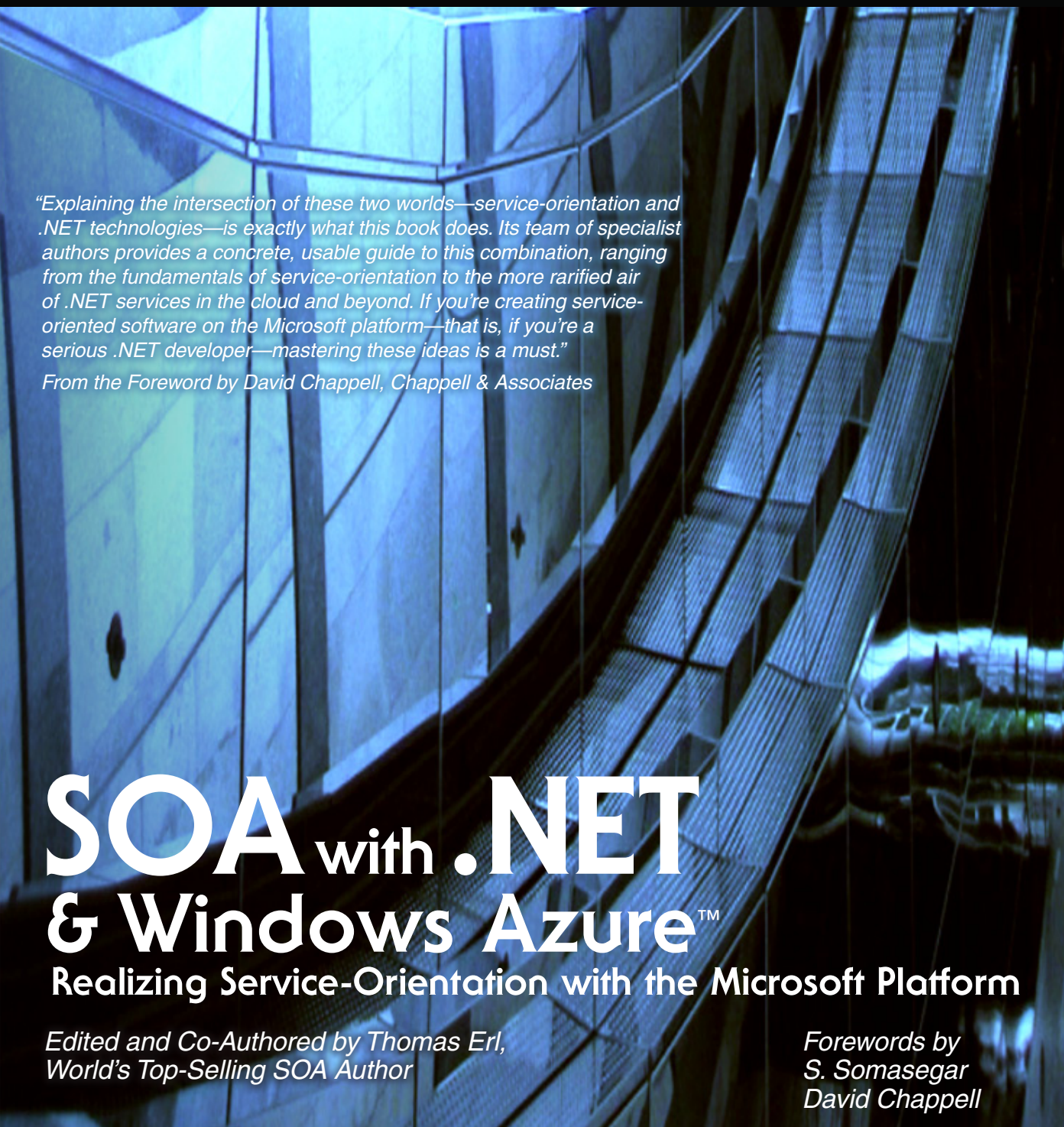
THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



"Explaining the intersection of these two worlds—service-orientation and .NET technologies—is exactly what this book does. Its team of specialist authors provides a concrete, usable guide to this combination, ranging from the fundamentals of service-orientation to the more rarified air of .NET services in the cloud and beyond. If you're creating service-oriented software on the Microsoft platform—that is, if you're a serious .NET developer—mastering these ideas is a must."

From the Foreword by David Chappell, Chappell & Associates

SOA with .NET & Windows Azure™

Realizing Service-Orientation with the Microsoft Platform

Edited and Co-Authored by Thomas Erl, World's Top-Selling SOA Author

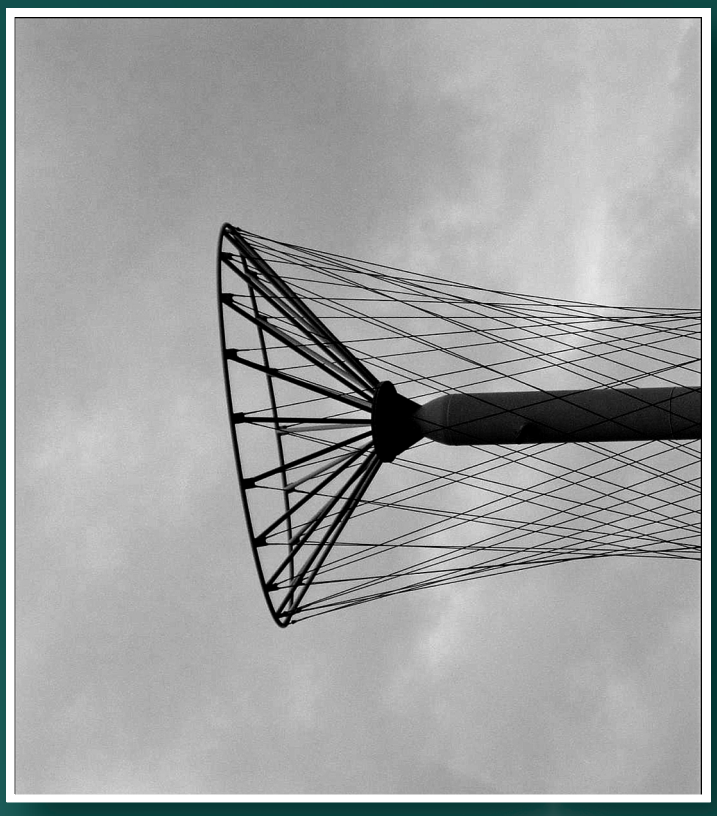
Forewords by S. Somasegar David Chappell

David Chou, John deVadoss, Nitin Gandhi, Hanu Kommapalati, Brian Loesgen, Christoph Shittko, Herbjorn Wilhelmsen, Mickie Williams

With contributions from Scott Golightly, Daryl Hogan, Jeff King, Scott Seely
With additional contributions by members of the Microsoft Windows Azure and AppFabric teams

PRENTICE HALL

SOA with .NET and Windows Azure™



Chapter 8

Cloud Services with Windows Azure

8.1 Cloud Computing 101

8.2 Windows Azure Platform Overview

8.3 Windows Azure Roles

8.4 Hello World in Windows Azure

8.5 A Web Service in Windows Azure

8.6 A REST Service in Windows Azure

8.7 Windows Azure Storage

Microsoft's Software-plus-Services strategy represents a view of the world where the growing feature-set of devices and the increasing ubiquity of the Web are combined to deliver more compelling solutions. Software-plus-Services represents an evolutionary step that is based on existing best practices in IT and extends the application potential of core service-orientation design principles.

Microsoft's efforts to embrace the Software-plus-Services vision are framed by three core goals:

- user experiences should span beyond a single device
- solution architectures should be able to intelligently leverage and integrate on-premise IT assets with cloud assets
- tightly coupled systems should give way to federations of cooperating systems and loosely coupled compositions

The Windows Azure platform represents one of the major components of the Software-plus-Services strategy, as Microsoft's cloud computing operating environment, designed from the outset to holistically manage pools of computation, storage and networking; all encapsulated by one or more services.

8.1 Cloud Computing 101

Just like service-oriented computing, cloud computing is a term that represents many diverse perspectives and technologies. In this book, our focus is on cloud computing in relation to SOA and Windows Azure.

Cloud computing enables the delivery of scalable and available capabilities by leveraging dynamic and on-demand infrastructure. By leveraging these modern service technology advances and various pervasive Internet technologies, the "cloud" represents an abstraction of services and resources, such that the underlying complexities of the technical implementations are encapsulated and transparent from users and consumer programs interacting with the cloud.

At the most fundamental level, cloud computing impacts two aspects of how people interact with technologies today:

- how services are consumed
- how services are delivered

Although cloud computing was originally, and still often is, associated with Web-based applications that can be accessed by end-users via various devices, it is also very much about applications and services themselves being consumers of cloud-based services. This fundamental change is a result of the transformation brought about by the adoption of SOA and Web-based industry standards, allowing for service-oriented and Web-based resources to become universally accessible on the Internet as on-demand services.

One example has been an approach whereby programmatic access to popular functions on Web properties is provided by simplifying efforts at integrating public-facing services and resource-based interactions, often via RESTful interfaces. This was also termed "Web-oriented architecture" or "WOA," and was considered a subset of SOA. Architectural views such as this assisted in establishing the Web-as-a-platform concept, and helped shed light on the increasing inter-connected potential of the Web as a massive collection (or cloud) of ready-to-use and always-available capabilities.

This view can fundamentally change the way services are designed and constructed, as we reuse not only someone else's code and data, but also their infrastructure resources, and leverage them as part of our own service implementations. We do not need to understand the inner workings and technical details of these services; Service Abstraction (696), as a principle, is applied to its fullest extent by hiding implementation details behind clouds.

With regards to service delivery, we are focused on the actual design, development, and implementation of cloud-based services. Let's begin by establishing high-level characteristics that a cloud computing environment can include:

- generally accessible
- always available and highly reliable
- elastic and scalable
- abstract and modular resources

SOA PRINCIPLES & PATTERNS

There are several SOA design patterns that are closely related to common cloud computing implementations, such as Decoupled Contract [735], Redundant Implementation [766], State Repository [785], and Stateful Services [786]. In this and subsequent chapters, these and other patterns will be explored as they apply specifically to the Windows Azure cloud platform.

- service-oriented
- self-service management and simplified provisioning

Fundamental topics regarding service delivery pertain to the cloud deployment model used to provide the hosting environment and the service delivery model that represents the functional nature of a given cloud-based service. The next two sections explore these two types of models.

Cloud Deployment Models

There are three primary cloud deployment models. Each can exhibit the previously listed characteristics; their differences lie primarily in the scope and access of published cloud services, as they are made available to service consumers.

Let's briefly discuss these deployment models individually.

Public Cloud

Also known as external cloud or multi-tenant cloud, this model essentially represents a cloud environment that is openly accessible. It generally provides an IT infrastructure in a third-party physical data center that can be utilized to deliver services without having to be concerned with the underlying technical complexities.

Essential characteristics of a public cloud typically include:

- homogeneous infrastructure
- common policies
- shared resources and multi-tenant
- leased or rented infrastructure; operational expenditure cost model
- economies of scale and elastic scalability

Note that public clouds can host individual services or collections of services, allow for the deployment of service compositions, and even entire service inventories.

Private Cloud

Also referred to as internal cloud or on-premise cloud, a private cloud intentionally limits its access to its resources to service consumers that belong to the same organization that owns the cloud. In other words, the infrastructure that is managed and operated for one

organization only, primarily to maintain a consistent level of control over security, privacy, and governance.

Essential characteristics of a private cloud typically include:

- heterogeneous infrastructure
- customized and tailored policies
- dedicated resources
- in-house infrastructure (capital expenditure cost model)
- end-to-end control

Community Cloud

This deployment model typically refers to special-purpose cloud computing environments shared and managed by a number of related organizations participating in a common domain or vertical market.

Other Deployment Models

There are variations of the previously discussed deployment models that are also worth noting. The *hybrid cloud*, for example, refers to a model comprised of both private and public cloud environments. The *dedicated cloud* (also known as the hosted cloud or virtual private cloud) represents cloud computing environments hosted and managed off-premise or in public cloud environments, but dedicated resources are provisioned solely for an organization's private use.

The Intercloud (Cloud of Clouds)

The intercloud is not as much a deployment model as it is a concept based on the aggregation of deployed clouds (Figure 8.1). Just like the Internet, which is a network of networks; intercloud refers to an inter-connected global cloud of clouds. Also like the World Wide Web, intercloud represents a massive collection of services that organizations can explore and consume.

From a services consumption perspective, we can look at the intercloud as an on-demand SOA environment where useful services managed by other organizations can be leveraged and composed. In other words, services that are outside of an organization's own boundaries and operated and managed by others can become a part of the aggregate portfolio of services of those same organizations.

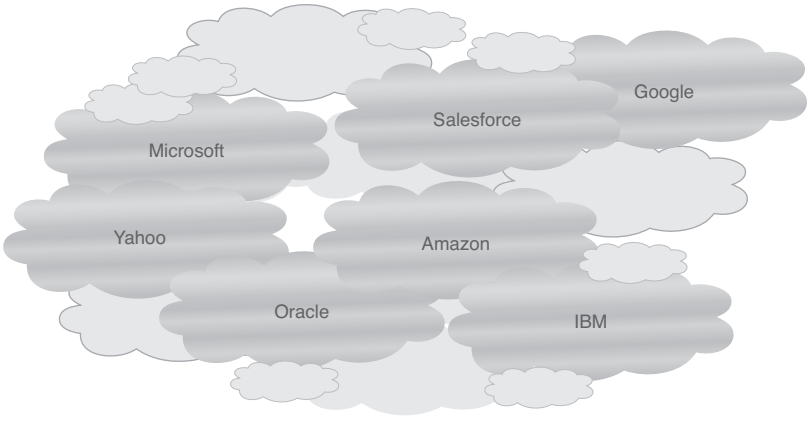


Figure 8.1
Examples of how vendors establish a commercial intercloud.

Deployment Models and Windows Azure

Windows Azure exists in a public cloud. Windows Azure itself is not made available as a packaged software product for organizations to deploy into their own IT enterprises. However, Windows Azure-related features and extensions exist in Microsoft's on-premise software products, and are collectively part of Microsoft's private cloud strategy. It is important to understand that even though the software infrastructure that runs Microsoft's public cloud and private clouds are different, layers that matter to end-user organizations, such as management, security, integration, data, and application are increasingly consistent across private and public cloud environments.

Service Delivery Models

Many different types of services can be delivered in the various cloud deployment environments. Essentially, any IT resource or function can eventually be made available as a service. Although cloud-based ecosystems allow for a wide range of service delivery models, three have become most prominent:

Infrastructure-as-a-Service (IaaS)

This service delivery model represents a modern form of utility computing and outsourced managed hosting. IaaS environments manage and provision fundamental computing resources (networking, storage, virtualized servers, etc.). This allows consumers to deploy and manage assets on leased or rented server instances, while the service providers own and govern the underlying infrastructure.

Platform-as-a-Service (PaaS)

The PaaS model refers to an environment that provisions application platform resources to enable direct deployment of application-level assets (code, data, configurations, policies, etc.). This type of service generally operates at a higher abstraction level so that users manage and control the assets they deploy into these environments. With this arrangement, service providers maintain and govern the application environments, server instances, as well as the underlying infrastructure.

Software-as-a-Service (SaaS)

Hosted software applications or multi-tenant application services that end-users consume directly correspond to the SaaS delivery model. Consumers typically only have control over how they use the cloud-based service, while service providers maintain and govern the software, data, and underlying infrastructure.

Other Delivery Models

Cloud computing is not limited to the aforementioned delivery models. Security, governance, business process management, integration, complex event processing, information and data repository processing, collaborative processes—all can be exposed as services and consumed and utilized to create other services.

NOTE

Cloud deployment models and service delivery models are covered in more detail in the upcoming book *SOA & Cloud Computing* as part of the *Prentice Hall Service-Oriented Computing Series from Thomas Erl*. This book will also introduce several new design patterns related to cloud-based service, composition, and platform design.

IaaS vs. PaaS

In the context of SOA and developing cloud-based services with Windows Azure, we will focus primarily on IaaS and PaaS delivery models in this chapter. Figure 8.2 illustrates a helpful comparison that contrasts some primary differences. Basically, IaaS represents a separate environment to host the same assets that were traditionally hosted on-premise, whereas PaaS represents environments that can be leveraged to build and host next-generation service-oriented solutions.

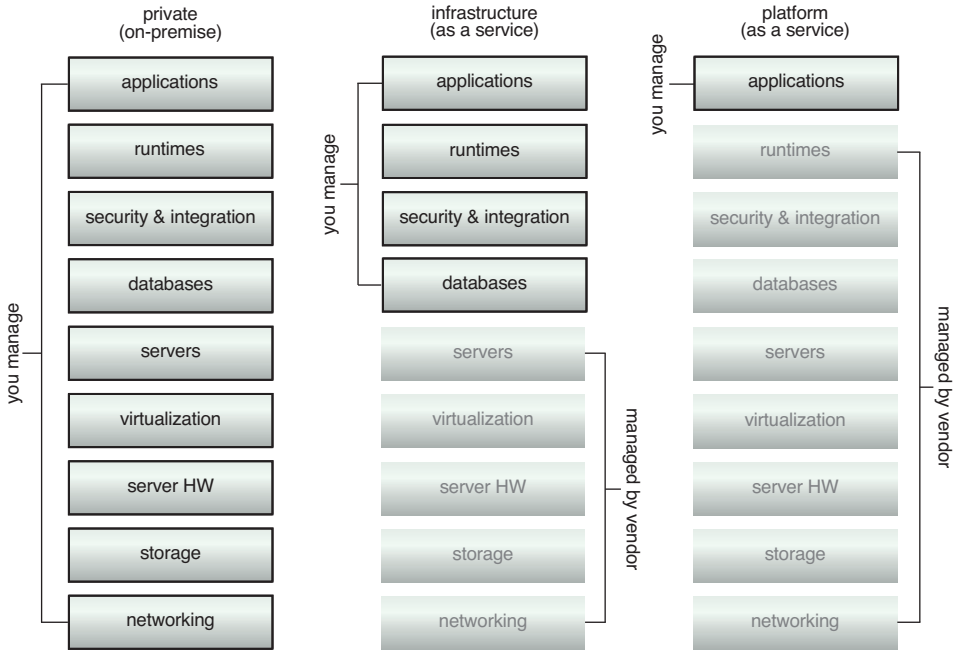


Figure 8.2
Common differentiations between delivery models.

We interact with PaaS at a higher abstraction level than with IaaS. This means we manage less of the infrastructure and assume simplified administration responsibilities. But at the same time, we have less control over this type of environment.

IaaS provides a similar infrastructure to traditional on-premise environments, but we may need to assume the responsibility to re-architect an application in order to effectively leverage platform service clouds. In the end, PaaS will generally achieve a higher level of scalability and reliability for hosted services.

IN PLAIN ENGLISH

An on-premise infrastructure is like having your own car. You have complete control over when and where you want to drive it, but you are also responsible for its operation and maintenance. IaaS is like using a car rental service. You still have control over when and where you want to go, but you don't need to be concerned with the vehicle's maintenance. PaaS is more comparable to public transportation. It is easier to use as you don't need to know how to operate it and it costs less. However, you don't have control over its operation, schedule, or routes.

SUMMARY OF KEY POINTS

- Cloud computing enables the delivery of scalable and available capabilities by leveraging dynamic and on-demand infrastructure.
- There are three common types of cloud deployment models: public cloud, private cloud, and community cloud.
- There are three common types of service delivery models: IaaS, PaaS, and SaaS.

8.2 Windows Azure Platform Overview

The Windows Azure platform is an Internet-scale cloud computing services platform hosted in Microsoft data centers. Windows tools provide functionality to build solutions that include a cloud services operating system and a set of developer services. The key parts of the Windows Azure platform are:

- Windows Azure (application container)
- Microsoft SQL Azure
- Windows Azure platform AppFabric

The Windows Azure platform is part of the Microsoft cloud, which consists of multiple categories of services:

- *cloud-based applications* – These are services that are always available and highly scalable. They run in the Microsoft cloud that consumers can directly utilize. Examples include Bing, Windows Live Hotmail, Office Live, etc.
- *software services* – These services are hosted instances of Microsoft's enterprise server products that consumers can use directly. Examples include Exchange Online, SharePoint Online, Office Communications Online, etc.

SOA PRINCIPLES & PATTERNS

The infrastructure and service architectures that underlie many of these native services (as well as cloud-based services in general) are based on direct combined application of Stateful Services [786] and Redundant Implementation [766]. This is made possible by leveraging several of the built-in extensions and mechanisms provided by the Windows Azure platform (as explained in this chapter and Chapter 16).

- *platform services* – This is where the Windows Azure platform itself is positioned. It serves as an application platform public cloud that developers can use to deploy next-generation, Internet-scale, and always available solutions.
- *infrastructure services* – There is a limited set of elements of the Windows Azure platform that can support cloud-based infrastructure resources.

Figure 8.3 illustrates the service categories related to the Windows Azure platform. Given that Windows Azure is itself a platform, let's explore it as an implementation of the PaaS delivery model.

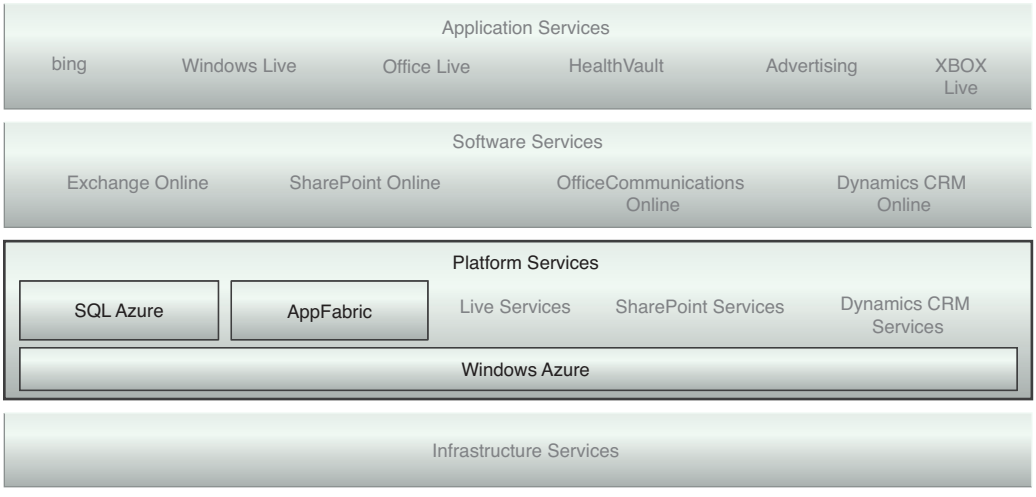


Figure 8.3
A high-level representation of categories of services available in the Windows Azure cloud.

The Windows Azure platform was built from the ground up using Microsoft technologies, such as the Windows Server Hyper-V-based system virtualization layer. However, the Windows Azure platform is not intended to be just another off-premise Windows Server hosting environment. It has a cloud fabric layer, called the *Windows Azure Fabric Controller*, built on top of its underlying infrastructure.

The Windows Azure Fabric Controller pools an array of virtualized Windows Server instances into a logical entity and automatically manages the following:

- resources
- load balancing
- fault-tolerance

- geo-replication
- application lifecycle

These are managed without requiring the hosted applications to explicitly deal with the details. The fabric layer provides a parallel management system that abstracts the complexities in the infrastructure and presents a cloud environment that is inherently elastic. As a form of PaaS, it also supports the access points for user and application interactions with the Windows Azure platform.

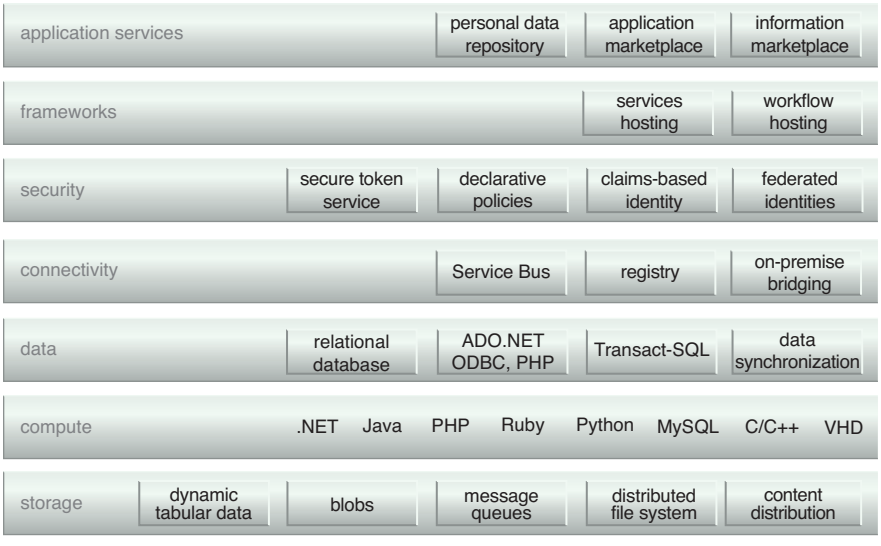


Figure 8.4
An overview of common Windows Azure platform capabilities.

The Windows Azure platform essentially provides a set of cloud-based services that are symmetric with existing mainstream on-site enterprise application platforms (Figure 8.4).

For example:

- *storage services* – a scalable distributed data storage system that supports many types of storage models, including hash map or table-like structured data, large binary files, asynchronous messaging queues, traditional file systems, and content distribution networks
- *compute services* – application containers that support existing mainstream development technologies and frameworks, including .NET, Java, PHP, Python, Ruby on Rails, and native code.

- *data services* – highly reliable and scalable relational database services that also support integration and data synchronization capabilities with existing on-premise relational databases
- *connectivity services* – these are provided via a cloud-based service bus that can be used as a message intermediary to broker connections with other cloud-based services and services behind firewalls within on-premise enterprise environments
- *security services* – policy-driven access control services that are federation-aware and can seamlessly integrate with existing on-premise identity management systems
- *framework services* – components and tools that support specific aspects and requirements of solution frameworks
- *application services* – higher-level services that can be used to support application development, such as application and data marketplaces

All of these capabilities can be utilized individually or in combination.

Windows Azure (Application Container)

Windows Azure serves as the development, service hosting, and service management environment. It provides the application container into which code and logic, such as Visual Studio projects, can be deployed. The application environment is similar to existing Windows Server environments. In fact, most .NET projects can be deployed directly without significant changes.

A Windows Azure instance represents a unit of deployment, and is mapped to specific virtual machines with a range of variable sizes. Physical provisioning of the Windows Azure instances is handled by the cloud fabric. We are required only to specify, by policy, how many instances we want the cloud fabric to deploy for a given service.

We have the ability to manually start and shut down instances, and grow or shrink the deployment pool; however, the cloud fabric also provides automated management of the health and lifecycles of instances. For example, in the event of an instance failure, the cloud fabric would automatically shut down the instance and attempt to bring it back up on another node.

Windows Azure also provides a set of storage services that consumers can use to store and manage persistent and transient data. Storage services support geo-location and offer high durability of data by triple-replicating everything within a cluster and across

data centers. Furthermore, they can manage scalability requirements by automatically partitioning and load balancing services across servers.

Also supported by Windows Azure is a VHD-based deployment model as an option to enable some IaaS requirements. This is primarily geared for services that require closer integration with the Windows Server OS. This option provides more control over the service hosting environment and can better support legacy applications.

SQL Azure

SQL Azure is a cloud-based relational database service built on SQL Server technologies that exposes a fault-tolerant, scalable, and multi-tenant database service. SQL Azure does not exist as hosted instances of SQL Server. It also uses a cloud fabric layer to abstract and encapsulate the underlying technologies required for provisioning, server administration, patching, health monitoring, and lifecycle management. We are only required to deal with logical administration tasks, such as schema creation and maintenance, query optimization, and security management.

A SQL Azure database instance is actually implemented as three replicas on top of a shared SQL Server infrastructure managed by the cloud fabric. This cloud fabric delivers high availability, reliability, and scalability with automated and transparent replication

SOA PRINCIPLES & PATTERNS

Services deployed within Windows Azure containers and made available via Windows Azure instances establish service architectures that, on the surface, resemble typical Web service or REST service implementations. However, the nature of the back-end processing is highly extensible and scalable and can be further subject to various forms of Service Refactoring [783] over time to accommodate changing usage requirements. This highlights the need for Windows Azure hosted services to maintain the freedom to be independently governed and evolved. This, in turn, places a greater emphasis on the balanced design of the service contract and its proper separation as part of the overall service architecture.

Specifically, it elevates the importance of the Standardized Service Contract (693), Service Loose Coupling (695), and Service Abstraction (696) principles that, through collective application, shape and position service contracts to maximize abstraction and cross-service standardization, while minimizing negative forms of consumer and implementation coupling. Decoupled Contract [735] forms an expected foundation for Windows Azure-hosted service contracts, and there will generally be the need for more specialized contract-centric patterns, such as Validation Abstraction [792], Canonical Schema [718], and Schema Centralization [769].

and failover. It further supports load-balancing of consumer requests and the synchronization of concurrent, incremental changes across the replicas. The cloud fabric also handles concurrency conflict resolutions when performing bi-directional data synchronization between replicas by using built-in policies (such as *last-writer-wins*) or custom policies.

SOA PRINCIPLES & PATTERNS

In addition to reliability and scalability improvements, SQL Azure's replication mechanism can be used to apply Service Data Replication [773] in support of the Service Autonomy (699) principle. This is significant, as individual service autonomy within cloud environments can often fluctuate due to the heavy emphasis on shared resources across pools of cloud-based services.

Because SQL Azure is built on SQL Server, it provides a familiar relational data model and is highly symmetric to on-premise SQL Server implementations. It supports most features available in the regular SQL Server database engine and can also be used with tools like SQL Server 2008 Management Studio, SQLCMD, and BCP, and SQL Server Integration Services for data migration.

Windows Azure Platform AppFabric

In Chapter 7, as part of our coverage of .NET Enterprise Services, we introduced Windows Server AppFabric. This represents the version of AppFabric that is local to the Windows Server environment. *Windows Azure platform AppFabric* (with the word "platform" intentionally not capitalized), is the cloud-based version of AppFabric that runs on Windows Azure.

Windows Azure platform AppFabric helps connect services within or across clouds and enterprises. It provides a Service Bus for connectivity across networks and organizational boundaries, and an Access Control service for federated authorization as a service.

The Service Bus acts as a centralized message broker in the cloud to relay messages between services and service consumers. It has the ability to connect to on-premise services through firewalls, NATs, and over any network topology.

Its features include:

- connectivity using standard protocols and standard WCF bindings
- multiple communication models (such as publish-and-subscribe, one-way messaging, unicast and multicast datagram distribution, full-duplex bi-directional connection-oriented sessions, peer-to-peer sessions, and end-to-end NAT traversal)

- service endpoints that are published and discovered via Internet-accessible URLs
- global hierarchical namespaces that are DNS and transport-independent
- built-in intrusion detection and protection against denial-of-service attacks

Access Control acts as a centralized cloud-based security gateway that regulates access to cloud-based services and Service Bus communications, while integrating with standards-based identity providers (including enterprise directories such as Active Directory and online identity systems like Windows Live ID). Access Control and other Windows Azure-related security topics are covered in Chapter 17.

SOA PRINCIPLES & PATTERNS

The Windows Azure Service Bus complies to the familiar Enterprise Service Bus [741] compound pattern, and focuses on realizing this pattern across network, security, and organizational domains.

Service Bus also provides a service registry to provide registration and discovery of service metadata, which allows for the application of Metadata Centralization [754] and emphasizes the need to apply the Service Discoverability (702) principle.

Unlike Windows Azure and SQL Azure, which are based on Windows Server and SQL Server, Access Control Service is not based on an existing server product. It uses technology included in Windows Identity Foundation and is considered a purely cloud-based service built specifically for the Windows Azure platform environment.

SUMMARY OF KEY POINTS

- The Windows Azure platform is primarily a PaaS deployed in a public cloud managed by Microsoft.
- Windows Azure platform provides a distinct set of capabilities suitable for building scalable and reliable cloud-based services.
- The overall Windows Azure platform further encompasses SQL Azure and Windows Azure platform AppFabric.

8.3 Windows Azure Roles

A cloud service in Windows Azure will typically have multiple concurrent instances. Each instance may be running all or a part of the service's codebase. As a developer, you control the number and type of roles that you want running your service.

Web Roles and Worker Roles

Windows Azure roles are comparable to standard Visual Studio projects, where each instance represents a separate project. These roles represent different types of applications that are natively supported by Windows Azure. There are two types of roles that you can use to host services with Windows Azure:

- Web roles
- worker roles

Web roles provide support for HTTP and HTTPS through public endpoints and are hosted in IIS. They are most comparable to regular ASP.NET projects, except for differences in their configuration files and the assemblies they reference.

Worker roles can also expose external, publicly facing TCP/IP endpoints on ports other than 80 (HTTP) and 443 (HTTPS); however, worker roles do not run in IIS. Worker roles are applications comparable to Windows services and are suitable for background processing.

Virtual Machines

Underneath the Windows Azure platform, in an area that you and your service logic have no control over, each role is given its own virtual machine or VM. Each VM is created when you deploy your service or service-oriented solution to the cloud. All of these VMs are managed by a modified hypervisor and hosted in one of Microsoft's global data centers.

Each VM can vary in size, which pertains to the number of CPU cores and memory. This is something that you control. So far, four pre-defined VM sizes are provided:

- small – 1.7ghz single core, 2GB memory
- medium – 2x 1.7ghz cores, 4GB memory
- large – 4x 1.7ghz cores, 8GB memory
- extra large – 8x 1.7ghz cores, 16GB memory

Notice how each subsequent VM on this list is twice as big as the previous one. This simplifies VM allocation, creation, and management by the hypervisor.

Windows Azure abstracts away the management and maintenance tasks that come along with traditional on-premise service implementations. When you deploy your service into Windows Azure and the service's roles are spun up, copies of those roles are

replicated automatically to handle failover (for example, if a VM were to crash because of hard drive failure). When a failure occurs, Windows Azure automatically replaces that "unreliable" role with one of the "shadow" roles that it originally created for your service.

This type of failover is nothing new. On-premise service implementations have been leveraging it for some time using clustering and disaster recovery solutions. However, a common problem with these failover mechanisms is that they are often server-focused. This means that the entire server is failed over, not just a given service or service composition.

When you have multiple services hosted on a Web server that crashes, each hosted service experiences downtime between the current server crashing and the time it takes to bring up the backup server. Although this may not affect larger organizations with sophisticated infrastructure too much, it can impact smaller IT enterprises that may not have the capital to invest in setting up the proper type of failover infrastructure.

Also, suppose you discover in hindsight after performing the failover that it was some background worker process that caused the crash. This probably means that unless you can address it quick enough, your failover server is under the same threat of crashing.

Windows Azure addresses this issue by focusing on application and hosting roles. Each service or solution can have a Web frontend that runs in a Web role. Even though each role has its own "active" virtual machine (assuming we are working with single instances), Windows Azure creates copies of each role that are physically located on one or more servers. These servers may or may not be running in the same data center. These shadow VMs remain idle until they are needed.

Should the background process code crash the worker role and subsequently put the underlying virtual machine out of commission, Windows Azure detects this and automatically brings in one of the shadow worker roles. The faulty role is essentially discarded. If the worker role breaks again, then Windows Azure replaces it once more. All of this is happening without any downtime to the solution's Web role front end, or to any other services that may be running in the cloud.

Input Endpoints

Web roles used to be the only roles that could receive Internet traffic, but now worker roles can listen to any port specified in the service definition file. Internet traffic is received through the use of *input endpoints*. Input endpoints and their listening ports are declared in the service definition (*.csdef) file.

Keep in mind that when you specify the port for your worker role to listen on, Windows Azure isn't actually going to assign that port to the worker. In reality, the load balancer will open two ports—one for the Internet and the other for your worker role. Suppose you wanted to create an FTP worker role and in your service definition file you specify port 21. This tells the fabric load balancer to open port 21 on the Internet side, open pseudo-random port 33476 on the LAN side, and begin routing FTP traffic to the FTP worker role.

In order to find out which port to initialize for the randomly assigned internal port, use the `RoleEnvironment.CurrentRoleInstance.InstanceEndpoints["FtpIn"].IPEndpoint` object.

Inter-Role Communication

Inter-Role Communication (IRC) allows multiple roles to talk to each other by exposing internal endpoints. With an internal endpoint, you specify a name instead of a port number. The Windows Azure application fabric will assign a port for you automatically and will also manage the name-to-port mapping.

Here is an example of how you would specify an internal endpoint for IRC:

```
<ServiceDefinition xmlns=
  "http://schemas.microsoft.com/ServiceHosting/2008/10/
  ServiceDefinition" name="HelloWorld">
  <WorkerRole name="WorkerRole1">
    <Endpoints>
      <InternalEndpoint name="NotifyWorker" protocol="tcp" />
    </Endpoints>
  </WorkerRole>
</ServiceDefinition>
```

Example 8.1

In this example, `NotifyWorker` is the name of the internal endpoint of a worker role named `WorkerRole1`. Next, you need to define the internal endpoint, as follows:

```
RoleInstanceEndpoint internalEndPoint =
RoleEnvironment.CurrentRoleInstance.
  InstanceEndpoints["NotificationService"];
this.serviceHost.AddServiceEndpoint(
  typeof(INameOfYourContract),
  binding,
```

```
  String.Format("net.tcp://{0}/NotifyWorker",
    internalEndPoint.IPEndpoint));
WorkerRole.factory = new ChannelFactory<IClientNotification>(binding);
```

Example 8.2

You only need to specify the IP endpoint of the other worker role instances in order to communicate with them. For example, you could get a list of these endpoints with the following routine:

```
var current = RoleEnvironment.CurrentRoleInstance;
var endPoints = current.Role.Instances
  .Where(instance => instance != current)
  .Select(instance => instance.InstanceEndpoints["NotifyWorker"]);
```

Example 8.3

IRC only works for roles in a single application deployment. Therefore, if you have multiple applications deployed and would like to enable some type of cross-application role communication, IRC won't work. You will need to use queues instead.

SUMMARY OF KEY POINTS

- Windows Azure roles represent different types of supported applications or services.
- There are two types of roles: Web roles and worker roles.
- Each role is assigned its own VM.

8.4 Hello World in Windows Azure

The following section demonstrates the creation of a simple "Hello World" service in a Windows Azure hosted application.

NOTE

If you are carrying out the upcoming steps with Visual Studio 2008, you will need to be in an elevated mode (such as Administrator). A convenient way of determining whether the mode setting is correct is to press the F5 key in order to enter debug mode. If you receive an error stating *"the development fabric must be run elevated,"* then you will need to restart Visual Studio as an administrator.

Also, ensure the following on your SQL Express setup:

- SQL Server Express Edition 2008 must be running under the '.\SQL-EXPRESS' instance
- your Windows account must have a login in .\SQLEXPRESS
- your login account is a member of the sysadmin role

If SQL Express isn't configured properly, you will get a permissions error.

1. Create a Cloud Service Project

First you need to open the New Project window to create a new cloud service project using VB.NET or C# (Figure 8.5).

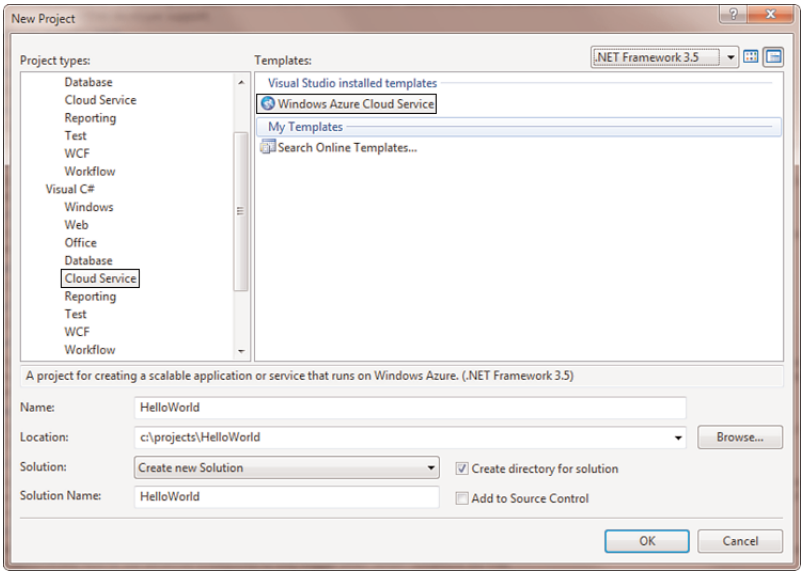


Figure 8.5
The New Project window.

2. Choose an ASP.NET Web Role

After you click OK on the New Project window, the New Cloud Service Project wizard will start. You will then see a window (Figure 8.6) that will allow you to choose the type of role that you would like as part of your service deployment.

For the Hello World project, you will only need the ASP.NET Web Role type. Once you select this role, you can choose the role name.

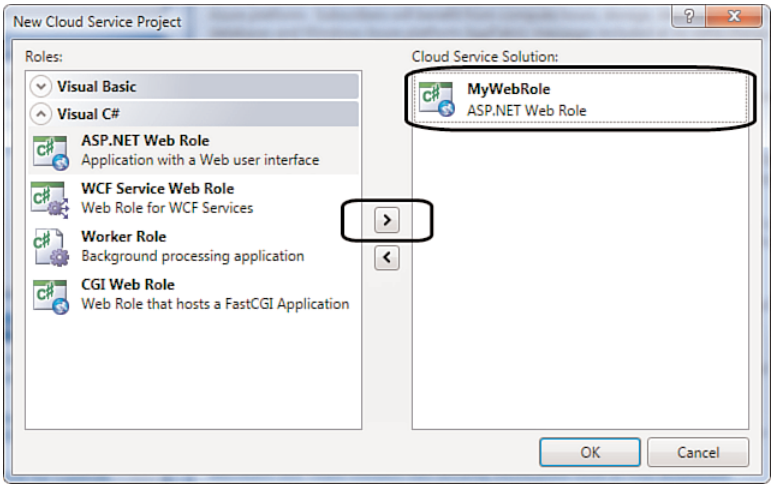


Figure 8.6
The New Cloud Service Project window.

3. Create the Solution

After clicking OK, the wizard will generate the solution, which you can then view using the Solution Explorer window (Figure 8.7).

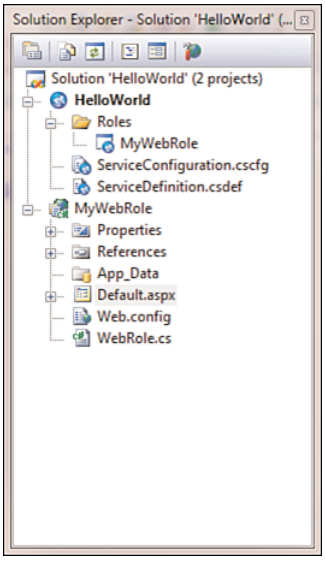


Figure 8.7
The HelloWorld solution structure displayed in the Solution Explorer window.

4. Instantiate the Service

Now you can open the Default.aspx file using the Solution Explorer window, put "Hello, Cloud!" in the `Body` element and press F5 to run. You should see something like what is shown in Figure 8.8.

This example was executed locally on IIS. If we were to deploy this service into the Windows Azure cloud, it would still be running in IIS because it is hosted in a Web role.

SOA PRINCIPLES & PATTERNS

Mainstream SOA design patterns and service-orientation principles can be applied to Windows Azure-hosted services very similarly to how they are applied to internal enterprise-hosted services. Furthermore, Windows Azure-hosted services support different service implementation mediums (such as Web services and REST services) and allow for the same service to be accessed via multiple protocols. This supports the creative application of specialized patterns, such as Concurrent Contracts [726] and Dual Protocols [739].

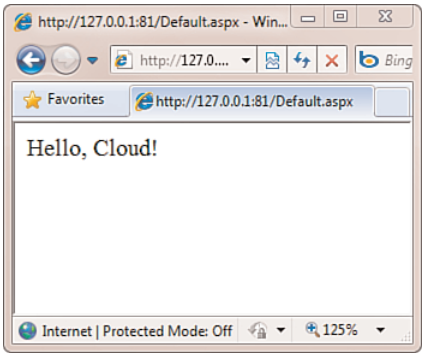


Figure 8.8
The HelloWorld service in action.

SUMMARY OF KEY POINTS

- The development environment for Windows Azure is fully integrated into Visual Studio, which provides a simulated runtime for Windows Azure for local desktop-based development and unit testing.
- Creating and deploying cloud-based services with Windows Azure is simplified using available wizards and development UIs.

8.5 A Web Service in Windows Azure

In this section example, we take a closer look at a Web service that is deployed to Windows Azure in order to better understand the code-level impacts of moving a service to a cloud.

Let's assume we moved a service contact interface definition and a data contract into a custom C# project. We choose ServiceClient to test our service and ServiceDemo contains the Windows Azure application configuration and definition files.

We further opt to host this project in a Web role, which means that there is a little bit of bootstrapping that needs to be done. The `WebRole` class inherits from the `RoleEntryPoint` class, which contains methods that are used by Windows Azure to start or stop the role. You can optionally override those methods to manage the initialization or shutdown process of your role. Worker roles must extend `RoleEntryPoint`, but it is optional for Web roles. The Visual Studio tools will automatically extend this class for you, as you can see from the WebRole.cs code:

```
using System.Linq;
using Microsoft.WindowsAzure.Diagnostics;
using Microsoft.WindowsAzure.ServiceRuntime;
using System.Diagnostics;
namespace ServiceDemo_WebRole
{
  public class WebRole : RoleEntryPoint
  {
    public override bool OnStart()
    {
      DiagnosticMonitor.Start("DiagnosticsConnectionString");
      RoleEnvironment.Changing += RoleEnvironmentChanging;
      Trace.TraceInformation("WebRole starting...");
      return base.OnStart();
    }
    private void RoleEnvironmentChanging(object sender,
      RoleEnvironmentChangingEventArgs e)
    {
      if (e.Changes.Any(change => change is
        RoleEnvironmentConfigurationSettingChange))
      {
        e.Cancel = true;
      }
    }
  }
}
```

Example 8.4

Our cloud service project includes two configuration files: ServiceDefinition.csdef and ServiceConfiguration.cscfg. These files are packaged together with the cloud service when it is deployed to Windows Azure.

The ServiceDefinition.csdef file contains the metadata needed by the Windows Azure environment to understand the requirements of the service, including the roles it contains. It also establishes configuration settings that will be applied to all specified service roles:

```
<ServiceDefinition name="ServiceDemo" xmlns=
  "http://schemas.microsoft.com/ServiceHosting/2008/10/
  ServiceDefinition">
  <WebRole name="ServiceDemo_WebRole">
    <InputEndpoints>
      <InputEndpoint name=
        "HttpIn" protocol="http" port="80" />
    </InputEndpoints>
    <ConfigurationSettings>
      <Setting name="DiagnosticsConnectionString" />
    </ConfigurationSettings>
  </WebRole>
</ServiceDefinition>
```

Example 8.5

The ServiceConfiguration.cscfg file sets values for the configuration settings defined in the service definition file and specifies the number of instances to run for each role. Here is the ServiceConfiguration.cscfg for the ServiceDemo service project:

```
<ServiceConfiguration serviceName="ServiceDemo"
  xmlns="http://schemas.microsoft.com/
  ServiceHosting/2008/10/ServiceConfiguration">
  <Role name="ServiceDemo_WebRole">
    <Instances count="2" />
    <ConfigurationSettings>
      <Setting name="DiagnosticsConnectionString"
        value="UseDevelopmentStorage=true" />
    </ConfigurationSettings>
  </Role>
</ServiceConfiguration>
```

Example 8.6

The `Instances` element tells the Windows Azure runtime fabric how many instances to spin up for the ServiceDemo_WebRole role. By default, Visual Studio tools set this to "1", but this is generally not a good idea. If you only have one role running and it crashes, it could take a while before Windows Azure spins up another one. However, if you had multiple roles and one goes down, the application wouldn't experience a work stop while a new instance is being generated. This is why it is a good practice to have at least two role instances per role.

In the `ConfigurationSettings` section, there is a statement worth singling out:

```
<Setting name="DiagnosticsConnectionString"
  value="UseDevelopmentStorage=true" />
```

Example 8.7

There is a set of logging and diagnostic APIs that you can use to instrument your code and provide better traceability. With these APIs, you can not only detect and troubleshoot problems, but you can also gain insight into the overall performance of an application.

This line of code passes in the configuration setting name that is equal to the connection string for the storage account that the Diagnostic Monitor needs to use to store the diagnostic data. By default, the setting name is `DiagnosticsConnectionString`, but you can name it whatever you like as long as the name matches up with the service definition and service configuration files.

In the WebRole.cs, you will see the following statement:

```
DiagnosticMonitor.Start("DiagnosticsConnectionString");
```

Example 8.8

This line of code starts up the Diagnostic Monitor when the role starts. By default, the connection string is set to use development storage, such as the SQL table that was created when the SDK was installed. Before you deploy the service to the Windows Azure cloud, you will need to update this setting with the storage account name and account key information.

For example:

```
<ConfigurationSettings>
  <Setting name="DiagnosticsConnectionString"
    value="DefaultEndpointsProtocol=https;AccountName=
    [ACCOUNT NAME};AccountKey=[ACCOUNT KEY]" />
</ConfigurationSettings>
```

Example 8.9

If we take a look at the Web role's Web.Config file, we'll also see that the project wizard automatically created the following:

```
<system.diagnostics>
  <trace>
    <listeners>
      <addtype="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>
```

Example 8.10

This creates a tracing listener for the diagnostic monitor, which means that we continue to use the `System.Diagnostics.Trace` class for instrumentation. The diagnostic monitor will just hook into those calls and push them into storage.

The following examples show the `IOrderService` interface contract and the Order data contract, followed by the final output:

```
namespace Contract
{
  [ServiceContract]
  public interface IOrderService
  {
    [OperationContract]
    int CreateOrder(Order o);
    [OperationContract]
    void UpdateOrder(string id, Order o);
    [OperationContract]
```

```
    Order GetOrderByOrderId(string id);
    [OperationContract]
    List<Order> GetOrdersByCustomer(string custName);
    [OperationContract]
    List<Order> GetOrders();
    [OperationContract]
    void DeleteOrder(string id);
  }
}
```

Example 8.11

```
namespace Contract
{
[DataContract(Namespace=
  "http://example.cloudapp.net/servicedemo/1.0")]
  public class Order
  {
    [DataMember]
    public int OrderId { get; set; }
    [DataMember]
    public string OrderItem { get; set; }
    [DataMember]
    public string CustomerName { get; set; }
  }
}
```

Example 8.12

```
namespace ServiceDemo_WebRole
{
  [ServiceBehavior(InstanceContextMode =
    InstanceContextMode.Single,
    AddressFilterMode =
    AddressFilterMode.Any)]
  public class OrderService : Contract.IOrderService
  {
    int id = 0;
    List<Order> Orders = new List<Order>();
    #region IOrderService Members
    int IOrderService.CreateOrder(Order o)
    {
      o.OrderId = ++id;
      Orders.Add(o);
      return o.OrderId;
```

```
    }
    void IOrderService.UpdateOrder(string id, Order o)
    {
      var first = Orders.First(order =>
        order.OrderId ==
        Convert.ToInt64(id));
      first = o;
    }
    List<Order> IOrderService.GetOrders()
    {
      return Orders;
    }
    void IOrderService.DeleteOrder(string orderId)
    {
      Orders.RemoveAll(order =>
        order.OrderId.Equals
        (Convert.ToInt64(orderId)));
    }
    Order IOrderService.GetOrderByOrderId(string orderId)
    {
      return Orders.First(o =>
        o.OrderId.Equals(Convert.ToInt64(orderId)));
    }
    public List<Order> GetOrdersByCustomer(string custName)
    {
      return (string.IsNullOrEmpty(custName))?
        Orders : Orders.FindAll(o =>
        o.CustomerName.Equals(custName));
    }
    #endregion
  }
}
```

Example 8.13

Note that the `InstanceContextMode` setting is set to to single because we want to use the same service object instance across the communication session established between the service and its consumer. In a real world scenario, you would choose a more robust solution like SQL Azure or Windows Azure table storage (covered later in this chapter).

Let's briefly walk through the steps required to actually deploy the service to Windows Azure.

1. Create a Host Service and Storage Service

When you create a storage service, you have to create a globally unique storage account name, not to be confused with the overarching Windows Azure account that is mapped to your Windows LiveID. For our example, we chose juggercloud as the account name and received three storage endpoints. Two access keys are also generated.

Before we deploy our Web service, however, we will update the Web role service configuration *.cscfg file with the account name and account key information, as follows:

```
<ServiceConfiguration serviceName="StandardMoldHost"
  xmlns="http://schemas.microsoft.com/
  ServiceHosting/2008/10/ServiceConfiguration">
  <Role name="ServiceDemo_WebRole">
    <Instances count="2" />
      <ConfigurationSettings>
       <Setting name="DiagnosticsConnectionString"
         value="DefaultEndpointsProtocol=https;
         AccountName=standardmold;AccountKey=01g82Oj...==" />
      </ConfigurationSettings>
   </Role>
</ServiceConfiguration>
```

Example 8.14

2. Create and Deploy a Service Package

We deploy the service by uploading a package through the Windows Azure portal. When using the Windows Azure UI, we can navigate to the host service to determine whether we are deploying to staging or production.

There's really no difference in hardware resource configuration between these two settings. In fact, the separation between the two environments is managed through the network load balancer's routing tables.

Once we click "Deploy," the package and configuration file will be uploaded.

NOTE

We could have also pulled these bits from a Windows Azure storage account. For example, we could create a custom MSBuild task leveraged within a Team Foundation Server Team Build definition file. Instead of dropping the package to a normal file drop, this would upload it into blob storage using the REST API, or perhaps even leverage Windows Azure Drive.

3. Promote the Service to Production

Let's imagine the previous step initially deployed the service to staging so that we could test it before moving it into the production enviroment. The Windows Azure UI allows you to invoke the service by clicking "Run," resulting in a page similar to Figure 8.9.

Figure 8.9

After verifying that the Web service is performing as desired, it can be deployed to production (Figure 8.10).

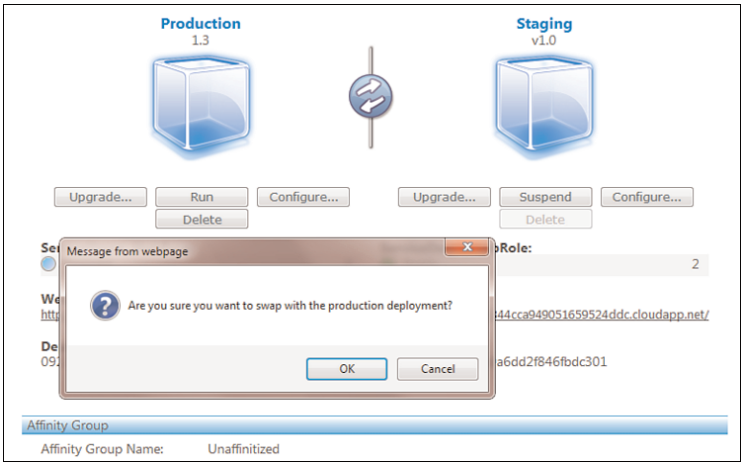


Figure 8.10

8.6 A REST Service in Windows Azure

In order to explore how REST services are created and exist within Windows Azure, this section takes the Web service from the previous section and makes it RESTful. But, before we dive into the implementation details of this change, let's first take a step back and think about REST-specific design considerations.

REST Service Addressing

A common design practice with REST services is to make the addressing (the manner in which target resources are addressed) as intuitive as possible. The social bookmarking site Delicious is a great example of this.

With Delicious, every bookmark has one or more tags (think of tags as categories). Tags essentially replace folders within Web browsers with categories. In relation to our discussion, you can also group tags into a bundle, which basically creates "tag clouds." Access to tagged bookmarks is provided via REST services. Table 8.1 shows a set of sample URLs that can be used to get back a list of bookmarks for Azure, SOA, and Azure+SOA, respectively.

URL	Description
http://delicious.com/tag/azure	returns a list of bookmarks that have been tagged with Azure
http://delicious.com/tag/soa	returns a list of bookmarks that have been tagged with SOA
http://delicious.com/tag/soa+azure	returns a list of bookmarks that have been tagged with SOA and Azure

Table 8.1
Sample URLs used to retrieve different values from REST services at delicious.com.

What's important about this example is that we are able to search, create and update a large network of data via REST without writing code. The HTTP GET method and the appropriate URLs are all we need.

Returning to our Order service, we first need to define an appropriate resource addressing structure for the order data, as shown in Table 8.2.

Action	IOrderService Operation Name	URI Address Template	HTTP Method
get a list of all orders	GetOrders	./orders	GET
get an order given the order ID	GetOrderByOrderId	./order/{id}	GET
get a list of orders for a given customer	GetOrdersByCustomer	./orders/{custName}	GET
create an order	CreateOrder	./orders	POST
update an order	UpdateOrder	./order/{id}	PUT
delete an order	DeleteOrder	./order/{id}	DELETE

Table 8.2
The resource addressing structure for the Order service.

Creating a Windows Azure REST Service

We now need to carry out a series of steps to make this a REST service:

1. Add a reference to `System.ServiceModel.Web` in the Contract project.
2. Add HTTP attributes to the methods defined in the `IOrderService` interface.
3. Update the WCF behavior.
4. Update the OrderService.svc file by adding a Web factory reference.

The `System.ServiceModel.Web` namespace contains classes that make up the Web HTTP programming model.

For our purposes, we need to focus on the following:

- `WebGetAttribute` (maps to an HTTP GET)
- `WebInvokeAttribute` (maps to HTTP POST, PUT, and DELETE)
- `WebMessageFormat` (defines the format of the response message)

For the GET method, we use the `WebGet` attribute. We then use the `UriTemplate` attribute to define the addressing structure from Table 8.2. This is a manual process, which means that it's easy to make mistakes. It is therefore important to lay out the URI structure prior to working with the code.

We also need to specify the `{token}` parameters. For example, if we were calling the `GetOrderByOrderId` operation of the Web service via SOAP, we would just pass in the order ID argument by calling the Web method. But with REST, everything is through HTTP methods and URIs. The service consumer doesn't call `GetOrderByOrderId` directly, but rather does the HTTP GET method on http://server/OrderService.svc/order/2, where "2" is the order ID value.

Next, we need to determine the response message format by setting `ResponseFormat` to return XML messages.

Here's what `IOrderService` looks like now:

```
 [ServiceContract]
public interface IOrderService
{
  [WebInvoke(Method="POST",
    UriTemplate="orders",
    ResponseFormat=WebMessageFormat.Xml)]
  [OperationContract]
  int CreateOrder(Order o);
  [WebInvoke(Method="PUT",
    UriTemplate="order/{id}",
    ResponseFormat=WebMessageFormat.Xml)]
  [OperationContract]
  void UpdateOrder(string id, Order o);
  [WebGet(UriTemplate="order/{id}",
    ResponseFormat=WebMessageFormat.Xml)]
  [OperationContract]
  Order GetOrderByOrderId(string id);
  [WebGet(UriTemplate="orders/{custName}",
    ResponseFormat=WebMessageFormat.Xml)]
  [OperationContract]
  List<Order> GetOrdersByCustomer(string custName);
  [WebGet(UriTemplate="orders",
    ResponseFormat=WebMessageFormat.Xml)]
  [OperationContract]
  List<Order> GetOrders();
  [WebInvoke(Method="DELETE",
    UriTemplate="order/{id}",
    ResponseFormat=WebMessageFormat.Xml)]
  [OperationContract]
  void DeleteOrder(string id);
}
```

Example 8.15

We now need to update the WCF behavior in the Web.Config file by changing the endpoint binding to `WebHttpBinding` and the endpoint behavior to a Web behavior, as shown here:

```
<services>
  <servicebehaviorConfiguration=
    "ServiceDemo_WebRole.ServiceDemoBehavior"
    name="ServiceDemo_WebRole.OrderService">
    <endpoint address="" binding="WebHttpBinding"
      contract="Contract.IOrderService"
      behaviorConfiguration="Web">
    </endpoint>
  </service>
</services>
<behaviors>
  <endpointBehaviors>
    <behavior name="Web" />
  </endpointBehaviors>
  <serviceBehaviors>
    <behavior name=
      "ServiceDemo_WebRole.ServiceDemoBehavior">
  </serviceBehaviors>
</behaviors>
```

Example 8.16

Finally, we have to update the OrderService.svc file to include `WebServiceHostFactory`, as shown here:

```
<%@
  ServiceHost Language="C#" Debug="true"
  Service="ServiceDemo_WebRole.OrderService"
  CodeBehind="OrderService.svc.cs"
  Factory="System.ServiceModel.
    Activation.WebServiceHostFactory"
%>
```

Example 8.17

`WebServiceHostFactory` provides instances of `WebServiceHost` in managed hosting environments, where the host instance is created dynamically in response to incoming messages. This is necessary because the service is being hosted using a Web role in IIS.

SOA PRINCIPLES & PATTERNS

Cloud-based REST service architecture relates to several SOA design patterns relevant to REST service design. These are covered separately in the book *SOA with REST* as part of the *Prentice Hall Service-Oriented Computing Series with Thomas Erl.*

Finally, to deploy the REST version of the Order service to Windows Azure, we can follow the same steps described in the previous *A Web Service in Windows Azure* section.

SUMMARY OF KEY POINTS

- Programming models and deployment processes for Web services are very similar and consistent between cloud-based services in Windows Azure and on-premise services in Windows Server.
- Most significant differences with cloud-based services in Windows Azure are managed via service configurations.
- Development and deployment of REST-based services in Windows Azure are also consistent with the on-premise platform.

8.7 Windows Azure Storage

Windows Azure provides the following set of storage services (collectively referred to as Windows Azure Storage), each of which is suitable for different types of data access requirements:

- *Tables* provide structured storage, as they do in regular databases. Essentially, each table consists of a set of data entities that each contain a set of properties.
- *Queues* provide reliable storage and delivery of messages. They are often used between roles to communicate with each other.
- *Blobs* are used to store large binary objects (files). They provide a simple interface for storing named files along with metadata and include support for CDN (Content Delivery Network).
- *Windows Azure Drives* provide durable NTFS volumes for Windows Azure applications.

Windows Azure Storage supplies a managed API and a REST API, both of which essentially provide the same level of functionality. The managed API is provided through the Microsoft.`WindowsAzure.StorageClient` namespace. To interact with the storage services, you can also use familiar programming interfaces, such as ADO.NET Data Services (available in the .NET framework version 3.5 SP1).

Note that access to storage is regulated via Windows Azure Storage accounts that use 256-bit secret keys. Also, there are some storage size limitations. For example, each storage account will have a maximum 100 terabytes of total storage capacity.

Tables

Windows Azure Tables (WATs) are similar to relational tables insofar as they both are used to store structured data. However, it's important to understand that WAT storage is not a relational database management system for the cloud (that's what SQL Azure is for). In other words, there is no support for common database features, such as joins, aggregates, stored procedures, or indexes.

WATs were built primarily to realize scalability, availability, and durability of data. Individual tables can be scaled to billions of entities (rows) with data totaling into the order of terabytes. Part of the scaling algorithm is that as application traffic and usage grows, WATs will automatically scale out to potentially tens, to hundreds, to thousands of servers. With regards to availability, each WAT is replicated at least three times.

Entities and Properties

Windows Azure Storage introduces some specific terminology and relationships for WATs:

- You create a *storage account*, each of which can have multiple *tables*.
- Data stored within a table is organized into *entities*. A database row is comparable to an entity.
- Each entity contains a set of *properties*. A database column is comparable to a property.
- A table is comprised of a set of entities, each of which is comprised of a set of properties.

Each entity contains two key properties that together form the unique ID of the entity in that table. The first key is the *PartitionKey*, which allows you to group entities together.

This tells the Windows Azure Storage system to not split this group up when scaling out the table.

In other words, partition keys are used to group table entities into partitions that provide a unit of scale that Windows Azure Storage uses to properly load balance data. Partition keys also allow you to control the physical locality of the entity data. Everything within a partition will live on a single server.

The second key is the *RowKey*, which provides uniqueness within a partition in that the PartitionKey together with the RowKey uniquely identify a given entity (as well as the sort order). You can think of these two keys as a clustered index for a table.

The third required attribute is the *Timestamp*, which is a read-only attribute used to control optimistic concurrency. That is, if you try to update a row that another program has already updated, your update attempt will fail because of the timestamp mismatch.

Data Access

When interacting with entities and properties, you are provided the full range of regular data access functions (get, insert, update, delete), in addition to special features, such as the partial update (merge), the entire update (replace), and the entity group transaction.

Entity group transactions allow you to atomically perform multiple insert, update, and delete commands over a set of entities in the same partition as part of a single transaction.

Queues

As with traditional messaging queues, the Windows Azure queues provide a reliable intermediary mechanism for delivering messages. For example, a common scenario is to set up a queue as the communication proxy between an application's Web role (of which there may be one or two instances) and its worker roles (of which there can be many instances). For this scenario you would likely set up at least two queues. The first would allow the Web role to submit messages for the worker roles to process. The worker roles would poll the queue for new messages until one is received. The second queue would then be for the worker roles to communicate back to the Web role. This architecture allows the Web role to delegate and spread out resource-intensive work to the worker roles.

Just like with tables, queues are scoped by the storage account that you create. An account can have many queues, each of which can contain an unlimited amount of

messages. Also, dequeued counts are tracked, allowing you to determine how often a given message has been dequeued by a worker process.

Queues offer a range of data access functions, including the ability to create, delete, list, and get/set queued metadata. Additionally, you can add (enqueue) and retrieve (dequeue) sets of messages, and delete and "peek" at messages individually.

Blobs

Each storage account can have containers that can be used to store blobs. There is no limit to the number of containers that you can have as long as they will fit into your storage account limit.

Containers have the ability to set public or private access policies. The private access level will only allow access to consumers that have been given permission. Public access allows any consumer to interact with the container's blobs using a URL. You can also have container metadata, which, like blob metadata, is stored in name-value pairs.

You have two choices for the type of blob that you can use: block and page. Both types have characteristics that make them applicable to specific requirements.

Block Blobs

A block blob is primarily geared towards streaming media files. Each blob is organized into a sequential list of "blocks" that can be created and uploaded out of order and in parallel for increased performance. Once uploaded, each block is in an uncommitted state, meaning that you cannot access the blob until its blocks are committed. To commit the blocks as well as define the correct block order, you use the `PutBlobList` command.

Each block is immutable and is further defined by a block ID. After you have successfully uploaded a block, that block (identified by its block ID) cannot be changed. That also means that if you have updated a block on-premise, then you will need to re-upload or copy the entire block with the same block ID.

Blobs can be accessed via an available REST API that provides standard data access operations, as well as special functions, such as *CopyBlob* that allows you to copy an existing blob to a new blob name.

Page Blobs

Page blobs are suitable for random I/O operations. With this kind of blob, you must first pre-allocate space (up to 1TB), wherein the blob is divided into 512-byte "pages." To access or update a page, you must address it using a byte offset. Another key difference is that changes to page blobs are immediate.

You can expand the blob size at any point by increasing its maximum size setting. You are also allowed to shrink the blob by truncating pages. You can update a page in one of two ways: *PutPage* or *ClearPage*. With PutPage, you specify the payload and the range of pages, whereas ClearPage basically zeroes out a page range up to the entire blob. There are several other commands that can be used to work with page blobs.

Windows Azure Drive

Windows Azure Drive is a storage service that provides a durable NTFS volume for Windows Azure applications. An application needs to mount the volume prior to using it and, when done, the application then unmounts the same volume. Throughout this period, the volume data is kept intact, even if the application should crash.

A Windows Azure Drive volume is actually a page blob. Specifically, it exists as a page blob that has been formatted as an NTFS single volume virtual hard drive (VHD). As such, these drives can be up to 1TB in size.

SUMMARY OF KEY POINTS

- Windows Azure Storage provides a set of services for distributed cloud-based data storage.
- The four types of storage services provided are tables, queues, blobs, and Windows Azure Drive.
- Windows Azure Storage services are available via both .NET managed APIs and REST-based APIs.