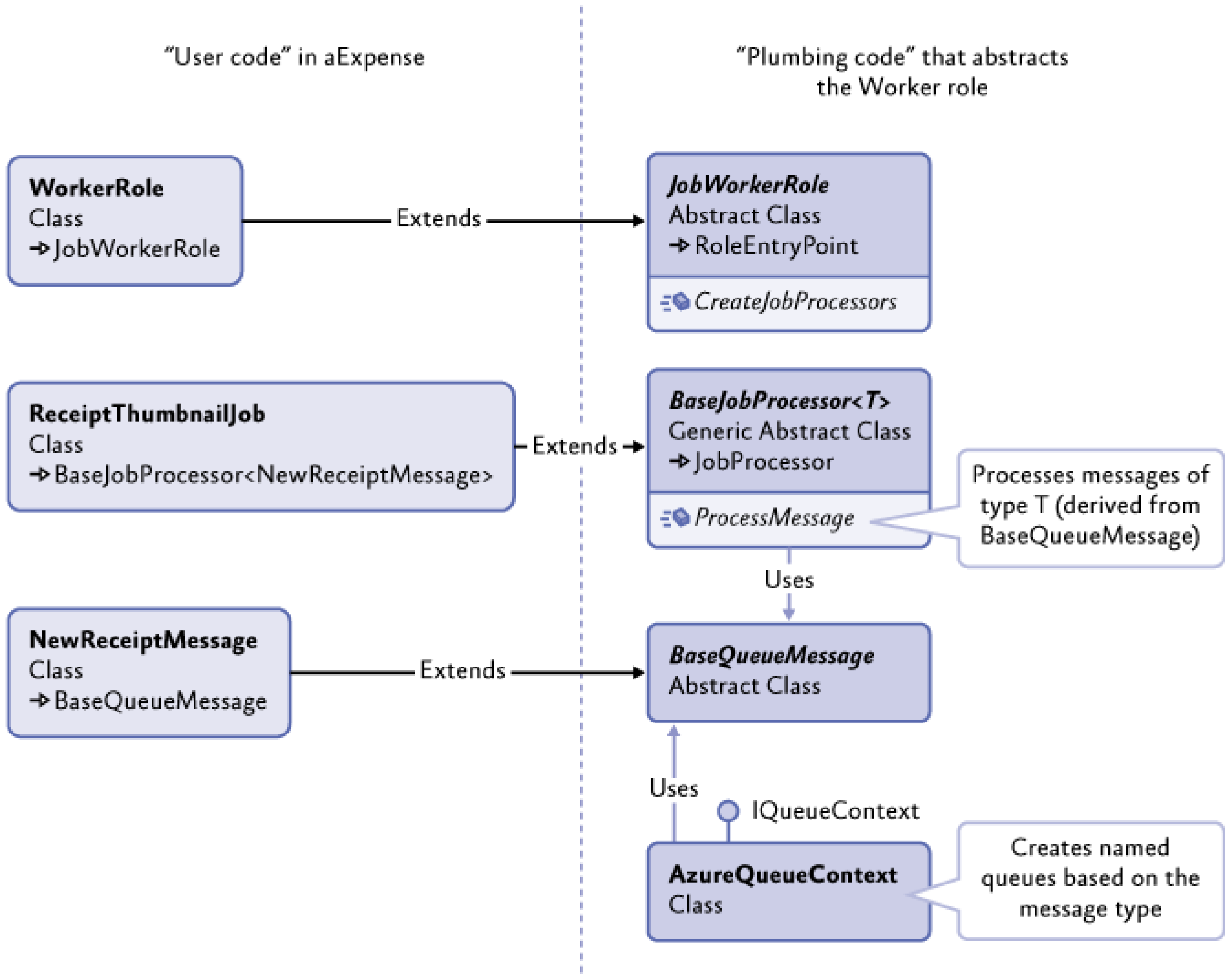


Figure 6

Relationship of "user code" to plumbing code

The "user code" classes are the ones that you will implement for each worker role and job type. The "plumbing code" classes are the re-usable elements. The "plumbing code" classes are packaged in the **AExpense.Jobs**, **AExpense.Queues**, and**AExpense.QueueMessages** namespaces. The following two sections first discuss the "user code" and then the "plumbing code."

Markus says:

For any new background task, you only need to implement the "user code" components.

"User Code" in the aExpense Application

The code you'll see described in this section implements the job that will compress images and generate thumbnails in the worker role for the aExpense application. What you should see in this section is how easy it is to define a new job type for the worker role to perform. This code uses the "plumbing code" that the following section describes in detail.

The following code shows how the application initializes the worker role using the "plumbing code": you create a new class that derives from the **JobWorkerRole** and override the **CreateJobProcessors** method. In this method, you instantiate your job processing objects that implement the **IJobProcessor** interface. As you can see, this approach makes it easy to plug in any additional job types that implement the **IJobProcessor** interface.

```
public class WorkerRole : JobWorkerRole
{
    protected override IEnumerable<IJobProcessor>
       CreateJobProcessors()
    {
        return new IJobProcessor[] { new receiptThumbnailJob() };
    }
}
```

This code does not match exactly what you'll find in the downloaded solution because Adatum changed this code in the subsequent phase of this project. For more information, see Chapter 8, "Phase 4: Adding More Tasks and Tuning the Application."

The constructor for the **ReceiptThumbnailJob** class specifies the interval the worker role uses to poll the queue and instantiates an **AzureQueueContext** object, an **ExpenseReceiptStorage** object, and an **ExpenseRepository** object for the worker role to use. The "plumbing code" passes a **NewReceiptMessage** object that contains the details of the image to process to the **ProcessMessage** method. This method then compresses the image referenced by the message and generates a thumbnail. The following code shows the constructor and the **ProcessMessage** method in the **ReceiptThumbnailJob** class.

Markus says:

Worker roles must poll the queue for new messages.

```
public ReceiptThumbnailJob()
    : base(2000, new AzureQueueContext())
{
    this.receiptStorage = new ExpenseReceiptStorage();
    this.expenseRepository = new ExpenseRepository();
}

public override bool ProcessMessage(NewReceiptMessage message)
```

```
{
    …
}
```

In the aExpense application, to send a message containing details of a new receipt image to the worker role, the web role creates a **NewReceiptMessage** object and calls the **AddMessage** method of the **AzureQueueContext** class. The following code shows the definition on the **NewReceiptMessage** class.

```
[DataContract]
public class NewReceiptMessage : BaseQueueMessage
{
    [DataMember]
    public string ExpenseItemId { get; set; }
}
```

It's important to use the **DataContract** and **DataMember** attributes in your message class because the **AzureQueueContext** class serializes message instances to the JSON format.

The last line in the **SaveExpense** method shown in the following code sample shows how the web role in aExpense puts a message onto the queue.

```
public void SaveExpense(Expense expense)
{
    var context = new ExpenseDataContext(this.account);
    ExpenseRow expenseRow = expense.ToTableEntity();

    foreach (var expenseItem in expense.Details)
    {
        // create row
        var expenseItemRow = expenseItem.ToTableEntity();
        expenseItemRow.PartitionKey = expenseRow.PartitionKey;
        expenseItemRow.RowKey =
          string.Format(CultureInfo.InvariantCulture, "{0}_{1}",
          expense.Id, expenseItemRow.Id);

        context.AddObject(ExpenseDataContext.ExpenseItemTable,
          expenseItemRow);

        // save receipt image if any
        if (expenseItem.Receipt != null &&
            expenseItem.Receipt.Length > 0)
        {
            this.receiptStorage.AddReceipt(
                expenseItem.Id.ToString(), expenseItem.Receipt,
                string.Empty);
        }
    }

    context.SaveChanges(SaveChangesOptions.Batch);
    context.AddObject(ExpenseDataContext.ExpenseTable,
        expenseRow);
    context.SaveChanges();

    var queue = new AzureQueueContext(this.account);
    expense.Details.ToList().ForEach(
        i => queue.AddMessage(new NewReceiptMessage
            { ExpenseItemId = i.Id.ToString() }));
}
```

The order of the operations in the **SaveExpense** method is important: because the detail and header records are stored in different tables, Adatum cannot use a transaction to guarantee data integrity; therefore, Adatum saves the details before the header. This way, if a failure occurs before, there won't be any header records without details and Adatum can implement a cleanup task to look for any orphaned details and receipt images. Orphaned detail and receipt images won't show up in the UI or in any exported data. Saving header records before details could potentially result in header records without details if a failure occurred, and the incomplete data would show up in the UI and in any exported data.

Markus says:

In Chapter 8, "Phase 4: Adding More Tasks and Tuning the Application," Adatum explores using a single table for both header and detail records so that it can use a transaction for the whole operation.

The second point to note about the order of the operations is that the method does not place any messages on the queue to notify the worker role of a new image to process until after all the records are saved. This way, there is no chance that the worker role will process an image before the associated record has been saved and fail because it can't find the record (remember that the worker role must update the URLs of the image and thumbnail in the detail record).

The "Plumbing Code" Classes

Adatum developed these abstractions to simplify the way that you send messages from a web role to a worker role and to simplify the way that you code a worker role. The idea is that you can put a typed message onto a queue, and when the worker role retrieves the message, it routes it to the correct job processor for that message type. The previous section described a job processor in the aExpense application that processes scanned receipt images and that uses these abstractions. The following core elements make up these "plumbing code" classes:

The "plumbing code" classes simplify the way that you send messages from a web role to a worker role and the way that you implement a worker role.

Jana says:

Adatum expect to implement additional background processes, so it makes sense to create this "plumbing code."

- A wrapper for the standard Windows Azure worker role's **RoleEntryPoint** class that abstracts the worker role's life cycle and threading behavior.
- A customizable job processor that enables users of the "plumbing code" classes to define their own job types for the worker role.
- A wrapper for the standard Windows Azure **CloudQueue** class that implements typed messages to enable message routing within the **JobWorkerRole** class.

Figure 7 summarizes how the "plumbing code" classes handle messages derived from the **BaseQueueMessage** class.

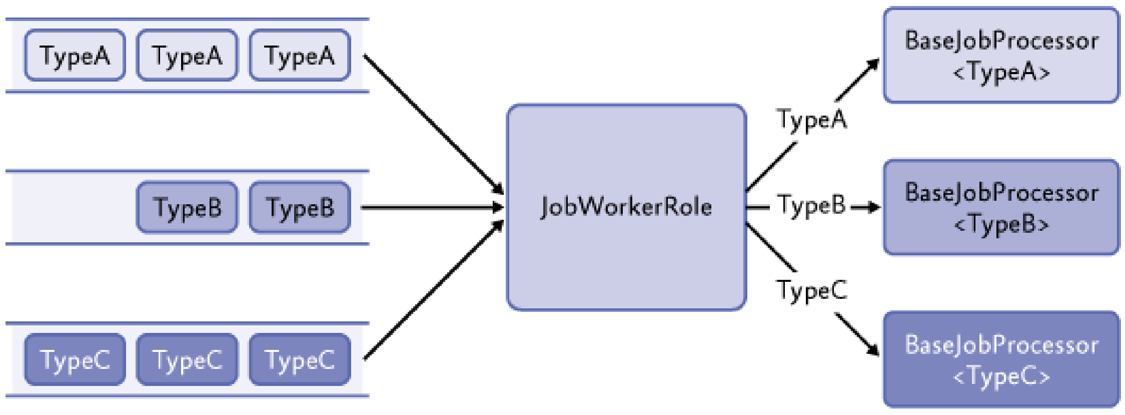


Figure 7
Worker role "plumbing code" elements

The message types that the "plumbing code" classes handle (like the **NewReceiptMessage** type in aExpense) are derived from the **BaseQueueMessage** class shown in the following code example.

```
[DataContract]
public abstract class BaseQueueMessage
{
    private object context;

[System.Diagnostics.CodeAnalysis.SuppressMessage(…)
    public object GetContext()
    {
        return this.context;
    }
    public void SetContext(object value)
    {
        this.context = value;
    }
}
```

The "plumbing code" classes use the **AzureQueueContext** class to deliver messages to the worker role. The **AzureQueueContext** class creates named queues based on the message types, one queue for each message type that was registered with it. The following code shows the **Add** method in the **AzureQueueContext** class that you use to add a new message to a queue and the **ResolveQueueName** method that figures out the name of the queue to use.

```
public void AddMessage(BaseQueueMessage message)
{
    var queueName = ResolveQueueName(message.GetType());
    this.EnsureQueueExists(queueName);

    var json = Serialize(message.GetType(), message);
    var queue = this.queue.GetQueueReference(queueName);
    queue.AddMessage(new CloudQueueMessage(json));
}

public static string ResolveQueueName(MemberInfo messageType)
{
    return messageType.Name.ToLowerInvariant();
}
```

There are two other details to point out in the **AddMessage** method implementation. First, the "plumbing code" serializes messages to the JSON format, which typically produces smaller message sizes than an XML encoding (but possibly larger than a binary encoding). Second, the **EnsureQueueExists** method calls the **CreateIfNotExist** method of the Windows Azure **CloudQueue** class. Calling the **CreateIfNotExist** method counts as a storage transaction and will add to your application's running costs.

Poe says:

If you are concerned about the running costs of the application, you should be aware of which calls in your code are chargeable!

Microsoft currently bills you for storage transactions at $0.01/100 K. If you have high volumes of messages, you should check how frequently your application calls this method.

In Chapter 8, "Phase 4: Adding More Tasks and Tuning the Application," the place where the application calls the **CreateIfNotExist** is revaluated following the results from performance testing on the application.

The "plumbing code" classes deliver messages to job processor components, where a job processor handles a specific message type. The "plumbing code" classes include an interface named **IJobProcessor** that defines two void methods named **Run** and **Stop** for starting and stopping a processor. The abstract **BaseJobProcessor** and **JobProcessor** classes implement this interface. In the aExpense application, the **ReceiptThumbnailJob**class that you've already seen extends the**BaseJobProcessor**class. The following code example shows how the **JobProcessor** class implements the **IJobProcessor** interface.

```
private bool keepRunning;

public void Stop()
{
    this.keepRunning = false;
}

public void Run()
{
    this.keepRunning = true;
    while (this.keepRunning)
    {
Thread.Sleep(this.SleepInterval);
        this.RunCore();
    }
}

protected abstract void RunCore();
```

The following code example shows how the **BaseJobProcessor** class provides an implementation of the **RunCore** method.

```
protected bool RetrieveMultiple { get; set; }
protected int RetrieveMultipleMaxMessages { get; set; }

protected override void RunCore()
{
    if (this.RetrieveMultiple)
    {
        var messages = this.Queue.GetMultipleMessages<T>
            (this.RetrieveMultipleMaxMessages);
        if (messages != null)
        {
            foreach (var message in messages)
            {
                this.ProcessMessageCore(message);
            }
        }
        else
        {
            this.OnEmptyQueue();
        }
    }
    else
    {
        var message = this.Queue.GetMessage<T>();
        if (message != null)
        {
            this.ProcessMessageCore(message);
        }
        else
        {
            this.OnEmptyQueue();
        }
    }
}
```

As you can see, the **RunCore** method can retrieve multiple messages from the queue in one go. The advantage of this approach is that one call to the **GetMessages** method of the Windows Azure **CloudQueue** class only counts as a single storage transaction, regardless of the number of messages it retrieves. The code example also shows how the **BaseJobProcessor** class calls the generic **GetMessage** and **GetMultipleMessages** of the **AzureQueueContext** class specifying the message type by using a generic type parameter.

Bharath says:

It's cheaper and more efficient to retrieve multiple messages in one go if you can. However, these benefits must be balanced against the fact that it will take longer to process multiple messages; this risks the messages becoming visible to other queue readers before you delete them.

The following code example shows how the **BaseJobProcessor** constructor assigns the job's polling interval and the **AzureQueueContext** reference.

```
protected BaseJobProcessor(int sleepInterval,
    IQueueContext queue) : base(sleepInterval)
{
    if (queue == null)
    {
        throw new ArgumentNullException("queue");
    }

    this.Queue = queue;
}
```

The remaining significant methods in the **BaseJobProcessor** class are the **ProcessMessageCore** and the abstract **ProcessMessage** methods shown below.

```
protected int MessagesProcessed { get; set; }

private void ProcessMessageCore(T message)
{
    var processed = this.ProcessMessage(message);
    if (processed)
    {
        this.Queue.DeleteMessage(message);
        this.MessagesProcessed++;
    }
}

public abstract bool ProcessMessage(T message);
```

The **RunCore** method invokes the **ProcessMessageCore** method when it finds new messages to process. The **ProcessMessageCore** method then calls the "user-supplied" implementation of the **ProcessMessage** method before it deletes the message from the queue. In the aExpense application, this implementation is in the **ReceiptThumbnailJob** class.

The final component of the "plumbing code" is the abstract **JobWorkerRole** class that wraps the standard Windows Azure **RoleEntryPoint** class for the worker role. The following code example shows the **Run** method in this class.

```
protected IEnumerable<IJobProcessor> Processors { get; set; }

protected abstract IEnumerable<IJobProcessor>
 CreateJobProcessors();

public override void Run()
{
    this.Processors = this.CreateJobProcessors();

    var threads = new List<Thread>();

    foreach (var processor in this.Processors)
    {
        var t = new Thread(processor.Run);
        t.Start();
        threads.Add(t);
    }

    foreach (var thread in threads)
    {
        thread.Join();
    }
}
```

The **Run** method invokes the abstract **CreateJobProcessors** method that is implemented in "user code." In the aExpense application, you can find this implementation in the **WorkerRole** class. The **Run** method then creates a new thread for each job processor and then waits for all the threads to finish.

Markus says:

You need to keep the worker role alive!

Processing the Images

The following code example shows how the aExpense application implements the image processing functionality in the **ProcessMessage** method in the **ReceiptThumbnailJob** class.

```
public override bool ProcessMessage(NewReceiptMessage message)
{
    var expenseItemId = message.ExpenseItemId;
    var imageName = expenseItemId + ".jpg";

    byte[] originalPhoto =
        this.receiptStorage.GetReceipt(expenseItemId);

    if (originalPhoto != null && originalPhoto.Length > 0)
    {
        var thumb = ResizeImage(originalPhoto, ThumbnailSize);
        var thumbUri =
            this.receiptStorage.AddReceipt(Path.Combine(
            "thumbnails", imageName), thumb, "image/jpeg");

        var photo = ResizeImage(originalPhoto, PhotoSize);
        var photoUri = this.receiptStorage.AddReceipt(imageName,
            photo, "image/jpeg");

        this.expenses.UpdateExpenseItemImages(expenseItemId,
             photoUri, thumbUri);

        this.receiptStorage.DeleteReceipt(expenseItemId);

        return true;
    }

    return false;
}
```

This method retrieves the image name from the message sent to the worker role, and then it creates two new versions of the image, one a thumbnail, and one a "standard" size. It then deletes the original image. The method can process images in any standard format, but it always saves images as JPEGs.

Jana says:

Although we limit uses to uploading images that are less than 1,024 KB in size, in order to save space we decided not to store the original images. We found that resizing to a standard size provided acceptable quality.

The **ProcessMessage** method should be idempotent, so there are no unwanted side effects if a message is delivered multiple times. The **ProcessMessage** method should also contain some logic to handle "poison" messages that cannot be processed for some reason.

Making the Images Available Using Shared Access Signatures

To make images of receipts viewable in the UI, the team at Adatum used Shared Access Signatures (SAS) to generate short-lived, secure URLs to address the images in blob storage. This approach avoids having to give public access to the blob container and minimizes the amount of work that the web server has to perform because the client can access the image directly from blob storage.

The aExpense application uses Shared Access Signatures to provide limited access to blobs in private containers.

The following code example shows how the application generates the SAS URLs in the **GetExpenseByID** method in the **ExpenseRepository** class by appending the SAS to the blob URL. The aExpense application uses an HTTPS endpoint, so the blob reference and signature elements of the blob's URL are protected by SSL from "man-in-the-middle" attacks.

Using SSL ensures that all URL data except for the hostname is encrypted.

```
CloudBlob receiptBlob =
    container.GetBlobReference(item.ReceiptUrl.ToString());
item.ReceiptUrl = new Uri(item.ReceiptUrl.AbsoluteUri +
    receiptBlob.GetSharedAccessSignature(policy));
```

Markus says:

The Shared Access Signature is appended to the standard blob URL.

The **GetSharedAccessSignature** method takes a **SharedAccessPolicy** object as a parameter. This policy specifies the access permissions and the lifetime of the generated URL. The following code shows the policy that the aExpense application uses, granting read permission for one minute to an image. The application generates a new SAS whenever a user tries to access an expense submission.

```
private readonly TimeSpan sharedSignatureValiditySpan;
```

```
var policy = new SharedAccessPolicy
{
    Permissions = SharedAccessPermissions.Read,
    SharedAccessExpiryTime = DateTime.UtcNow +
        this.sharedSignatureValiditySpan
};
```

The application does not specify a value for the **SharedAccessStartTime** property of the **SharedAccessPolicy** object. Setting this value to the current time can cause problems if there is a clock skew between the client and the server and you try to access the blob immediately.

As long as the **Get** request for a blob starts before the expiry time, the request will succeed, even if the response streaming continues past the expiration time. As a part of the logical **Get** operation, if your application chooses to retry on a **Get** failure, which is the default StorageClient library policy, any retry request made after the expirationtime will fail. If you decide to have a short validity period for the URL, make sure that you issue a single **Get** request for the entire blob and use a custom retry policy so that when you retry the request, you get a new SAS for the URL.

More Information

MSDN® contains plenty of information about Windows Azure blob storage. A good starting place is at http://msdn.microsoft.com/en-us/library/dd135733.aspx.

To find out more about controlling access to Azure storage, including shared access signatures, look at http://msdn.microsoft.com/en-us/library/ee393343.aspx.

You can find a summary of the Windows Azure service architecture at http://msdn.microsoft.com/en-us/library/gg432976.aspx.

7 – Application Life Cycle Management for Windows Azure Applications

This chapter outlines an application life cycle management approach for Windows® Azure™ technology platform applications. Although specific requirements will vary between applications and across organizations, everyone develops applications, then tests them, and finally deploys them. This chapter focuses on where applications should be tested, and how the deployment process should be managed to make sure that only authorized personnel have access to the live production environment.

The Premise

Adatum have a well-defined set of processes for deploying applications to their on-premises servers. They use separate servers for testing, staging, and production. When the development team releases a new version of the application, its progression through the testing, staging, and production servers is tightly controlled. Very rarely, though, small fixes, such as updating the text on an ASPX page, are applied directly to the production servers.

Goals and Requirements

Adatum has a number of goals for application life cycle management for Windows Azure. Adatum wants to have a clearly defined process for deploying applications to Windows Azure, with clearly defined roles and responsibilities. More specifically, it wants to make sure that access to the live environment is only available to a few key people, and that any changes to the live environment are traceable.

You should have a clearly defined process for deploying applications to Windows Azure.

In addition, Adatum wants to be able to roll back the live environment to a previous version if things go wrong. In a worst-case scenario, they would like to be able to pull the application back to be an on-premises application.

Adatum would like to perform some testing on an application in an environment that is as similar as possible to the live environment.

Bharath says:

All Windows Azure environments in the cloud are the same; there's nothing special about a test or staging area. However, there are differences between the local Compute Emulator and the cloud environment, which is why it's important to test your application in the cloud.

Finally, Adatum would like to minimize the costs of maintaining all the required environments and be able to identify the costs of running the development and test environments separately from the live production environment.

Don't forget that any time you do something in Windows Azure, even if it's only testing it costs money.

Overview of the Solution

Adatum can use the local Compute Emulator and Storage Emulator for all development tasks and for a great deal of testing before it deploys anything to Windows Azure. However, Adatum must test its applications in the cloud before it deploys them to the live production environment.

To achieve this goal, Adatum has two Windows Azure subscriptions. One is an account used for testing, and one is the live production account. Because each account has its own Windows Live® ID, and its own set of API keys, Adatum can limit access to each environment to a particular set of personnel. Members of the testing team and key members of the development team have access to the testing account. Only two key people in the Adatum operations department have access to the production account.

Both of these accounts are standard Windows Azure accounts, and because they are identical environments, Adatum can be confident that application code will run in the same way on both of them. Adatum can also be sure that application deployment will work in the same way in both environments because it can use the same package to deploy to both test and production.

Bharath says:

In Windows Azure, you can be sure that different accounts have identical environments—this is something that's very difficult to guarantee in your on-premises environments.

Adatum can also perform some testing by running the application on the Compute Emulator, but pointing the application at storage in the test Windows Azure environment. This is important because there are more differences between development storage and cloud storage than between the Compute Emulator and the cloud runtime environment. Also, cloud storage is relatively cheap to use as compared to other cloud resources.

Poe says:

Because development and testing staff don't have access to the production storage account keys, there's no risk of accidentally testing on live data.

Microsoft bills Adatum separately for the two environments, which makes it easy for Adatum to separate the running costs for the live environment from the costs associated with development and test. This allows Adatum to manage the product development budget separately from the operational budget, in a manner similar to the way it manages budgets for on-premises applications.

Setup and Physical Deployment

Figure 1 summarizes the application life cycle management approach at Adatum.

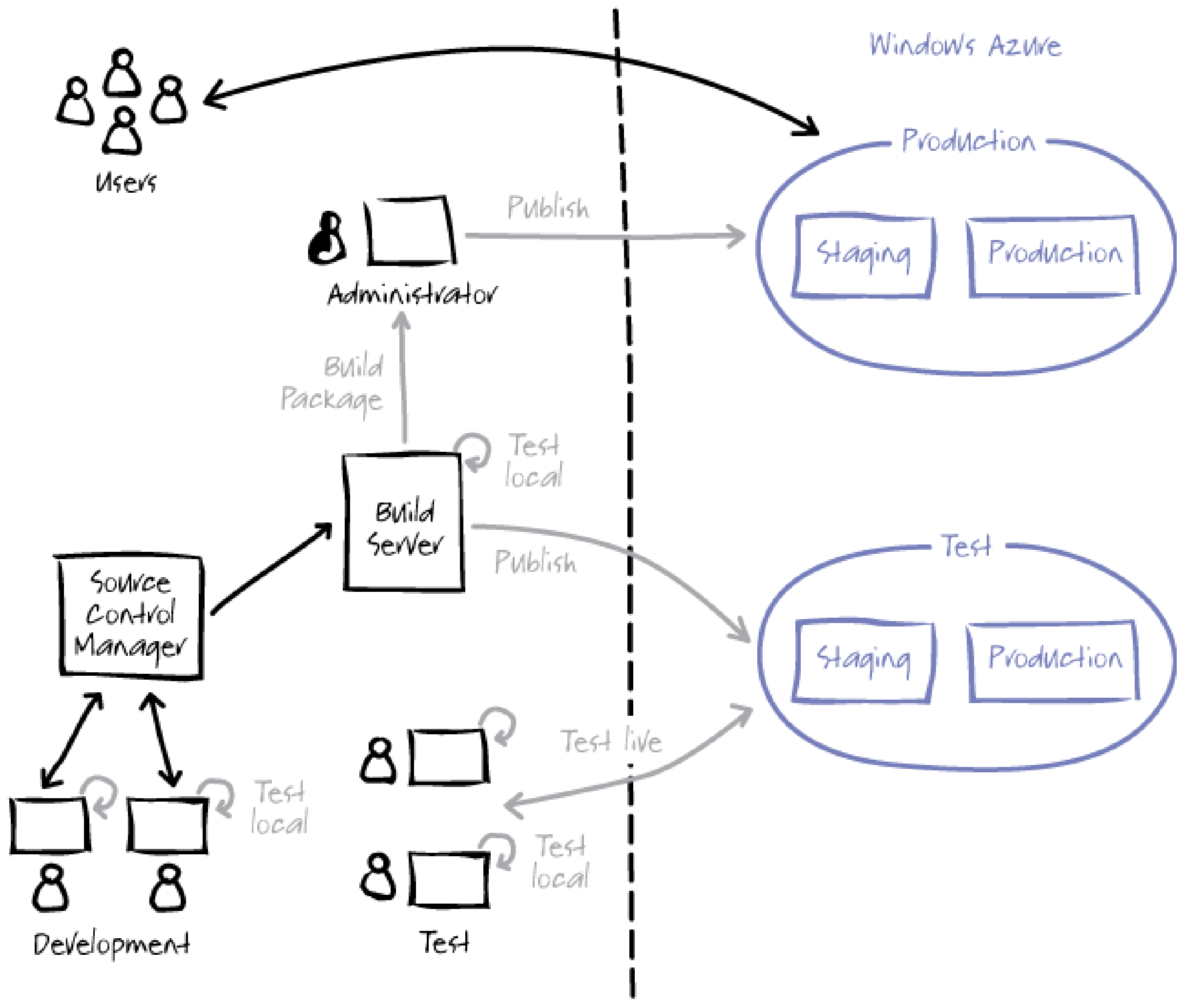


Figure 1

Adatum's application life cycle management environment

The preceding diagram shows the two separate Windows Azure environments that Adatum uses for test and production, as well as the on-premises environment that consists of development computers, test computers, a build server, and a source code management tool.

Windows Azure Environments

There are two ways to access a Windows Azure environment to perform configuration and deployment tasks. The first is through the Windows Azure Management Portal, where a single Windows Live ID has access to everything in the portal. The second is by using the Windows Azure Service Management API, where API certificates are used to access to all the functionality exposed by the API. In both cases, there is currently no way to restrict users to only be able to manage a subset of the available functionality, for example to only be able to manage the storage aspects of the account. Within Adatum, almost all operations that affect the test or production environment are performed using scripts instead of the Management Portal.

Poe says:

We use scripts because it means that we can reliably repeat operations. We're also planning to enhance some of our scripts to record when they were run, by whom, and what parameters were used.

For information about the Windows Azure Service Management API, see Chapter 5, "Phase 2: Automating Deployment and Using Windows Azure Storage," of this guide.

Within Adatum, only key developers and key members of the testing team have access to the test Windows Azure environment. Only two people in the operations department have access to the production Windows Azure environment.

The Windows Live ID and API certificates used to access the test environment are different from the ones used to access the production system.

Production and Staging Areas

In Windows Azure, you can deploy an application to either a staging or a production area. A common scenario is to deploy first to the staging area and then, at the appropriate time, move the new version to the production area. The two areas are identical environments; the only difference is in the URL you use to access them. In the staging area, the URL will be something obscure like http://bb7ea70c3be24eb08a08b6d39f801985.cloudapp.net,while in the production area you will have a friendly URL like http://aexpense.cloupapp.net. This allows you to test new and updated applications in a private environment that others don't know about.You can also swap the contents of the production and staging areas, which makes it easy to roll back the application to the previous version.

You can quickly roll back a change in production by using the swap operation.

Bharath says:

Don't forget, Microsoft will bill you for using the staging area as well as for using the production area.

The swap is nearly instantaneous because it just involves Windows Azure changing the DNS entries for the two areas.

For information about rolling upgrades across multiple role instances and partial application upgrades that upgrade individual roles, see the MSDN documentation at http://msdn.microsoft.com/en-us/library/ee517254.aspx.

Deployment

To deploy an application to Windows Azure you must upload two files: the service package file that contains all your application's files and the service configuration file that contains your application's configuration data. You can generate the service package file either by using the Cspack.exe utility or by using Visual Studio if you have installed the Windows Azure SDK.

Markus says:

We use the command-line utility on the build server to automate the generation of our deployment package and configuration file.

Adatum uses the same package file to deploy to the test and production environments, but they do modify the configuration file. For the aExpense application, the key difference between the contents of the test and production configuration files is the storage connection strings. This information is unique to each Windows Azure subscription and uses randomly generated access keys. Only the two key people in the operations department have access to the storage access keys for the production environment, which makes it impossible for anyone else to use production storage accidentally during testing. Adatum uses a set of scripts to perform the deployment, and one of the tasks of these scripts is to replace the local development storage connection strings in the configuration file with the specific connection strings for the cloud-based storage.

For more information about the deployment scripts, see Chapter 5, "Phase 2: Automating Deployment and Using Windows Azure Storage."

Testing

Adatum tests its cloud-based applications in a number of different environments before it deploys the application to the live production environment.

Developers run unit tests and ad-hoc tests on the Compute Emulator running on their local computers. Although the Compute Emulator is not identical to the cloud environment, it is suitable for developers to run tests on their own code. The build server also runs a suite of tests as a part of the standard build process. This is no different from the normal development practices for on-premises applications.

Most testing can be performed using the Compute Emulator and Storage Emulator.

The testing team performs the majority of its tests using the local Compute Emulator as well. They only deploy the application to the Windows Azure test environment to test the final configuration of the application before it is passed to the admin team for deployment to production. This way, they can minimize the costs associated with the test environment by limiting the time that they have an application deployed in the cloud.

Jana says:

You can deploy an application to your Windows Azure test environment just while you run the tests. You only get charged when you use the environment. However, you are charged by the hour, so a 15-minute test will cost the same as a 55-minute test.

More Information

MSDN® contains plenty of information about Windows Azure. A good starting to learn more about deploying applications to Windows Azure is at http://msdn.microsoft.com/en-us/library/dd163896.aspx.

The Windows Azure Developer Center (http://www.windowsazure.com/en-us/develop/overview/) contains links to plenty of resources to help you learn about managing applications in Windows Azure.

8 – Phase 4: Adding More Tasks and Tuning the Application

This chapter walks you through the data export feature that Adatum added to the aExpense application, and some of the changes they made following performance testing. The chapter revisits the worker role abstractions introduced in the last phase of the project to illustrate how you can run multiple tasks within a single worker role. It also discusses some architectural issues such as session state management, and the choice of partition and row keys for Windows® Azure™ technology platform table storage.

The Premise

In this phase of the aExpense migration project, the team at Adatum implemented the last key piece of functionality in the application. The aExpense application must generate a file of data that summarizes the approved business expense submissions for a period. Adatum's on-premises payments system imports this data file and then makes the payments to Adatum employees. The final on-premises version of aExpense uses a scheduled SQL Server® Integration Services job to generate the output file and sets the status of an expense submission to "processing" after it is exported. The on-premises application also imports data from the payments processing system to update the status of the expense submissions after the payment processing system makes a payment. This import process is not included in the current phase of the migration project. Figure 1 summarizes the current export process in the on-premises application.

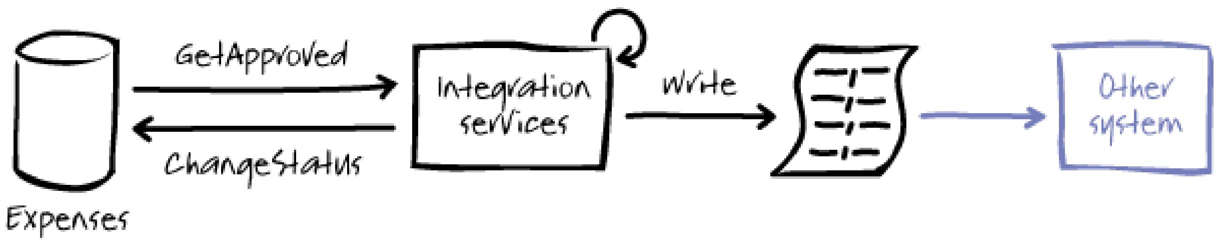


Figure 1
aExpense export process

In this phase, Adatum also evaluated the results from performance testing the application and implemented a number of changes based on those results.

Adatum made a number of changes to aExpense following the performance test on the application.

Goals and Requirements

The design of the export process for the cloud version of aExpense must meet a number of goals. First, the costs of the export process should be kept to a minimum while making sure that the export process does not have a negative impact on the performance of the application for users. The export process must also be robust and be able to recover from a failure without compromising the integrity of aExpense's data or the accuracy of the exported data.

Markus says:

Approved business expense submissions could be anywhere in the table. We want to try to avoid the performance impact that would result from scanning the entire table.

The solution must also address the question of how to initiate the export and evaluate whether it should be a manual operation or a scheduled operation. If it is the latter, the team at Adatum must design a mechanism for scheduling tasks in Windows Azure.

The final requirement is to include a mechanism for transferring the data from the cloud-environment to the on-premises environment where the payment processing application can access it.

Poe says:

Adatum must automate the process of downloading the expense submission report data for input into the payments processing system.

Overview of the Solution

There are three elements of the export process to consider: how to initiate the process, how to generate the data, and how to download the data from the cloud.

Initiating the Data Export Process

The simplest option for initiating the data export is to have a web page that returns the data on request. There are some potential disadvantages to this approach: First, it adds to the web server's load and potentially affects the other users of the system. In the case of aExpense, this will probably not be significant because the computational requirements for producing the report are low. Second, if the process that generates the data is complex and the data volumes are high, the web page must be able to handle timeouts. Again, for aExpense, it is unlikely that this will be a significant problem. The most significant drawback to this solution in aExpense is that the current storage architecture for expense submission data is optimized for updating and retrieving individual expense submissions by using the user ID. The export process will need to access expense submission data by date and expense state. Unlike Windows Azure SQL Database where you can define multiple indexes on a table, Windows Azure table storage currently only has a single index on each table.

Jana says:

Choosing the right PartitionKey and RowKey for your tables is crucial for the performance of your application. Any process that needs a "tablescan" across all your partitions will be slow.

Figure 2 illustrates the second option for initiating the data export. Each task has a dedicated worker role, so the image compression and thumbnail generation would be handled by Task 1 in Worker 1, and the data export would be performed by Task 2 in Worker 2. This would also be simple to implement, but in the case of aExpense where the export process will run twice a month, it's not worth the overhead of having a separate role instance. If your task ran more frequently and if it was computationally intensive, you might consider an additional worker role.

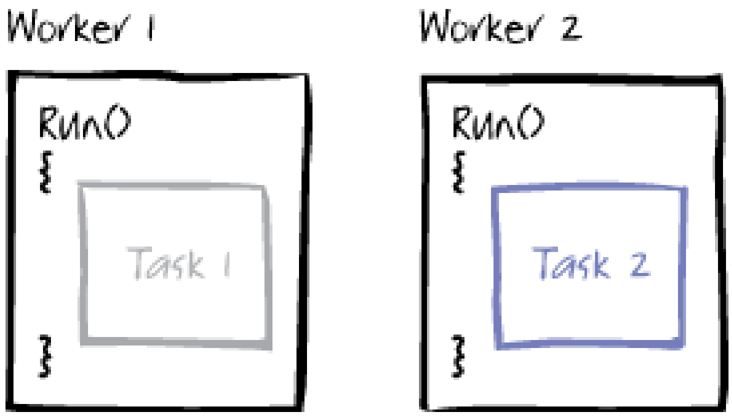


Figure 2
Separate worker roles for each task

Bharath says:

You should try to use your compute nodes to the full. Remember, you pay for a deployed role instance whether or not it's doing any work. You can also opt for a larger compute instance if you want to do more work in a single role instance.

Figure 3 illustrates the third option where an additional task inside an existing worker role performs the data export process. This approach makes use of existing compute resources and makes sense if the tasks are not too computationally intensive. At the present time, the Windows Azure SDK does not include any task abstractions, so you need to either develop or find a framework to handle task-based processing for you. The team at Adatum will use the "plumbing code" classes described in Chapter 6, "Phase 3: Uploading Images and Adding a Worker Role," to define the tasks in the aExpense application. Designing and building this type of framework is not very difficult, but you do need to include all your own error handling and scheduling logic.

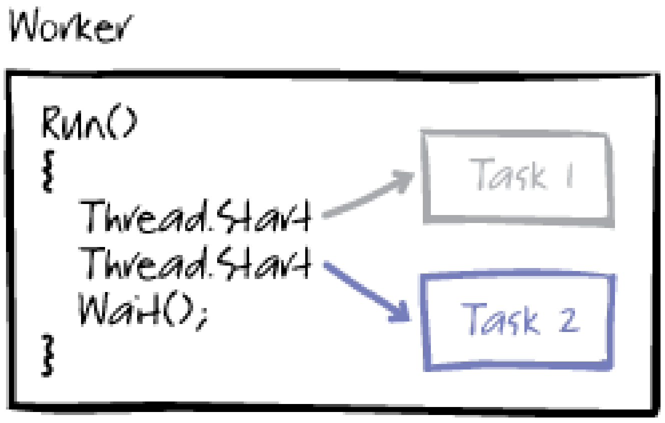


Figure 3

Multiple tasks in a single worker role

Adatum already has some simple abstractions that enable them to run multiple tasks in a single worker role.

Markus says:

Windows Azure can only monitor at the level of a worker, and it tries to restart a failed worker if possible. If one of your task processing threads fails, it's up to you to handle the situation.

Generating the Export Data

The team at Adatum decided to split the expense report generation process into two steps. The first step "flattens" the data model and puts the data into a Windows Azure table. This table uses the expense submission's approval date as the partition key, the expense ID as the row key, and it stores the expense submission total. The second step reads this table and generates a Windows Azure blob that contains the data ready for export as a Comma Separated Values (CSV) file. Adatum implemented each of these two steps as a task by using the "plumbing code" described in Chapter 6, "Phase 3: Uploading Images and Adding a Worker Role." Figure 4 illustrates how the task that adds data to the Windows Azure table works.

Bharath says:

Windows Azure table storage does not have all the features of a relational database. You may need multiple tables that present the same data in different ways based on the needs of the application. To make this work, you should be able to generate the derived table in an idempotent way from the data in the original table. Table storage is cheap!

First, a manager approves a business expense submission. This places a message that contains the expense submission's ID and approval date onto a queue (1), andupdates the status of the submission in table storage (2). The task retrieves the message from the queue, calculates the total value of the expense submission from the expense detail items, and stores this as a single line in the Expense Export table. The task also updates the status of the expense submission to be "in process" before it deletes the message from the queue.

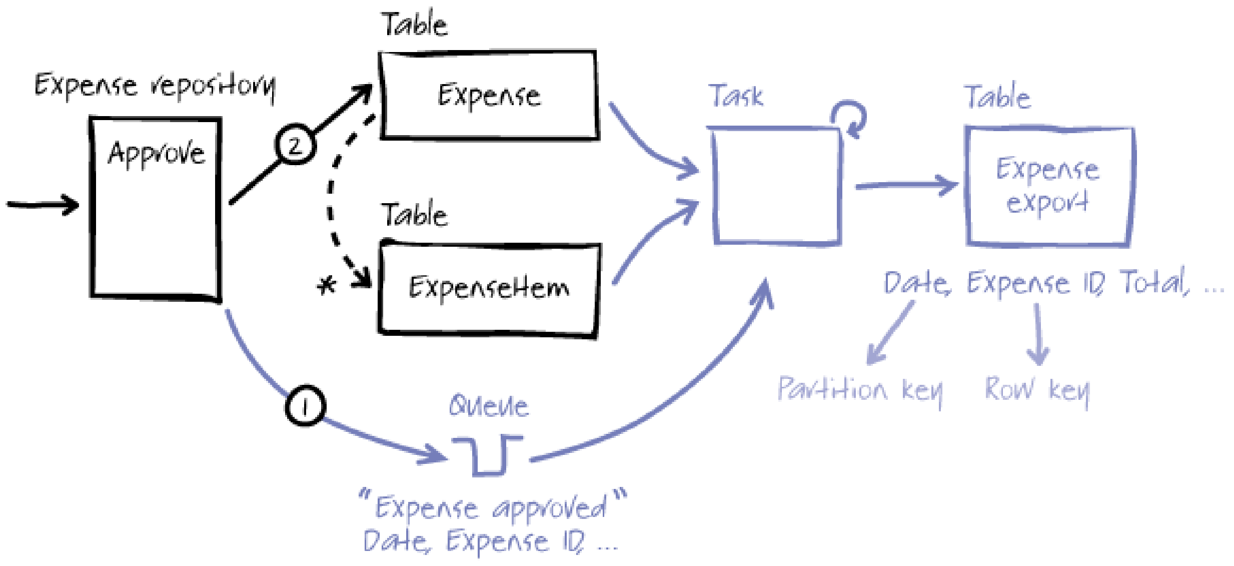


Figure 4

Generating the Expense Report table

Exporting the Report Data

To export the data, Adatum considered two options. The first was to have a web page that enables a user to download the expense report data as a file. This page would query the expense report table by date and generate a CSV file that the payments processing system can import. Figure 5 illustrates this option.

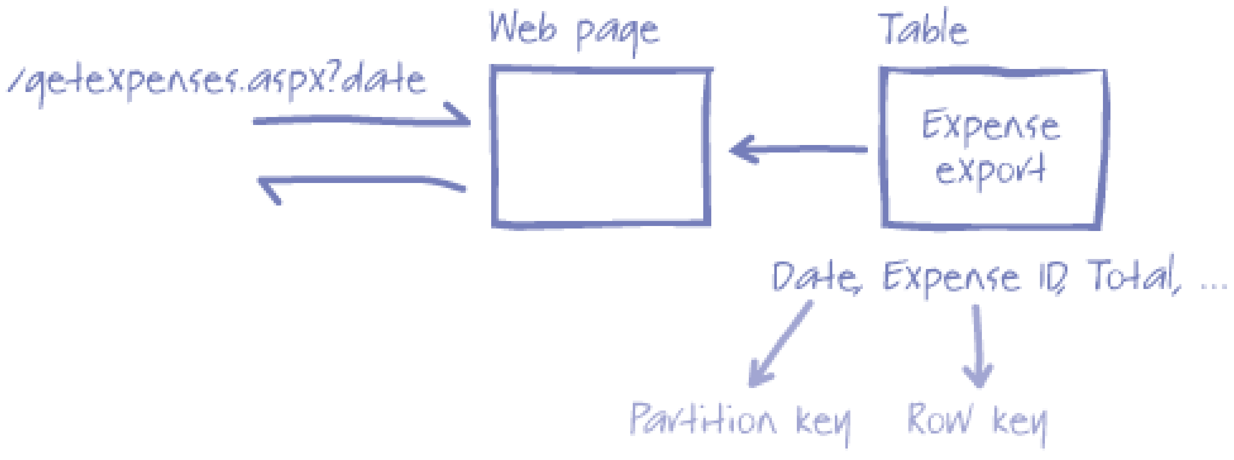


Figure 5

Downloading the expense report from a web page

The second option, shown in Figure 6, was to create another job in the worker process that runs on a schedule to generate the file in blob storage ready for download. Adatum will modify the on-premises payment processing system to download this file before importing it. Adatum selected this option because it enables them to schedule the job to run at a quiet time in order to avoid any impact on the performance of the application for users. The on-premises application can access the blob storage directly without involving either the Windows Azure web role or worker role.

Poe says:

This approach makes it easier for us to automate the download and get the data in time for the payments processing run.

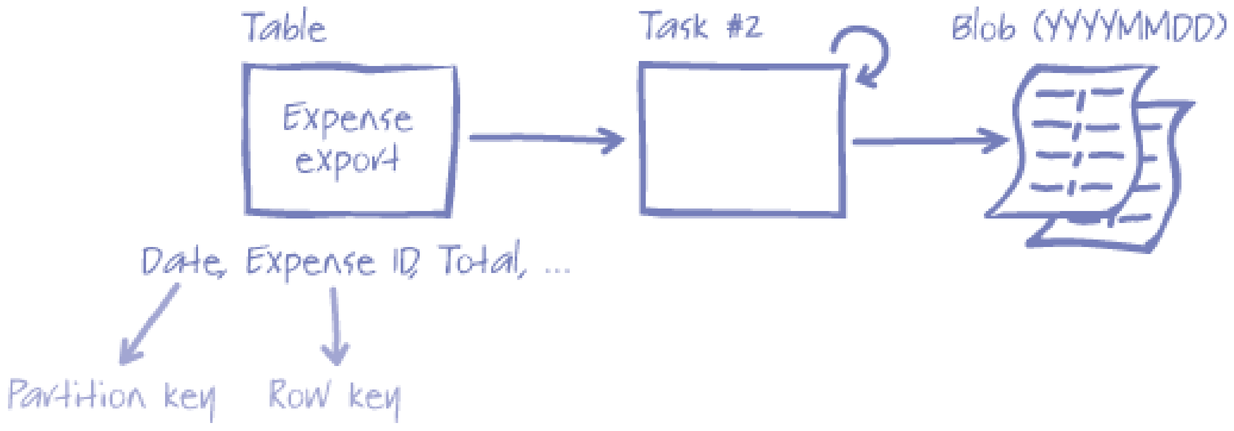


Figure 6

Generating the expense report in blob storage

Adatum had to modify slightly the worker role "plumbing code" to support this process. In the original version of the "plumbing code," a message in a queue triggered a task to run, but the application now also requires the ability to schedule tasks.

Markus says:

We had to modify our "plumbing code" classes slightly to accommodate scheduled tasks.

Inside the Implementation

Now is a good time to walk through these changes in more detail. As you go through this section, you may want to download the Microsoft® Visual Studio® development system solution from http://wag.codeplex.com/. This solution contains the implementation of aExpense after the changes made in this phase. If you are not interested in the mechanics, you should skip to the next section.

Generating the Expense Report Table

The task that performs this operation uses the worker role "plumbing code" described in Chapter 6, "Phase 3: Uploading Images and Adding a Worker Role." The discussion here will focus on the task implementation and table design issues; it does not focus on the "plumbing code."

Markus says:

For this task, we were able to use our worker role "plumbing code" without modification.

This task is the first of two tasks that generate the approved expense data for export. It is responsible for generating the "flattened" table of approved expense data in Windows Azure table storage. The following code sample shows how the expense report export process begins in the **ExpenseRepository** class where the **UpdateApproved** method adds a message to a queue and updates the **Approved** property of the expense header record.

```
public void UpdateApproved(Expense expense)
{
    var context = new ExpenseDataContext(this.account);

    ExpenseRow expenseRow =
        GetExpenseRowById(context, expense.Id);
    expenseRow.Approved = expense.Approved;

    var queue = new AzureQueueContext(this.account);
    queue.AddMessage(new ApprovedExpenseMessage {
        ExpenseId = expense.Id.ToString(),
        ApproveDate = DateTime.UtcNow });

    context.UpdateObject(expenseRow);
    context.SaveChanges();
}
```

This code uses a new message type named **ApprovedExpenseMessage** that derives from the "plumbing code" class named **BaseQueueMessage**. The following code sample shows the two properties of the **ApprovedExpenseMessage** class.

```
[DataContract]
public class ApprovedExpenseMessage : BaseQueueMessage
{
    [DataMember]
    public string ExpenseId { get; set; }

    [DataMember]
    public DateTime ApproveDate { get; set; }
}
```

The following code shows how the **ProcessMessage** method in the **ExpenseExportJob** class retrieves the message from the queue and creates a new **ExpenseExport** entity to save to table storage.

Jana says:

We "flatten" the data and calculate the expense submission total before saving the data into an intermediate table. This table contains the data structured exactly as we need it to do the export.

```
public override bool ProcessMessage(
ApprovedExpenseMessage message)
{
    try
    {
        Expense expense = this.expenses.GetExpenseById(
new ExpenseKey(message.ExpenseId));

        if (expense == null)
        {
            return false;
        }

        // If the expense was not updated but a message was
        // persisted, we need to delete it.
        if (!expense.Approved)
        {
            return true;
        }

        double totalToPay = expense.Details.Sum(x => x.Amount);
        var export = new ExpenseExport
            {
ApproveDate = message.ApproveDate,
              ApproverName = expense.ApproverName,
              CostCenter = expense.CostCenter,
              ExpenseId = expense.Id,
              ReimbursementMethod = expense.ReimbursementMethod,
              TotalAmount = totalToPay,
              UserName = expense.User.UserName
             };
        this.expenseExports.Save(export);
    }
    catch (InvalidOperationException ex)
    {
        var innerEx =
 ex.InnerException as DataServiceClientException;
        if (innerEx != null &&
 innerEx.StatusCode == (int)HttpStatusCode.Conflict)
```

```
        {
// The data already exists, so we can return true
            // because we have processed this before.
            return true;
        }
        Log.Write(EventKind.Error, ex.TraceInformation());
        return false;
    }

    return true;
}
```

If this method fails for any reason other than a conflict on the insert, the "plumbing code" classes ensure that message is left on the queue. When the **ProcessMessage** method tries to process the message from the queue a second time, the insert to the expense report table fails with a duplicate key error and the inner exception reports this as a conflict in its **StatusCode** property. If this happens, the method can safely return a **true** result.

Jana says:

We need to ensure that this process is robust. We don't want to lose any expense submissions or pay anyone twice.

If the **Approved** property of the **Expense** object is false, this indicates a failure during the **UpdateApproved** method after it added a message to the queue, but before it updated the table. In this circumstance, the **ProcessMessage** method removes the message from the queue without processing it.

The partition key of the Expense Export table is the expense approval date, and the row key is the expense ID. This optimizes access to this data for queries that use approval date in the **where** clause, which is what the export process requires.

Bharath says:

Choose partition keys and rows keys to optimize your queries against the data. Ideally, you should be able to include the partition key in the **where** block of the query.

Exporting the Data

This task is the second of two tasks that generate the approved expense data for export and is responsible for creating a Windows Azure blob that contains a CSV file of approved expense submissions data.

The task that generates the blob containing the expense report data is slightly different from the two other tasks in the aExpense application. The other tasks poll a queue to see if there is any work for them to do. This task is triggered by a schedule, which sets the task to run at fixed times. The team at Adatum had to modify their worker role "plumbing code" classes to support scheduled tasks.

The worker role "plumbing code" classes now support scheduled tasks in addition to tasks that are triggered by a message on a queue.

Markus says:

We could extend the application to enable an on-premises application to generate an ad-hoc expense data report by allowing an on-premises application to place a message onto a Windows Azure queue. We could then have a task that generated the report data when it received a message on the queue.

You can use the abstract class **JobProcessor**, which implements the **IJobProcessor** interface, to define new scheduled tasks. The following code example shows the **JobProcessor** class.

```
public abstract class JobProcessor : IJobProcessor
{
    private bool keepRunning;

    protected JobProcessor(int sleepInterval)
    {
        if (sleepInterval <= 0)
        {
            throw new
                ArgumentOutOfRangeException("sleepInterval");
        }

        this.SleepInterval = sleepInterval;
    }

    protected int SleepInterval { get; set; }

    public void Run()
    {
        this.keepRunning = true;
        while (this.keepRunning)
        {
Thread.Sleep(this.SleepInterval);
            this.RunCore();
        }
    }

    public void Stop()
    {
        this.keepRunning = false;
    }

    protected abstract void RunCore();
}
```

This implementation does not make it easy to specify the exact time that scheduled tasks will run. The time between tasks will be the value of the sleep interval, plus the time taken to run the task. If you need the task to run at fixed time, you should measure how long the task takes to run and subtract that value from the sleep interval.

> The **BaseJobProcessor** class that defines tasks that read messages from queues extends the **JobProcessor** class.

In the aExpense application, the **ExpenseExportBuilderJob** class extends the **JobProcessor** class to define a scheduled task. The **ExpenseExportBuilderJob** class, shown in the following code example, defines the task that generates the expense report data and stores it as a blob. In this class, the **expenseExports** variable refers to the table of approved expense submissions, and the **exportStorage** variable refers to the report data for downloading in blob storage. The call to the base class constructor specifies the interval at which the job runs. In the **RunCore** method, the code first retrieves all the approved expense submissions from the export table based on the job date. Next, the code appends a CSV record to the export data in the blob storage for each approved expense submission. Finally, the code deletes all the records it copied to the blob storage from the table.

> The following code sets the scheduled interval to a low number for testing and demonstration purposes. You should change this interval for a "real" schedule.

```
public class ExpenseExportBuilderJob : JobProcessor
{
    private readonly ExpenseExportRepository expenseExports;
    private readonly ExpenseExportStorage exportStorage;

    public ExpenseExportBuilderJob() : base(100000)
    {
        this.expenseExports = new ExpenseExportRepository();
        this.exportStorage = new ExpenseExportStorage();
    }

    protected override void RunCore()
    {
        DateTime jobDate = DateTime.UtcNow;
        string name = jobDate.ToExpenseExportKey();

        IEnumerable<ExpenseExport> exports =
            this.expenseExports.Retreive(jobDate);
        if (exports == null || exports.Count() == 0)
        {
            return;
        }
        string text = this.exportStorage.GetExport(name);
        var exportText = new StringBuilder(text);
        foreach (ExpenseExport expenseExport in exports)
        {
            exportText.AppendLine(expenseExport.ToCsvLine());
        }

        this.exportStorage.AddExport(name,
            exportText.ToString(), "text/plain");

        // Delete the exports.
        foreach (ExpenseExport exportToDelete in exports)
        {
            try
            {
                this.expenseExports.Delete(exportToDelete);
            }
            catch (InvalidOperationException ex)
            {
                Log.Write(EventKind.Error,
                    ex.TraceInformation());
            }
        }
    }
}
```

If the process fails before it deletes all the approved expense submissions from the export table, any undeleted approved expense submissions will be exported a second time when the task next runs. However, the exported CSV data includes the expense ID and the approval date of the expense submission, so the on-premises payment processing system will be able to identify duplicate items.

The following code shows the queries that the **RunCore** method invokes to retrieve approved expense submissions and deletes them after it copies them to the export blob. Because they use the job date to identify the partitions to search, these queries are fast and efficient.

```
public IEnumerable<ExpenseExport> Retreive(DateTime jobDate)
{
    var context = new ExpenseDataContext(this.account);
    string compareDate = jobDate.ToExpenseExportKey();
    var query = (from export in context.ExpenseExport
           where export.PartitionKey.CompareTo(compareDate) <= 0
           select export).AsTableServiceQuery();

    var val = query.Execute();
    return val.Select(e => e.ToModel()).ToList();
}

public void Delete(ExpenseExport expenseExport)
```

```
{
    var context = new ExpenseDataContext(this.account);
    var query = (from export in context.ExpenseExport
      where export.PartitionKey.CompareTo(
        expenseExport.ApproveDate.ToExpenseExportKey()) == 0 &&
        export.RowKey.CompareTo(
        expenseExport.ExpenseId.ToString()) == 0
      select export).AsTableServiceQuery();
    ExpenseExportRow row = query.Execute().SingleOrDefault();
    if (row == null)
    {
        return;
    }

    context.DeleteObject(row);
    context.SaveChanges();
}
```

Performance Testing, Tuning, To-do Items

As part of the work for this phase, the team at Adatum evaluated the results from performance testing the application, and as a result, it made a number of changes to the aExpense application. They also documented some of the key "missing" pieces in the application that Adatum should address in the next phase of the project.

Adatum made changes to the aExpense application following their performance testing.

Storing Session State

In preparation for performance testing, and to ensure the scalability of the application, the team at Adatum made some changes to the way that the application handles session state. The AddExpense.aspx page uses session state to maintain a list of expense items before the user saves the completed business expense submission. During development and testing, the application used the default, in-memory, session state provider. This default in-memory session state provider is not suitable for scenarios with multiple web role instances, because the session state is not synchronized between instances, and the Windows Azure load balancer could route a request to any available instance. Adatum evaluated several approaches to resolve this issue in the aExpense application.

Bharath says:

If your application has to scale to more than one web role, any session state must be shared across the web role instances.

The first option considered was to use ASP.NET view state instead of session state so that the application maintains its state data on the client. This solution would work when the application has multiple web role instances because the application does not store any state data on the server.

Because the aExpense application stores scanned images in the state before the application saves the whole expense submission, this means that the state can be quite large. Using view state would be a poor solution in this case because it would need to move the data in the view state over the network, using up bandwidth and adversely affecting the application's performance.

Markus says:

Using ASP.NET view state is an excellent solution as long as the amount of data involved is small. This is not the case with the aExpense application where the state data includes images.

The second option is to continue to use session state but use a provider that persists the state to server-side storage that is accessible from multiple web role instances. Adatum could either implement its own, custom session state provider that uses either SQL Database or Windows Azure storage, or use the Session State Provider for Windows Azure Caching. Adatum is planning to investigate the relative costs and performance of the Session State Provider for Windows Azure Caching and a custom-built provider: in both cases, use of the provider is enabled through configuration and requires no code changes to the aExpense application.

For additional information about the Session State Provider for Windows Azure Caching, see http://msdn.microsoft.com/en-us/library/gg185668.aspx.

Using a Multi-Schema Table

Adatum originally decided to use two tables for storing expense data: a header table and a details table with records linked through a foreign key. The advantage of this approach is simplicity because each table has its own, separate, schema. However, because transactions cannot span tables in Windows Azure storage, then as described in Chapter 6, "Phase 3: Uploading Images and Adding a Worker Role," there is a possibility that orphaned detail records could be left if there was a failure before the aExpense application saved the header record. In this phase, Adatum implemented a multi-schema table for expense data so that it could use a single transaction for saving both header and detail records.

For a discussion about how Adatum implemented a multi-schema table, see the section "Implementing a Multi-Schema Table in Windows Azure Table Storage" below.

Too Many Calls to the CreateIfNotExist Method

Originally, the constructor for the **ExpenseReceiptStorage** class was responsible for checking that the expense receipt container existed, and creating it if necessary. This constructor is invoked whenever the application instantiates an**ExpenseRepository** object or a **ReceiptThumbnailJob** object. The **CreateIfNotExist** method that checks whether a container exists requires a round-trip to the storage server and incurs a storage transaction cost. To avoid these unnecessary round-trips, Adatum moved this logic to the **ApplicationStorageInitializer** class: the **Application_Start** method in the Global.asax.cs file invokes the **Initialize** method in this class when the very first request is received from a client.

You must be careful not to access the request context or call methods of the **RoleManager** class from the **Application_Start** method. For further details, see http://msdn.microsoft.com/en-us/library/microsoft.servicehosting.serviceruntime.rolemanager.aspx.

Preventing Users from Uploading Large Images

To prevent users from uploading large images of receipt scans to aExpense, Adatum configured the application to allow a maximum upload size of 1,024 kilobytes (KB) to the AddExpense.aspx page. The following code example shows the setting in the Web.config file.

```
<location path="AddExpense.aspx">
<system.web>
<authorization>
<allow roles="Employee" />
<deny users="*"/>
</authorization>

<!–
Maximum request allowed to send a big image as receipt.
-->
<httpRuntime maxRequestLength="1024"/>
</system.web>
</location>
```

Validating User Input

The cloud-based version of aExpense does not perform comprehensive checks on user input for invalid or dangerous items. The AddExpense.aspx file includes some basic validation that checks the length of user input, but Adatum should add additional validation checks to the **OnAddNewExpenseItemClick** method in the AddExpense.aspx.cs file.

Paging and Sorting on the Default.aspx Page

During performance testing, the response times on Default.aspx degraded as the test script added more and more expense submissions for a user. This happened because the current version of the Default.aspx page does not include any paging mechanism, so it always displays all the expense submissions for a user. As a temporary measure, Adatum modified the LINQ query that retrieves expense submissions by user to include a **Take(10)** clause, so that the application only requests the first 10 expense submissions. In a future phase of the project, Adatum will add paging functionality to the Default.aspx page.

For a discussion about how Adatum might implement paging on the Default.aspx and Approve.aspx, see the section "Implementing Paging with Windows Azure Table Storage" below.

Adatum also realized that their original choice of a GUID as the row key for the expense table was not optimal. The default ordering of expense submissions should be in reverse chronological order, because most users will want to see their most recent expense submissions at the top of the list or on the first page when Adatum implements paging. The expense approval page will also use this ordering.

Markus says:

We realized that using a GUID as the row key for expense submissions did not return data in a useful order. We refactored the code to use the **ExpenseKey** class instead. This class generates an inverted timestamp value for us.

To implement this change, Adatum modified the **Expense** class in the **AExpense.Model** namespace by changing the type of the **Id** property from **Guid** to **ExpenseKey**. The following code example shows how the static **Now** property of the **ExpenseKey** class generates an inverted tick count to use in its **InvertedTicks** property.

```
public static ExpenseKey Now
{
    get
    {
        return new ExpenseKey(
            string.Format("{0:D19}",
            DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks));
    }
}
```

The query that retrieves expense submissions by user now returns them in reverse chronological order.

System.Net Configuration Changes

The following code example shows two configuration changes that Adatum made to the aExpense application to improve its performance.

```
<system.net>
<settings>
<servicePointManager expect100Continue="false" />
</settings>
<connectionManagement>
<add address = "*" maxconnection = "24" />
</connectionManagement>
</system.net>
```

The first change switches off the "Expect 100-continue" feature. If this feature is enabled, when the application sends a PUT or POST request, it can delay sending the payload by sending an "Expect 100-continue" header. When the server receives this message, it uses the available information in the header to check whether it could make the call, and if it can, it sends back a status code 100 to the client. The client then sends the remainder of the payload. This means that the client can check for many common errors without sending the payload. If you have tested the client well enough to ensure that it is not sending any bad requests, you can turn off the "Expect 100-continue" feature and reduce the number of round trips to the server. This is especially useful when the client sends many messages with small payloads, for example, when the client is using the table or queue service.

The second configuration change increases the maximum number of connections that the web server will maintain from its default value of **2**. If this value is set too low, the problem manifests itself through "Underlying connection was closed" messages.

The exact number to use for this setting depends on your application. The page at http://support.microsoft.com/kb/821268has useful information about how to set this for server side applications. You can also set it for a particular URI by specifying the URI in place of "*".

WCF Data Service Optimizations

Because of a known performance issue with WCF Data Services, Adatum defined a **ResolveType** delegate on the **DataServiceContext** class in the aExpense application. Without this delegate, query performance degrades as the number of entities that the query returns increases. The following code example shows the delegate definition.

Markus says:

We made a number of changes to our WCF Data Services code to improve performance.

```
private static Type ResolveEntityType(string name)
{
    var tableName = name.Split(new[] { '.' }).Last();
    switch (tableName)
    {
        case ExpenseTable:
            return typeof(ExpenseRow);
        case ExpenseItemTable:
            return typeof(ExpenseItemRow);
        case ExpenseExportTable:
            return typeof(ExpenseExportRow);
    }

    throw new ArgumentException(
        string.Format(
            CultureInfo.InvariantCulture,
            "Could not resolve the table name '{0}'
            to a known entity type.", name));
}
```

Instead of using the **ResolveType** delegate, you can avoid the performance problem by ensuring that your entity class names exactly match the table names.

Adatum added a further optimization to the WCF Data Services client code by setting the **MergeOption** to **NoTracking** for the queries in the **ExpenseRepository** class. If you are not making any changes to the entities that WCF Data Services retrieve, there is no need for the **DataContext** object to initialize change tracking for entities.

Implementing a Multi-Schema Table in Windows Azure Table Storage

Originally, Adatum used two tables for storing expenses: one table for header records, and one for detail records. This meant that Adatum had to use two transactions to save an expense, leading to the risk of orphaned detail records. The following code sample shows the outline of the original **SaveExpense** method in the **ExpenseRepository** class — each call to the **SaveChanges** method is a separate transaction.

```
public void SaveExpense(Expense expense)
{
    var context = new ExpenseDataContext(this.account);
    ExpenseRow expenseRow = expense.ToTableEntity();

    foreach (var expenseItem in expense.Details)
    {
        // create row
        var expenseItemRow = expenseItem.ToTableEntity();
        expenseItemRow.PartitionKey = expenseRow.PartitionKey;
        expenseItemRow.RowKey =
          string.Format(CultureInfo.InvariantCulture, "{0}_{1}",
          expense.Id, expenseItemRow.Id);

        context.AddObject(ExpenseDataContext.ExpenseItemTable,
          expenseItemRow);

        …
    }

    context.SaveChanges(SaveChangesOptions.Batch);

    context.AddObject(ExpenseDataContext.ExpenseTable,
        expenseRow);
    context.SaveChanges();

    …
}
```

As you go through this section, you may want to download the Microsoft® Visual Studio® development system solution from http://wag.codeplex.com/. This solution (in the Azure-MultiEntitySchema folder) contains the implementation of aExpense after Adatum made the changes described in this section.

Storing different entity types in the same table enables you to use transactions when you work with multiple types of entity. The following code sample shows the new version of the **SaveExpenses** method with only a single transaction.

```
public void SaveExpense(Expense expense)
{
    var context = new ExpenseDataContext(this.account);
    IExpenseRow expenseRow = expense.ToTableEntity();
    expenseRow.PartitionKey = ExpenseRepository
```

```
        .EncodePartitionAndRowKey(expenseRow.UserName);
    expenseRow.RowKey = expense.Id.ToString();
    context.AddObject(ExpenseDataContext.ExpenseTable,
        expenseRow);

    foreach (var expenseItem in expense.Details)
    {
        // Create an expense item row.
        var expenseItemRow = expenseItem.ToTableEntity();
        expenseItemRow.PartitionKey = expenseRow.PartitionKey;
        expenseItemRow.RowKey = string.Format(
            CultureInfo.InvariantCulture, "{0}_{1}", expense.Id,
            expenseItemRow.ItemId);
        context.AddObject(ExpenseDataContext.ExpenseTable,
            expenseItemRow);

        …
    }

    // Save expense header and items details in the same batch
    // transaction.
    context.SaveChanges(SaveChangesOptions.Batch);

    …
}
```

The changes that Adatum made to aExpense to implement multiple schemas in a single table are best understood by looking first at how the application retrieves data from the table, and then at how data is saved to the table.

Defining the Schemas

In the aExpense application, two types of entity are stored in the Expense table: expense header entities (defined by the **IExpenseRow** interface) and expense detail entities (defined by the **IExpenseItemRow** interface). The following code sample shows these two interfaces and the **IRow** interface that defines the entity key.

Markus says:

Both entity types share the same key structure defined in the **IRow** interface.

```
public interface IExpenseRow : IRow
{
    string Id { get; set; }
    string UserName { get; set; }
    bool? Approved { get; set; }
    string ApproverName { get; set; }
    string CostCenter { get; set; }
    // DateTime has to be Nullable to run it in the storage
    // emulator.
    // The same applies for all types that are not nullable like
    // bool and Guid.
    DateTime? Date { get; set; }
    string ReimbursementMethod { get; set; }
    string Title { get; set; }
}

public interface IExpenseItemRow : IRow
{
    // Guid has to be Nullable to run it in the storage emulator.
    // The same applies for all types that are not nullable like
    // bool and DateTime.
    Guid? ItemId { get; set; }
    string Description { get; set; }
    double? Amount { get; set; }
    string ReceiptUrl { get; set; }
    string ReceiptThumbnailUrl { get; set; }
}

public interface IRow
{
    string PartitionKey { get; set; }
    string RowKey { get; set; }
    DateTime Timestamp { get; set; }
string Kind { get; set; }
}
```

Adatum uses the **ExpenseAndExpenseItemRow** and **Row** classes to implement the **IRow**, **IExpenseRow**, and **IExpenseItemRow** interfaces, and to extend the **TableServiceEntity** class from the **StorageClient** namespace. The following code sample shows the **Row** and **ExpenseAndExpenseItemRow** classes. The **Row** class defines a **Kind** property that is used to distinguish between the two types of entity stored in the table (see the **TableRows** enumeration in the DataAccessLayer folder).

```
public abstract class Row : TableServiceEntity, IRow
{
    protected Row()
    {
    }

    protected Row(string kind) : this(null, null, kind)
    {
    }

    protected Row(
        string partitionKey, string rowKey, string kind)
        : base(partitionKey, rowKey)
    {
        this.Kind = kind;
    }

    public string Kind { get; set; }
}

public class ExpenseAndExpenseItemRow
    : Row, IExpenseRow, IExpenseItemRow
{
    public ExpenseAndExpenseItemRow()
    {
    }

    public ExpenseAndExpenseItemRow(TableRows rowKind)
    {
        this.Kind = rowKind.ToString();
    }

    // Properties from ExpenseRow
    public string Id { get; set; }
    public string UserName { get; set; }
    public bool? Approved { get; set; }
    public string ApproverName { get; set; }
    public string CostCenter { get; set; }
    public DateTime? Date { get; set; }
    public string ReimbursementMethod { get; set; }
    public string Title { get; set; }

    // Properties from ExpenseItemRow
    public Guid? ItemId { get; set; }
    public string Description { get; set; }
    public double? Amount { get; set; }
    public string ReceiptUrl { get; set; }
    public string ReceiptThumbnailUrl { get; set; }
}
```

The following code example shows how the **ExpenseDataContext** class maps the **ExpenseAndExpenseItemRow** class to the **MultiEntitySchemaExpenses** table.

```
public class ExpenseDataContext : TableServiceContext
{
    public const string ExpenseTable =
        "MultiEntitySchemaExpenses";


    …

    public IQueryable<ExpenseAndExpenseItemRow>
        ExpensesAndExpenseItems
    {
        get
        {
            return this.CreateQuery<ExpenseAndExpenseItemRow>(
                ExpenseTable);
        }
    }
    ...
}
```

Retrieving Records from a Multi-Schema Table

The query methods in the **ExpenseRepository** class use the **ExpenseAndExpenseItemRow** entity class when they retrieve either header or detail entities from the expense table. The following code example from the **GetExpensesByUser** method in the **ExpenseRespository** class shows how to retrieve a header row (defined by the **IExpenseRow** interface).

```
var context = new ExpenseDataContext(this.account)
    { MergeOption = MergeOption.NoTracking };

var query = (from expense in context.ExpensesAndExpenseItems
                where
                expense.UserName.CompareTo(userName) == 0 &&
                expense.PartitionKey.CompareTo(
                EncodePartitionAndRowKey(userName)) == 0
                select expense).Take(10).AsTableServiceQuery();

try
{
    return query.Execute().Select(
        e => (e as IExpenseRow).ToModel()).ToList();
}
```

Markus says:

This example does not need to use the **Kind** property because only header entities have a **UserName** property.

The following code sample from the **GetExpensesById** method in the **ExpenseRepository** class uses the **Kind** property to select only detail entities.

```
var expenseAndItemRows = query.Execute().ToList();

...

expenseAndItemRows.
    Where(e => e.Kind == TableRows.ExpenseItem.ToString()).
    Select(e => (e as IExpenseItemRow).ToModel()).
    ToList().ForEach(e => expense.Details.Add(e));
```

Adding New Records to the Multi-Schema Table

The aExpense application uses the **ExpenseAndExpenseItemRow** entity class when it adds new entities to the expense table. The following code sample from the **ExpenseRepository** class shows how the application saves an expense to table storage.

```
public void SaveExpense(Expense expense)
{
    var context = new ExpenseDataContext(this.account);
    IExpenseRow expenseRow = expense.ToTableEntity();
    expenseRow.PartitionKey = ExpenseRepository
        .EncodePartitionAndRowKey(expenseRow.UserName);
    expenseRow.RowKey = expense.Id.ToString();
    context.AddObject(ExpenseDataContext.ExpenseTable,
        expenseRow);

    foreach (var expenseItem in expense.Details)
    {
        // Create an expense item row.
        var expenseItemRow = expenseItem.ToTableEntity();
        expenseItemRow.PartitionKey = expenseRow.PartitionKey;
        expenseItemRow.RowKey = string.Format(
            CultureInfo.InvariantCulture, "{0}_{1}", expense.Id,
            expenseItemRow.ItemId);
        context.AddObject(ExpenseDataContext.ExpenseTable,
            expenseItemRow);
        ...
}

    // Save expense header and items details in the same batch
    // transaction.
    context.SaveChanges(SaveChangesOptions.Batch);
    ...
}
```

Markus says:

Notice how the two overloaded versions of the **ToTableEntity** extension method return either an **IExpenseRow** or an **IExpenseItemRow** instance.

Implementing Paging with Windows Azure Table Storage

Adatum have not implemented any paging functionality in the current phase of the project, but this section gives an outline of the approach they intend to take. The **ResultSegment** class in the Windows Azure StorageClient library provides an opaque **ContinuationToken** property that you can use to access the next set of results from a query if that query did not return the full set of results, for example, if the query used the **Take** operator to return a small number of results to display on a page. This **ContinuationToken** property will form the basis of any paging implementation.

Markus says:

By implementing paging, we can improve the performance of the application by returning just the data the user needs to see.

The **ResultSegment** class only returns a **ContinuationToken** object to access the next page of results, and not the previous page, so if your application requires the ability to page backward, you must store **ContinuationToken**objects that point to previous pages. A stack is a suitable data structure to use. Figure 7 shows the state of a stack after a user has browsed to the first page and then paged forward as far as the third page.

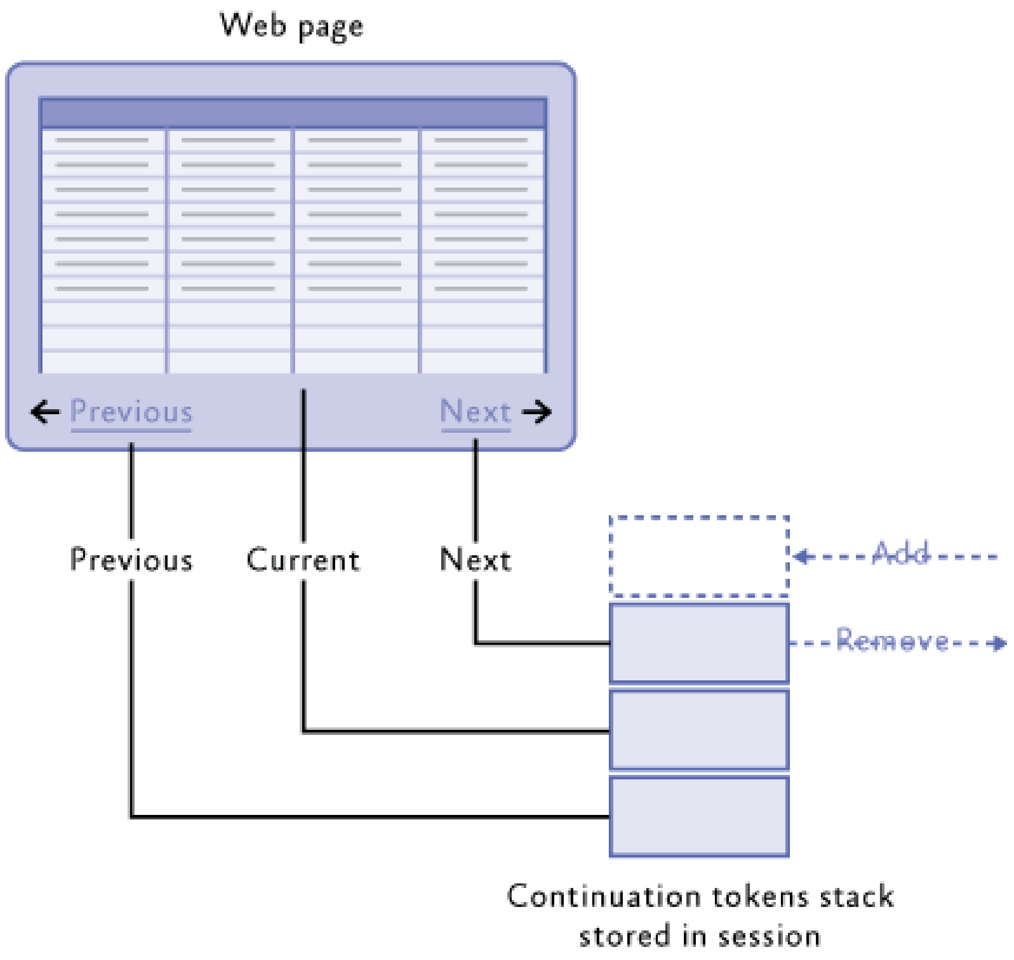


Figure 7
Displaying page 3 of the data from a table

If a user clicks the **Next** hyperlink to browse to page 4, the page peeks at the stack to get the continuation token for page 4. After the page executes the query with the continuation token from the stack, it pushes a new continuation token for page 5 onto the stack.

If a user clicks the **Previous** hyperlink to browse to page 2, the page will pop two entries from the stack, and then peek at the stack to get the continuation token for page 2. After the page executes the query with the continuation token from the stack, it will push a new continuation for page 3 onto the stack.

The following code examples show how Adatum could implement this behavior on an asynchronous ASP.NET page.

Using an asynchronous page frees up the pages thread from the thread pool while a potentially long-running I/O operation takes place. This improves throughput on the web server and increases the scalability of the application.

The following two code examples show how to create an asynchronous ASP.NET page. First, add an **Async="true"** attribute to the page directive in the .aspx file.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="Default.aspx.cs"
Inherits="ContinuationSpike._Default" Async="true"%>
```

Second, register begin and end methods for the asynchronous operation in the load event for the page.

```
protected void Page_Load(object sender, EventArgs e)
{
    AddOnPreRenderCompleteAsync(
        new BeginEventHandler(BeginAsyncOperation),
        new EndEventHandler(EndAsyncOperation)
    );
}
```

The following code example shows the definition of the **ContinuationStack** class that the application uses to store continuation tokens in the session state.

Markus says:

We need to store the stack containing the continuation tokens as a part of the session state.

```
public class ContinuationStack
{
    private readonly Stack stack;

    public ContinuationStack()
    {
        this.stack = new Stack();
    }

    public bool CanMoveBack()
    {
        if (this.stack.Count >= 2)
            return true;
        return false;
    }

    public bool CanMoveForward()
    {
        return this.GetForwardToken() != null;
    }

    public ResultContinuation GetBackToken()
    {
        if (this.stack.Count == 0)
            return null;
        // Need to pop twice and then return what is left.
        this.stack.Pop();
        this.stack.Pop();
        if (this.stack.Count == 0)
            return null;
        return this.stack.Peek() as ResultContinuation;
    }

    public ResultContinuation GetForwardToken()
    {
        if (this.stack.Count == 0)
            return null;

        return this.stack.Peek() as ResultContinuation;
    }

    public void AddToken(ResultContinuation result)
    {
        this.stack.Push(result);
    }
}
```

The following code example shows the **BeginAsyncOperation** method that starts the query execution for the next page of data. The **ct** value in the query string specifies the direction to move.

```
private IAsyncResult BeginAsyncOperation(object sender, EventArgs
e, AsyncCallback cb, object extradata)
{
    var query =
      new MessageContext(CloudConfiguration.GetStorageAccount())
      .Messages.Take(3).AsTableServiceQuery();
    if (Request["ct"] == "forward")
    {
        var segment = this.ContinuationStack.GetForwardToken();
        return query.BeginExecuteSegmented(segment, cb, query);
    }
    if (Request["ct"] == "back")
    {
        var segment = this.ContinuationStack.GetBackToken();
        return query.BeginExecuteSegmented(segment, cb, query);
    }
    return query.BeginExecuteSegmented(cb, query);
}
```

The **EndAsyncOperation** method puts the query results into the messages list and pushes the new continuation token onto the stack.

```
private List<MessageEntity> messages;

private void EndAsyncOperation(IAsyncResult result)
{
    var cloudTableQuery =
        result.AsyncState as CloudTableQuery<MessageEntity>;
    ResultSegment<MessageEntity> resultSegment =
        cloudTableQuery.EndExecuteSegmented(result);
    this.ContinuationStack.AddToken(
        resultSegment.ContinuationToken);
    this.messages = resultSegment.Results.ToList();
}
```

More Information

You can read detailed guidance on using Windows Azure table storage in this document: http://go.microsoft.com/fwlink/?LinkId=153401.

You can read detailed guidance on using Windows Azure blob storage in this document: http://go.microsoft.com/fwlink/?LinkId=153400.

Glossary

affinity group. A named grouping that is in a single data center. It can include all the components associated with an application, such as storage, Windows Azure SQL Database instances, and roles.

claim. A statement about a subject; for example, a name, identity, key, group, permission, or capability made by one subject about itself or another subject. Claims are given one or more values and then packaged in security tokens that are distributed by the issuer.

cloud. A set of interconnected servers located in one or more data centers.

code near. When an application and its associated database(s) are both in the cloud.

code far. When an application is on-premises and its associated database(s) are in the cloud.

Compute Emulator.The Windows Azure Compute Emulator enables you to run, test, debug, and fine-tune your application before you deploy it as a hosted service to Windows Azure. See also: Storage Emulator.

Content Delivery Network (CDN). A system composed of multiple servers that contain copies of data. These servers are located in different geographical areas so that users can access the copy that is closes to them.

Enterprise Library. A collection of reusable software components (application blocks) designed to assist software developers with common enterprise development cross-cutting concerns (such as logging, validation, data access, exception handling, and many others).

horizontal scalability. The ability to add more servers that are copies of existing servers.

hosted service. Spaces where applications are deployed.

idempotent operation. An operation that can be performed multiple times without changing the result. An example is setting a variable.

lease. An exclusive write lock on a blob that lasts until the lease expires.

optimistic concurrency. A concurrency control method that assumes that multiple changes to data can complete without affecting each other; therefore, there is no need to lock the data resources. Optimistic concurrency assumes that concurrency violations occur infrequently and simply disallows any updates or deletions that cause a concurrency violation.

poison message. A message that contains malformed data that causes the queue processor to throw an exception. The result is that the message isn't processed, stays in the queue, and the next attempt to process it once again fails.

Representational State Transfer (REST). An architectural style for retrieving information from Web sites. A resource is the source of specific information. Each resource is identified by a global identifier, such as a Uniform Resource Identifier (URI) in HTTP. The representation is the actual document that conveys the information.

service configuration file. Sets values for the service that can be configured while the service is running in the fabric. The values you can specify in the service configuration file include the number of instances that you want to deploy for each role, the values for the configuration parameters that you established in the service definition file, and the thumbprints for any SSL certificates associated with the service.

service definition file. Defines the roles that comprise a service, optional local storage resources, configuration settings, and certificates for SSL endpoints.

service package.Packages the role binaries and service definition file for publication to the Windows Azure fabric.

snapshot. A read-only copy of a blob.

Storage Emulator.The Windows Azure storage emulator provides local instances of the blob, queue, and table services that are available in the Windows Azure. If you are building an application that uses storage services, you can test locally by using the storage emulator.

vertical scalability. The ability to increase a computer's resources, such as memory or CPUs.

Web role.An interactive application that runs in the cloud. A web role can be implemented with any technology that works with Internet Information Services (IIS) 7.

Windows Azure. The Microsoft platform for cloud-based computing. It is provided as a service over the Internet. It includes a computing environment, Windows Azure storage, and management services.

Windows Azure SQL Database (SQL Database). A relational database management system (RDBMS) in the cloud. SQL Database is independent of the storage that is a part of Windows Azure. It is based on SQL Server® and can store structured, semi-structured, and unstructured data.

Windows Azure storage. Consists of blobs, tables, and queues. It is accessible with HTTP/HTTPS requests. It is distinct from SQL Database.

Worker role. Performs batch processes and background tasks. Worker roles can make outbound calls and open endpoints for incoming calls. Worker roles typically use queues to communicate with Web roles.