DavidChappell
& Associates

INTRODUCING THE AZURE SERVICES PLATFORM

AN EARLY LOOK AT WINDOWS AZURE, .NET SERVICES, SQL SERVICES, AND LIVE SERVICES

DAVID CHAPPELL

OCTOBER 2008

SPONSORED BY MICROSOFT CORPORATION

CONTENTS

AN OVERVIEW OF THE AZURE SERVICES PLATFORM

Using computers in the cloud can make lots of sense. Rather than buying and maintaining your own machines, why not exploit the acres of Internet-accessible servers on offer today? For some applications, their code and data might both live in the cloud, where somebody else manages and maintains the systems they use. Alternatively, applications that run inside an organization—on-premises applications—might store data in the cloud or rely on other cloud infrastructure services. Applications that run on desktops and mobile devices can use services in the cloud to synchronize information across many systems or in other ways. However it's done, exploiting the cloud's capabilities can improve our world.

But whether an application runs in the cloud, uses services provided by the cloud, or both, some kind of application platform is required. Viewed broadly, an application platform can be thought of as anything that provides developer-accessible services for creating applications. In the local, on-premises Windows world, for example, this includes technologies such as the .NET Framework, SQL Server, and more. To let applications exploit the cloud, cloud application platforms must also exist. And because there are a variety of ways for applications to use cloud services, different kinds of cloud platforms are useful in different situations.

Microsoft's Azure Services Platform is a group of cloud technologies, each providing a specific set of services to application developers. As Figure 1 shows, the Azure Services Platform can be used both by applications running in the cloud and by applications running on local systems.

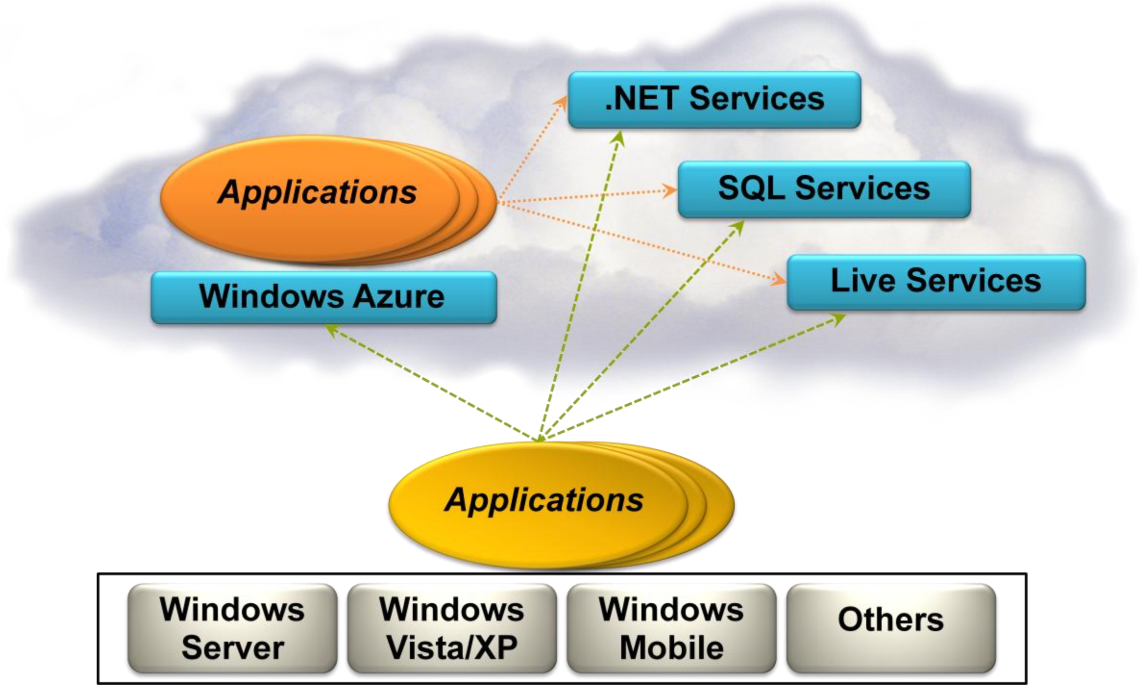

Figure 1: The Azure Services Platform supports applications running in the cloud and on local systems.

The components of the Azure Services Platform can be used by local applications running on a variety of systems, including various flavors of Windows, mobile devices, and others. Those components include:

- *Windows Azure*: Provides a Windows-based environment for running applications and storing data on servers in Microsoft data centers.
- *Microsoft .NET Services*: Offers distributed infrastructure services to cloud-based and local applications.
- *Microsoft SQL Services*: Provides data services in the cloud based on SQL Server.
- *Live Services*: Through the Live Framework, provides access to data from Microsoft's Live applications and others. The Live Framework also allows synchronizing this data across desktops and devices, finding and downloading applications, and more.

Each component of the Azure Services Platform has its own role to play. This overview describes all four, first at a high level, then in a bit more detail. While none of them are yet final—details and more might change before their initial release—it's not too early to start understanding this new set of platform technologies.

WINDOWS AZURE

At a high level, Windows Azure is simple to understand: It's a platform for running Windows applications and storing their data in the cloud. Figure 2 shows its main components.

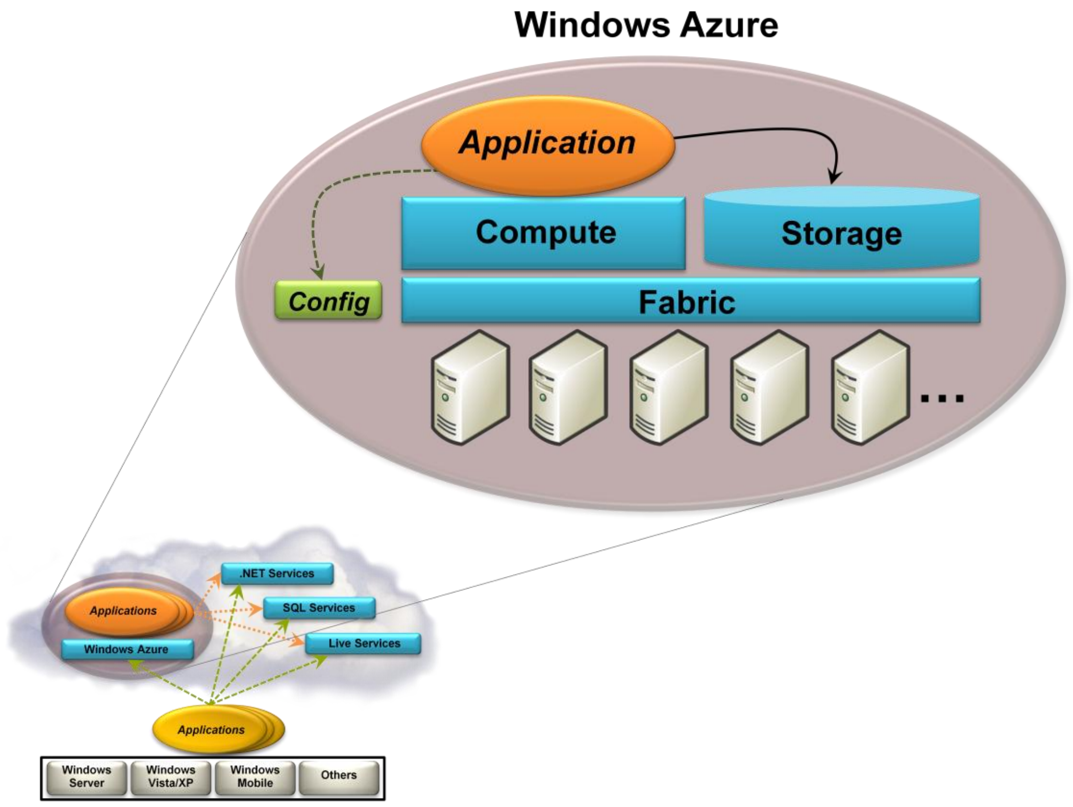

Figure 2: Windows Azure provides Windows-based compute and storage services for cloud applications.

As the figure suggests, Windows Azure runs on a large number of machines, all located in Microsoft data centers and accessible via the Internet. A common Windows Azure fabric knits this plethora of processing power into a unified whole. Windows Azure compute and storage services are built on top of this fabric.

The Windows Azure compute service is based, of course, on Windows. For the initial availability of this service, a Community Technology Preview (CTP) made public in the fall of 2008, Microsoft allowed Windows Azure to run only applications built on the .NET Framework. The company has announced plans to support unmanaged code as well, i.e., applications that aren't built on the .NET Framework, on Windows Azure in 2009.

In the CTP version of Windows Azure, developers can create .NET-based software such as ASP.NET applications and Windows Communication Foundation (WCF) services. To do this, they can use C# and other .NET languages, along with traditional development tools such as Visual Studio 2008. And while many developers are likely to use this initial version of Windows Azure to create Web applications, the platform also supports background processes that run independently—it's not solely a Web platform.

Both Windows Azure applications and on-premises applications can access the Windows Azure storage service, and both do it in the same way: using a RESTful approach. The underlying data store is not Microsoft SQL Server, however. In fact, Windows Azure storage isn't a relational system, and its query language isn't SQL. Because it's primarily designed to support applications built on Windows Azure, it provides simpler, more scalable kinds of storage. Accordingly, it allows storing binary large objects (blobs), provides queues for communication between components of Windows Azure applications, and even offers a form of tables with a straightforward query language.

Running applications and storing their data in the cloud can have clear benefits. Rather than buying, installing, and operating its own systems, for example, an organization can rely on a cloud provider to do this for them. Also, customers pay just for the computing and storage they use, rather than maintaining a large set of servers only for peak loads. And if they're written correctly, applications can scale easily, taking advantage of the enormous data centers that cloud providers offer.

Yet achieving these benefits requires effective management. In Windows Azure, each application has a configuration file, as shown in Figure 2. By changing the information in this file manually or programmatically, an application's owner can control various aspects of its behavior, such as setting the number of instances that Windows Azure should run. The Windows Azure fabric monitors the application to maintain this desired state.

To let its customers create, configure, and monitor applications, Windows Azure provides a browser-accessible portal. A customer provides a Windows Live ID, then chooses whether to create a *hosting* account for running applications, a *storage* account for storing data, or both. An application is free to charge its customers in any way it likes: subscriptions, per-use fees, or anything else.

Windows Azure is a general platform that can be used in various scenarios. Here are a few examples, all based on what the CTP version allows:

- A start-up creating a new Web site—the next Facebook, say—could build its application on Windows Azure. Because this platform supports both Web-facing services and background processes, the application can provide an interactive user interface as well as executing work for users asynchronously. Rather than spending time and money worrying about infrastructure, the start-up can instead focus solely on creating code that provides value to its users and investors. The company can also start small, incurring low costs while its application has only a few users. If their application catches on and usage increases, Windows Azure can scale the application as needed.

- An ISV creating a software-as-a-service (SaaS) version of an existing on-premises .NET application might choose to build it on Windows Azure. Because Windows Azure mostly provides a standard .NET environment, moving the application's .NET business logic to this cloud platform won't typically pose many problems. And once again, building on an existing platform lets the ISV focus on their business logic—the thing that makes them money—rather than spending time on infrastructure.

- An enterprise creating an application for its customers might choose to build it on Windows Azure. Because Windows Azure is .NET-based, developers with the right skills aren't difficult to find, nor are they prohibitively expensive. Running the application in Microsoft's data centers frees the enterprise from the responsibility and expense of managing its own servers, turning capital expenses into operating expenses. And especially if the application has spikes in usage—maybe it's an on-line flower store that must handle the Mother's Day rush—letting Microsoft maintain the large server base required for this can make economic sense.

Running applications in the cloud is one of the most important aspects of cloud computing. With Windows Azure, Microsoft provides a platform for doing this, along with a way to store application data.

As interest in cloud computing continues to grow, expect to see more Windows applications created for this new world.

.NET SERVICES

Running applications in the cloud is an important aspect of cloud computing, but it's far from the whole story. It's also possible to provide cloud-based services that can be used by either on-premises applications or cloud applications. Filling this gap is the goal of .NET Services.

Originally known as BizTalk Services, the functions provided by .NET Services address common infrastructure challenges in creating distributed applications. Figure 3 shows its components.

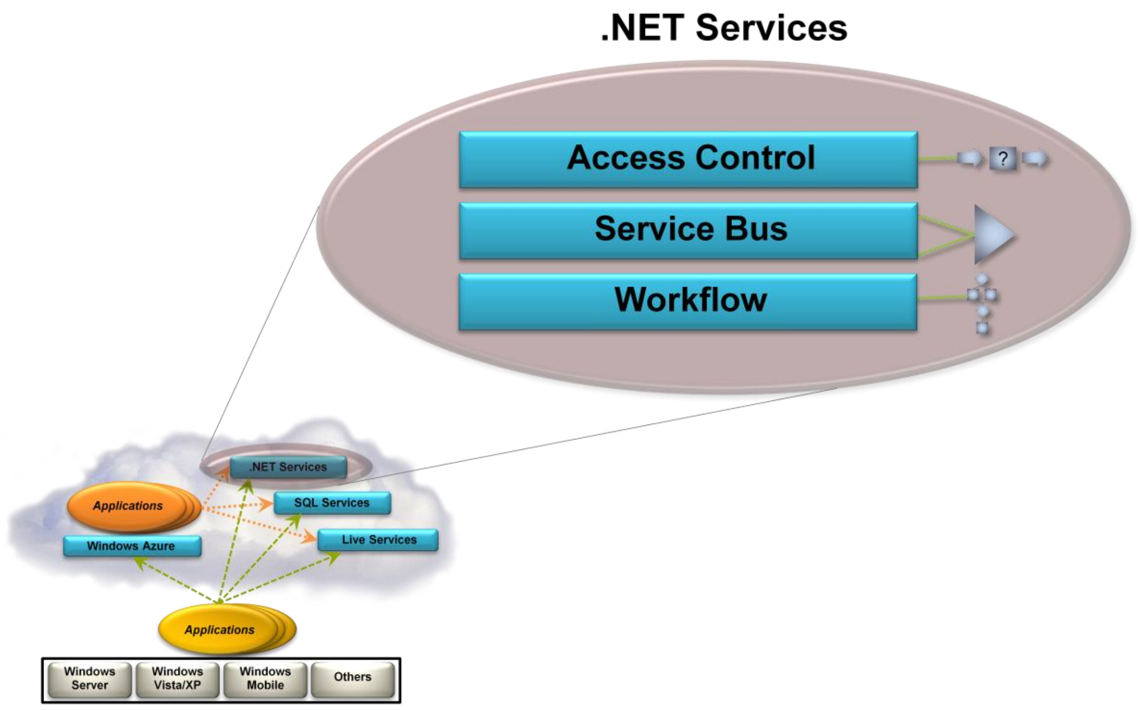

Figure 3: .NET Services provides cloud-based infrastructure that can be used by both cloud and on-premises applications.

The components of .NET Services are:

- Access Control: An increasingly common approach to identity is to have each user supply an application with a *token* containing some set of *claims*. The application can then decide what this user is allowed to do based on these claims. Doing this effectively across companies requires *identity federation*, which lets claims created in one identity scope be accepted in another. It might also require *claims transformation*, modifying claims when they're passed between identity scopes. The Access Control service provides a cloud-based implementation of both.

- Service Bus: Exposing an application's services on the Internet is harder than most people think. The goal of Service Bus is to make this simpler by letting an application expose Web services endpoints that can be accessed by other applications, whether on-premises or in the cloud. Each exposed endpoint is assigned a URI, which clients can use to locate and access the service. Service Bus also handles the challenges of dealing with network address translation and getting through firewalls without opening new ports for exposed applications.

- Workflow: Creating composite applications, as in enterprise application integration, requires logic that coordinates the interaction among the various parts. This logic is sometimes best implemented using a workflow capable of supporting long-running processes. Built on Windows Workflow Foundation (WF), the Workflow service allows running this kind of logic in the cloud.

Here are some examples of how .NET Services might be used:

- An ISV that provides an application used by customers in many different organizations might use the Access Control service to simplify the application's development and operation. For example, this .NET Services component could translate the diverse claims used in the various customer organizations, each of which might use a different identity technology internally, into a consistent set that the ISV's application could use. Doing this also allows offloading the mechanics of identity federation onto the cloud-based Access Control service, freeing the ISV from running its own on-premises federation software.

- Suppose an enterprise wished to let software at its trading partners access one of its applications. It could expose this application's functions through SOAP or RESTful Web services, then register their endpoints with Service Bus. Its trading partners could then use Service Bus to find these endpoints and access the services. Since doing this doesn't require opening new ports in the organization's firewall, it reduces the risk of exposing the application. The organization might also use the Access Control service, which is designed to work with Service Bus, to rationalize identity information sent to the application by these partners.

- Perhaps the organization in the previous example needs to make sure that a business process involving its trading partners must be executed consistently. To do this, it can use the Workflow service to implement a WF-based application that carries out this process. The application can communicate with partners using Service Bus and rely on the Access Control service to smooth out differences in identity information.

As with Windows Azure, a browser-accessible portal is provided to let customers sign up for .NET Services using a Windows Live ID. Microsoft's goal with .NET Services is clear: providing useful cloud-based infrastructure for distributed applications.

SQL SERVICES

One of the most attractive ways of using Internet-accessible servers is to handle data. This means providing a core database, certainly, but it can also include more. The goal of SQL Services is to provide a set of cloud-based services for storing and working with many kinds of data, from unstructured to relational.

Microsoft says that SQL Services will include a range of data-oriented facilities, such as reporting, data analytics, and others. The first SQL Services component to appear, however, is SQL Data Services. Figure 4 illustrates this idea.

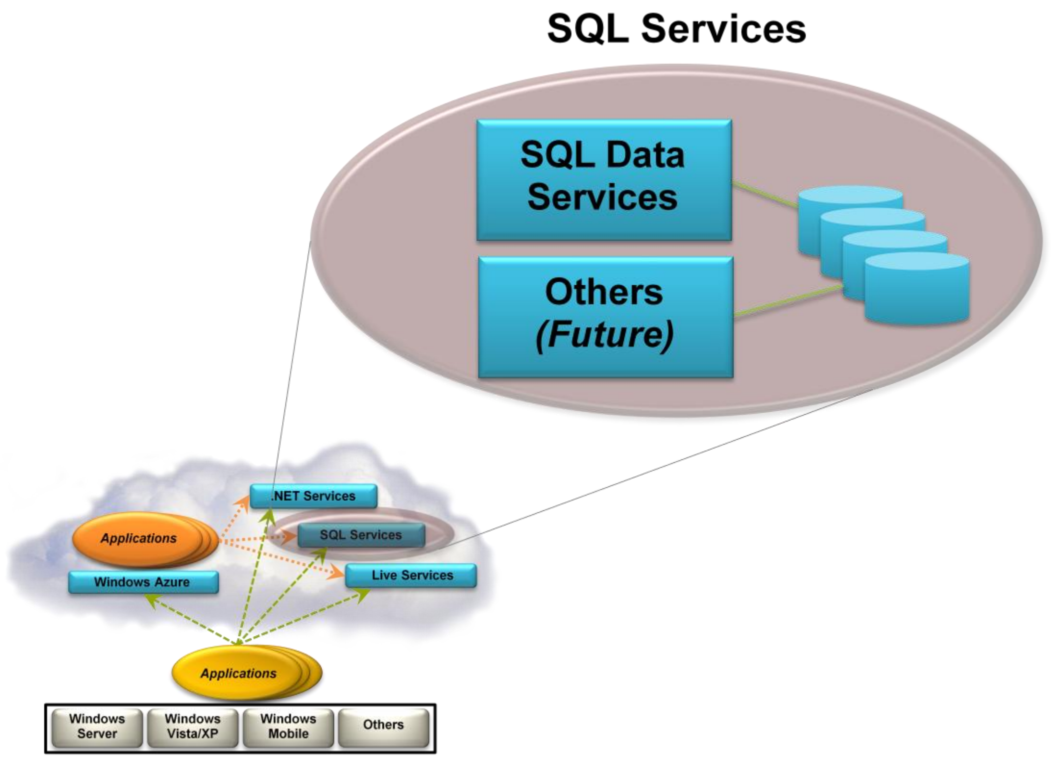

Figure 4: SQL Services provides data-oriented facilities in the cloud.

SQL Data Services, formerly known as SQL Server Data Services, provides a database in the cloud. As the figure suggests, this technology lets on-premises and cloud applications store and access data on Microsoft servers in Microsoft data centers. As with other cloud technologies, an organization pays only for what it uses, increasing and decreasing usage (and cost) as the organization's needs change. Using a cloud database also allows converting what would be capital expenses, such as investments in disks and database management systems (DBMSs), into operating expenses.

A primary goal of SQL Data Services is to be broadly accessible. Toward this end, it exposes both SOAP and RESTful interfaces, allowing its data to be accessed in various ways. And because this data is exposed through standard protocols, SQL Data Services can be used by applications on any kind of system—it's not a Windows-only technology.

Unlike the Windows Azure storage service, SQL Data Services is built on Microsoft SQL Server. Nonetheless, the service does not expose a traditional relational interface. Instead, SQL Data Services provides a hierarchical data model that doesn't require a pre-defined schema. Each data item stored in this service is kept as a property with its own name, type, and value. To query this data, applications can use direct RESTful access or a language based on the C# syntax defined by Microsoft's Language Integrated Query (LINQ).

There's an obvious question here: Why not just offer SQL Server in the cloud? Why instead provide a cloud database service that uses an approach different from what most of us already know? One answer is that providing this slightly different set of services offers some advantages. SQL Data Services can provide better scalability, availability, and reliability than is possible by just running a relational DBMS in the cloud. The way it organizes and retrieves data makes replication and load balancing easier and faster than with a traditional relational approach. Another advantage is that SQL Data Services doesn't require customers to manage their own DBMS. Rather than worry about the mechanics, such as monitoring disk usage, servicing log files, and determining how many instances are required, a SQL Data Services customer can focus on what's important: the data. And finally, Microsoft has announced plans to add more relational features to SQL Data Services. Expect its functionality to grow.

SQL Data Services can be used in a variety of ways. Here are some examples:

- An application might archive older data to SQL Data Services. For instance, think of an application that provides frequently updated RSS feeds. Information in these feeds that's more than, say, 30 days old probably isn't accessed often, but it still needs to be available. Moving this data to SQL Data Services could provide a low-cost, reliable alternative.

- Suppose a manufacturer wishes to make product information available to both its dealer network and directly to customers. Putting this data in SQL Data Services would allow it to be accessed by applications running at the dealers and by a customer-facing Web application run by the manufacturer itself. Because the data can be accessed through RESTful and SOAP interfaces, the applications that use it can be written using any technology and run on any platform.

Like other components of the Azure Services Platform, SQL Data Services makes it simple to use its services: Just go to a Web portal and provide the necessary information. Whether it's for archiving data cheaply, making data accessible to applications in diverse locations, or other reasons, a cloud database can be an attractive idea. As new technologies become available under the SQL Services umbrella, organizations will have the option to use the cloud for more and more data-oriented tasks.

LIVE SERVICES

While the idea of cloud platforms is relatively new, the Internet is not. Hundreds of millions of people around the world use it every day. To help them do this, Microsoft provides an expanding group of Internet applications, including the Windows Live family and others. These applications let people send instant messages, store their contact information, search, get directions, and do other useful things.

All of these applications store data. Some of that data, such as contacts, varies with each user. Others, like mapping and search information, doesn't—we all use the same underlying information. In either case, why not make this data available to other applications? While controls are required—freely exposing everyone's personal information isn't a good idea—letting applications use this information can make sense.

To allow this, Microsoft has wrapped this diverse set of resources into a group of Live Services. Existing Microsoft applications, such as the Windows Live family, rely on Live Services to store and manage their information. To let new applications access this information, Microsoft provides the Live Framework. Figure 5 illustrates some of the Framework's most important aspects.

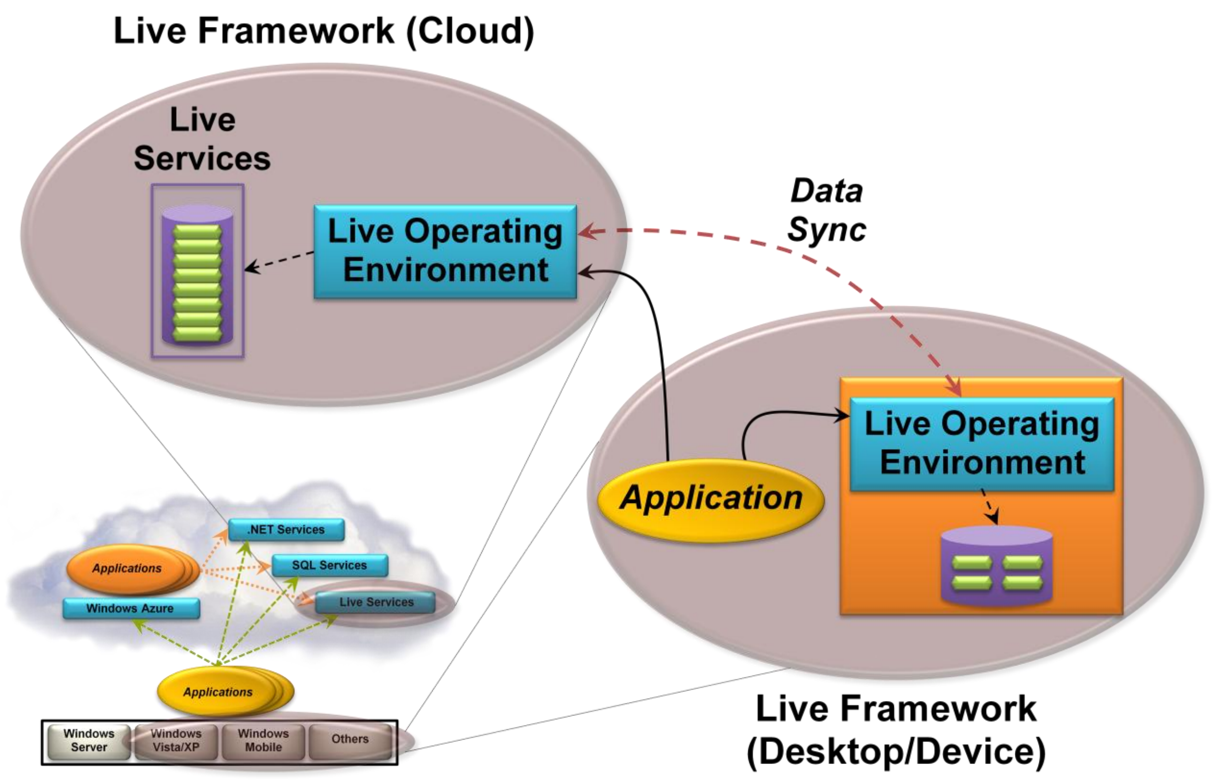

Figure 5: The Live Framework lets applications access Live Services data, optionally synchronizing that data across desktops and devices.

The fundamental component in the Live Framework is the Live Operating Environment. As the figure shows, this component runs in the cloud, and applications use it to access Live Services data. Data access through the Live Operating Environment relies on HTTP, which means that applications written using the .NET Framework, JavaScript, Java, or any other language can use Live Services data. Information in Live Services can also be accessed as an Atom or RSS feed, letting an application learn about changes to this data. And to set up and manage the Live Services her application needs, a developer can use the browser-based Live Services Developer Portal.

Figure 5 shows another aspect of the Live Framework: The Live Operating Environment can also live on desktop systems running Windows Vista, Windows XP, or Macintosh OS X, and on Windows Mobile 6 devices. To use this option, a user groups his systems into what's known as a *mesh*. For example, you might create a mesh that contains your desktop computer, your laptop, and your mobile phone. Each of these systems runs an instance of the Live Operating Environment.

A fundamental characteristic of every mesh is that the Live Operating Environment can synchronize data across all of the mesh's systems. Users and applications can indicate what types of data should be kept in sync, and the Live Operating Environment will automatically update all desktops, laptops, and devices in the mesh with changes made to that data on any of them. And since the cloud is part of every user's mesh—it acts like a special device—this includes Live Services data. For example, if a user has entries maintained in the contacts database used by Windows Live Hotmail, Windows Live Messenger, Windows Live Contacts, and other applications, this data is automatically kept in sync on every device in his mesh. (This ability isn't yet supported in the Live Framework November 2008 CTP, however.) The Live Operating

Environment also allows a user to expose data from his mesh to other users, letting him selectively share this information.

As Figure 5 shows, an application can access mesh data through either the local instance of the Live Operating Environment or the cloud instance. In both cases, access is accomplished in the same way: through HTTP requests. This symmetry lets an application work identically whether it's connected to the cloud or not—the same data is available, and it's accessed in the same way.

Any application, whether it's running on Windows or some other operating system, can access Live Services data in the cloud via the Live Operating Environment. If the application is running on a system that's part of a mesh, it also has the option of using the Live Operating Environment to access a local copy of that Live Services data, as just described. There's also a third possibility, however: A developer can create what's called a *mesh-enabled Web application*. This style of application is built using a multi-platform technology such as Microsoft Silverlight, and it accesses data through the Live Operating Environment. Because of these restrictions, a mesh-enabled application can potentially execute on any machine in a user's mesh—a Windows machine, a Macintosh, or a Windows Mobile device—and it always has access to the same (synchronized) data. To help users find these applications, the Live Framework environment provides a cloud-based application catalog for mesh-enabled Web applications. A user can browse this catalog, choose an application, then install it. And to help their creators build a business from their work, Microsoft plans to provide built-in support for displaying advertising in these applications.

The Live Framework offers a diverse set of functions that can be used in a variety of different ways. Here are a few examples:

- A Java application running on Linux could rely on the Live Framework to access a user's contacts information. The application is unaware that the technology used to expose this information is the Live Framework; all it sees is a consistent HTTP interface to the user's data.

- A .NET Framework application might require its user to create a mesh, then use the Live Framework as a data caching and synchronization service. When the machine this application runs on is connected to the Internet, the application accesses a copy of its data in the cloud. When the machine is disconnected—maybe it's running on a laptop that's being used on an airplane—the application accesses a local copy of the same data. Changes made to any copy of the data are propagated by the Live Operating Environment.

- An ISV can create a mesh-enabled Web application that lets people keep track of what their friends are doing. This application, which can run unchanged on all of its user's systems, exploits several aspects of the Live Framework that support social applications. Because the Live Framework can expose information in a user's mesh as a feed, for example, the application can track updates from any of the user's friends. Because the Live Framework provides a delivery mechanism for mesh-enabled Web apps, viral distribution is possible, with each user inviting friends to use the application. And because the mesh automatically includes a user's Live Services contacts, the user can ask the application to invite friends by name, letting the application contact them directly.

The Live Framework provides a straightforward way to access Live Services data (and don't be misled by the simple contacts example used here—there's lots more in Live Services). Its data synchronization functions can also be applied in a variety of applications. For applications that need what it provides, this platform offers a unique set of supporting functions.

A CLOSER LOOK AT THE TECHNOLOGIES

Having a broad understanding of the Azure Services Platform is an important first step. Getting a deeper understanding of each technology is also useful, however. This section takes a slightly more in-depth look at each member of the family.

WINDOWS AZURE

Windows Azure does two main things: It runs applications and it stores their data. Accordingly, this section is divided into two parts, one for each of these areas. How these two things are managed is also important, and so this description looks at this part of the story as well.

Running Applications

On Windows Azure, an application typically has multiple *instances*, each running a copy of all or part of the application's code. Each of these instances runs in its own virtual machine (VM). These VMs run 64-bit Windows Server 2008, and they're provided by a hypervisor that's specifically designed for use in the cloud.

Yet a Windows Azure application can't actually see the VM it's running in. A developer isn't allowed to supply his own VM image for Windows Azure to run, nor does he need to worry about maintaining this copy of the Windows operating system. Instead, the CTP version lets a developer create .NET 3.5 applications using *Web role* instances and/or *Worker role* instances. Figure 6 shows how this looks.

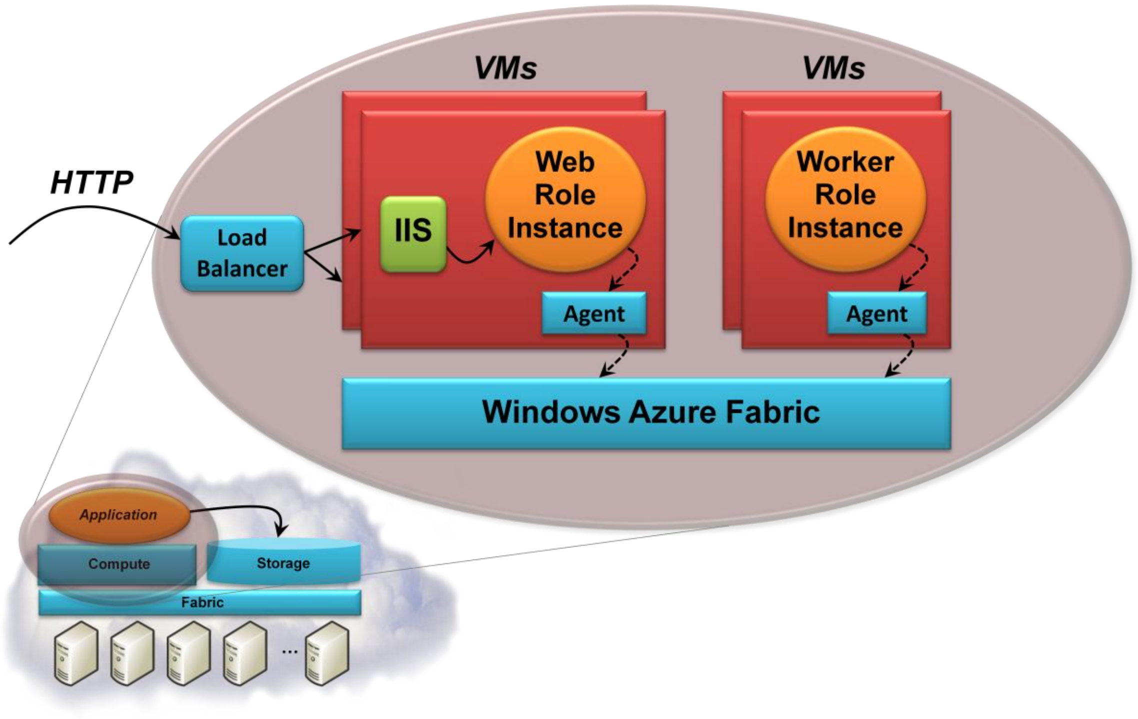

Figure 6: In the initial CTP version, Windows Azure applications can consist of Web role instances and Worker role instances, with each instance running in its own virtual machine.

As its name suggests, each Web role instance accepts incoming HTTP (or HTTPS) requests via Internet Information Services (IIS) 7. A Web role can be implemented using ASP.NET, WCF, or another .NET Framework technology that works with IIS. As Figure 6 shows, Windows Azure provides built-in load balancing to spread requests across Web role instances that are part of the same application.

A Worker role instance, by contrast, cannot accept requests directly from the outside world—it's not allowed to have any incoming network connections, and IIS isn't running in its VM. Instead, it gets its input from a Web role instance, typically via a queue in Windows Azure storage. The result of its work can be written to Windows Azure storage or sent to the outside world—outgoing network connections are allowed. Unlike a Web role instance, which is created to handle an incoming HTTP request and shut down when that request has been processed, a Worker role instance can run indefinitely—it's a batch job. Befitting this generality, a Worker role can be implemented using any .NET technology with a main() method (subject to the limits of Windows Azure trust, as described below).

Whether it runs a Web role instance or a Worker role instance, each VM also contains a Windows Azure agent that allows the application to interact with the Windows Azure fabric, as Figure 6 shows. The agent exposes a Windows Azure-defined API that lets the instance write to a Windows Azure-maintained log, send alerts to its owner via the Windows Azure fabric, and more.

While this might change over time, Windows Azure's initial release maintains a one-to-one relationship between a VM and a physical processor core. Because of this, the performance of each application can be guaranteed—each Web role instance and Worker role instance has its own dedicated processor core. To increase an application's performance, its owner can increase the number of running instances specified in the application's configuration file. The Windows Azure fabric will then spin up new VMs, assign them to cores, and start running more instances of this application. The fabric also detects when a Web role or Worker role instance has failed, then starts a new one.

Notice what this implies: To be scalable, Windows Azure Web role instances must be stateless. Any client-specific state should be written to Windows Azure storage or passed back to the client in a cookie. Web role statelessness is also all but mandated by Windows Azure's built-in load balancer. Because it doesn't allow creating an affinity with a particular Web role instance, there's no way to guarantee that multiple requests from the same user will be sent to the same instance.

Both Web roles and Worker roles are implemented using standard .NET technologies. Yet moving existing .NET Framework applications to Windows Azure unchanged usually won't work. For one thing, the way an application accesses storage is different. Access to Windows Azure storage uses ADO.NET Web Services, a relatively new technology that isn't yet ubiquitous in on-premises applications. Similarly, Worker role instances typically rely on queues in Windows Azure storage for their input, an abstraction that's not available in on-premises Windows environments. Another limitation is that Windows Azure applications don't run in a full trust environment. Instead, they're limited to what Microsoft calls *Windows Azure trust*, which is similar to the medium trust allowed by many ASP.NET hosters today.

For developers, building a Windows Azure application in the CTP version looks much like building a traditional .NET application. Microsoft provides Visual Studio 2008 project templates for creating Windows Azure Web roles, Worker roles, and combinations of the two. Developers are free to use any .NET language (although it's fair to say that Microsoft's initial focus for Windows Azure has been on C#). Also, the Windows Azure software development kit includes a version of the Windows Azure environment that runs on the developer's machine. This Windows Azure-in-a-box includes Windows Azure storage, a

Windows Azure agent, and everything else seen by an application running in the cloud. A developer can create and debug his application using this local simulacrum, then deploy it to Windows Azure in the cloud when it's ready. Still, some things really are different in the cloud. It's not possible to attach a debugger to a cloud-based application, for example, and so debugging cloud applications relies primarily on writing to a Windows Azure-maintained log via the Windows Azure agent.

Windows Azure also provides other services for developers. For example, a Windows Azure application can send an alert string through the Windows Azure agent, and Windows Azure will forward that alert via email, instant messaging, or some other mechanism to its specified recipient. If desired, the Windows Azure fabric can itself detect an application failure and send an alert. The Windows Azure platform also provides detailed information about the application's resource consumption, including processor time, incoming and outgoing bandwidth, and storage.

Accessing Data

Applications work with data in many different ways. Sometimes, all that's required are simple blobs, while other situations call for a more structured way to store information. And in some cases, all that's really needed is a way to exchange data between different parts of an application. Windows Azure storage addresses all three of these requirements, as Figure 7 shows.

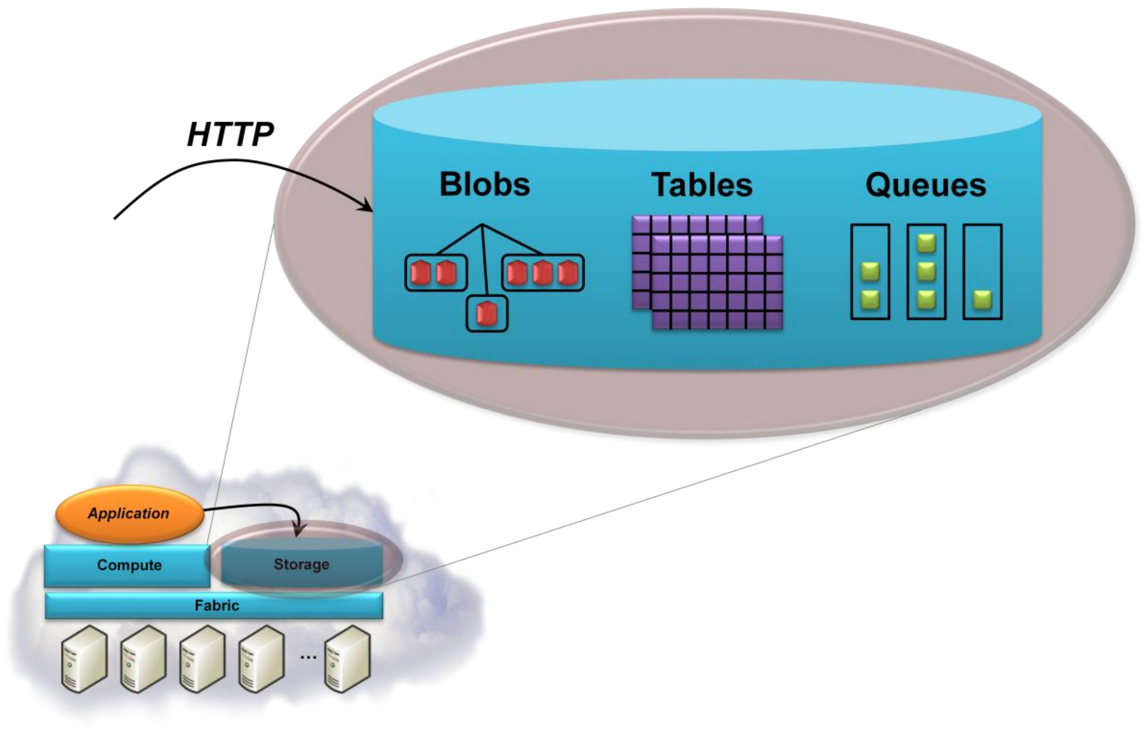

Figure 7: Windows Azure allows storing data in blobs, tables, and queues, all accessed in a RESTful style via HTTP.

The simplest way to store data in Windows Azure storage is to use blobs. As Figure 7 suggests, there's a simple hierarchy: A storage account can have one or more *containers*, each of which holds one or more blobs. Blobs can be big—up to 50 gigabytes each—and to make transferring large blobs more efficient, each one can be subdivided into blocks. If a failure occurs, retransmission can resume with the most recent block rather than sending the entire blob again. Blobs can also have associated metadata, such as information about where a JPEG photograph was taken or who the composer is for an MP3 file.

Blobs are just right for some kinds of data, but they're too unstructured for many situations. To allow applications to work with data in a more fine-grained way, Windows Azure storage provides tables. Don't be misled by the name: These aren't relational tables. In fact, even though they're called "tables", the data they contain is actually stored in a simple hierarchy of entities with properties. A table has no defined schema; instead, properties can have various types, such as int, string, Bool, or DateTime. And rather than using SQL, an application accesses a table's data using a query language with LINQ syntax. A single table can be quite large, with billions of entities holding terabytes of data, and Windows Azure storage can partition it across many servers if necessary to improve performance.

Blobs and tables are both focused on storing data. The third option in Windows Azure storage, queues, has a quite different purpose. The primary role of queues is to provide a way for Web role instances to communicate with Worker role instances. For example, a user might submit a request to perform some compute-intensive task via a Web page implemented by a Windows Azure Web role. The Web role instance that receives this request can write a message into a queue describing the work to be done. A Worker role instance that's waiting on this queue can then read the message and carry out the task it specifies. Any results can be returned via another queue or handled in some other way.

Regardless of how it's stored—in blobs, tables, or queues—all data held in Windows Azure storage is replicated three times. This replication allows fault tolerance, since losing a copy isn't fatal. The system guarantees consistency, however, so an application that reads data it has just written will get what it expects.

Windows Azure storage can be accessed either by a Windows Azure application or by an application running someplace else. In both cases, all three Windows Azure storage styles use the conventions of REST to identify and expose data. Everything is named using URIs and accessed with standard HTTP operations. A .NET client can also use ADO.NET Data Services and LINQ, but access to Windows Azure storage from, say, a Java application can just use standard REST. For example, a blob can be read with an HTTP GET against a URI formatted like this:

http://*<StorageAccount>*.blob.core.windows.net/*<Container>*/*<BlobName>*

<StorageAccount> is an identifier assigned when a new storage account is created, and it uniquely identifies the blobs, tables, and queues created using this account. *<Container>* and *<BlobName>* are just the names of the container and blob that this request is accessing.

Similarly, a query against a particular table is expressed as an HTTP GET against a URI formatted like this:

http://*<StorageAccount>.*table.core.windows.net/*<TableName>*?$filter=*<Query>*

Here, *<TableName>* specifies the table being queried, while *<Query>* contains the query to be executed against this table.

Even queues can be accessed by both Windows Azure applications and external applications by issuing an HTTP GET against a URI formatted like this:

http://*<StorageAccount>*.queue.core.windows.net/*<QueueName>*

The Windows Azure platform charges independently for compute and storage resources. This means that an on-premises application could use just Windows Azure storage, accessing its data in the RESTful way just described. Still, it's fair to say that the primary purpose of Windows Azure storage is to maintain data used by Azure applications. And because that data can be accessed directly from non-Windows Azure applications, it remains available even if the Windows Azure application that uses it isn't running.

The goal of application platforms, whether on-premises or in the cloud, is to support applications and data. Windows Azure provides a home for both of these things. Going forward, expect to see a share of what would have been on-premises Windows applications instead be built for this new cloud platform.

.NET SERVICES

Running applications in the cloud is useful, but so is providing cloud-based infrastructure services. These services can be used by either on-premises or cloud-based applications, and they can address problems that can't be solved as well in any other way. This section takes a closer look at Microsoft's offerings in this area: the .NET Access Control Service, .NET Service Bus, and the .NET Workflow Service, known collectively as .NET Services.

Access Control Service

Working with identity is a fundamental part of most distributed applications. Based on a user's identity information, an application makes decisions about what that user is allowed to do. To convey this information, applications can rely on tokens defined using the Security Assertion Markup Language (SAML). A SAML token contains claims, each of which carries some piece of information about a user. One claim might contain her name, another might indicate her role, such as manager, while a third contains her email address. Tokens are created by software known as a *security token service (STS)*, which digitally signs each one to verify its source.

Once a client (such as a Web browser) has a token for its user, it can present the token to an application. The application then uses the token's claims to decide what this user is allowed to do. There are a couple of possible problems, however:

- What if the token doesn't contain the claims this application needs? With claims-based identity, every application is free to define the set of claims that its users must present. Yet the STS that created this token might not have put into it exactly what this application requires.
- What if the application doesn't trust the STS that issued this token? An application can't accept tokens issued by just any STS. Instead, the application typically has access to a list of certificates for trusted STSs, allowing it to validate the signatures on tokens they create. Only tokens from these trusted STSs will be accepted.

Inserting another STS into the process can solve both problems. To make sure that tokens contain the right claims, this extra STS performs claims transformation. The STS can contain rules that define how input and output claims should be related, then use those rules to generate a new token containing the exact claims an application requires. To address the second problem, commonly called identity federation, requires that the application trust the new STS. It also requires establishing a trust relationship between this new STS and the one that generated the token the STS received.

Adding another STS allows claims transformation and identity federation, both useful things. But where should this STS run? It's possible to use an STS that runs inside an organization, an option that's provided by several vendors today. Yet why not run an STS in the cloud? This would make it accessible to users and applications in any organization. It also places the burden of running and managing the STS on a service provider.

This is exactly what the Access Control Service offers: It's an STS in the cloud. To see how this STS might be used, suppose an ISV provides an Internet-accessible application that can be used by people in many different organizations. While all of those organizations might be able to provide SAML tokens for their users, these tokens are unlikely to contain the exact set of claims this application needs. Figure 8 illustrates how the Access Control Service can address these challenges.

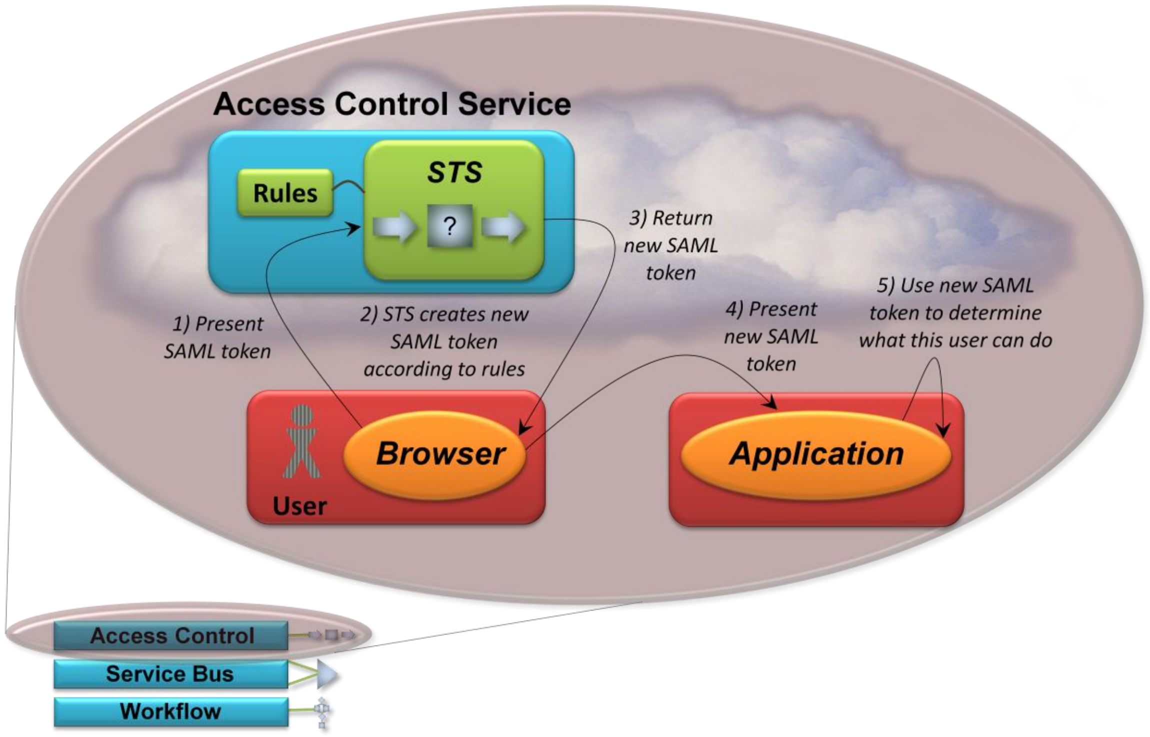

Figure 8: The Access Control Service provides rules-based claims transformation and identity federation.

First, the user's application (which in this example is a Web browser, but could also be a WCF client or something else) sends the user's SAML token to the Access Control Service (step 1). This service validates the signature on the token, verifying that it was created by an STS the service trusts. The service then creates and signs a new SAML token containing exactly the claims this application requires (step 2).

To do this, the Access Control Service's STS relies on rules defined by the owner of the application that the user is trying to access. For example, suppose the application grants specific access rights to any user who is a manager in her company. While each company might include a claim in its token indicating that a user is a manager, they'll likely all be different. One company might use the string "Manager", another the string "Supervisor", and a third an integer code. To help the application deal with this diversity, its owner could define rules in Access Control that convert all three of these claims to the common string "Decision Maker". The application's life is now made simpler, since the work of claims transformation is done for it.

Once it's been created, the STS in the Access Control Service returns this new token to the client (step 3) who then passes it on to the application (step 4). The application validates the signature on the token, making sure that it really was issued by the Access Control Service STS. Note that while the service's STS must maintain a trust relationship with the STS of each customer organization, the application itself need trust only the Access Control Service STS. Once it's certain of this token's provenance, the application can use the claims it contains to decide what this user is allowed to do (step 5).

Another way to use the Access Control Service is implicit in its name: An application can effectively offload to the service decisions about what kind of access each user is allowed. For example, suppose access to a certain function of an application requires the user to present a particular claim. The rules in the Access Control Service for the application could be defined to give this claim only to users that present other required claims, such as one of the manager claims described earlier. When the application receives a user's token, it can grant or deny access based on the presence of this claim—the decision was effectively made for it by the Access Control Service. Doing this lets an administrator define access control rules in one common place, and it can also help in sharing access control rules across multiple applications.

All communication with the Access Control Service relies on standard protocols such as WS-Trust and WS-Federation. This makes the service accessible from any kind of application on any platform. And to define rules, the service provides both a browser-based GUI and a client API for programmatic access.

Claims-based identity is on its way to becoming the standard approach for distributed environments. By providing an STS in the cloud, complete with rules-based claims transformation, the Access Control Service makes this modern approach to identity more attractive.

Service Bus

Suppose you have an application running inside your organization that you'd like to expose to software in other organizations through the Internet. At first glance, this can seem like a simple problem. Assuming your application provides its functionality as Web services (either RESTful or SOAP-based), you can just make those Web services visible to the outside world. When you actually try to do this, though, some problems appear.

First, how can applications in other organizations (or even in other parts of your own) find endpoints they can connect to for your services? It would be nice to have some kind of registry where others could locate your application. And once they've found it, how can requests from software in other organizations get through to your application? Network address translation (NAT) is very common, so an application frequently doesn't have a fixed IP address to expose externally. And even if NAT isn't being used, how can requests get through your firewall? It's possible to open firewall ports to allow access to your application, but most network administrators frown on this.

The Service Bus addresses these challenges. Figure 9 shows how.

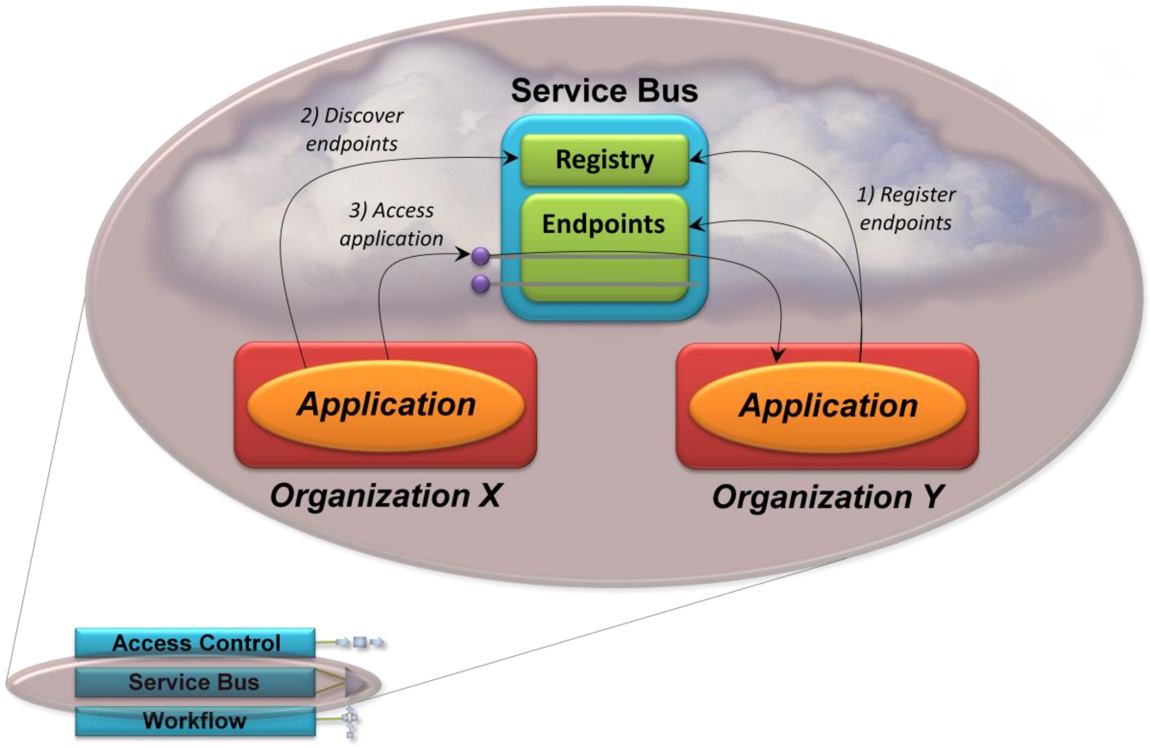

Figure 9: The Service Bus allows an application to register endpoints, then have other applications discover and use those endpoints to access its services.

To begin, your application registers one or more endpoints with the Service Bus (step 1), which exposes them on your behalf. The Service Bus assigns your organization a URI root, below which you're free to create any naming hierarchy you like. This allows your endpoints to be assigned specific, discoverable URIs. Your application must also open a connection with the Service Bus for each endpoint it exposes. The Service Bus holds this connection open, which solves two problems. First, NAT is no longer an issue, since traffic on the open connection with the Service Bus will always be routed to your application. Second, because the connection was initiated from inside the firewall, there's no problem passing information back to the application—the firewall won't block this traffic.

When an application in some other organization (or even a different part of your own) wishes to access your application, it contacts the Service Bus registry (step 2). This request uses the Atom Publishing Protocol, and it returns an AtomPub service document with references to your application's endpoints. Once it has these, it can invoke services offered through these endpoints (step 3). Each request is received by the Service Bus, then passed on to your application, with responses traveling the reverse path. And although it's not shown in the figure, the Service Bus establishes a direct connection between an application and its client whenever possible, making their communication more efficient.

Along with making communication easier, the Service Bus can also improve security. Because clients now see only an IP address provided by the Service Bus, there's no need to expose any IP addresses from within your organization. This effectively makes your application anonymous, since the outside world can't see its IP address. The Service Bus acts as an external DMZ, providing a layer of indirection to deter attackers. And finally, the Service Bus is designed to be used with the Access Control Service, allowing rules-based claims transformation. In fact, the Service Bus accepts only tokens issued by the Access Control Service STS.

An application that wishes to expose its services via the Service Bus is typically implemented using WCF. Clients can be built with WCF or other technologies, such as Java, and they can make requests via SOAP or HTTP. Applications and their clients are also free to use their own security mechanisms, such as encryption, to shield their communication from attackers and from the Service Bus itself.

Exposing applications to the outside world isn't as simple as it might seem. The intent of the Service Bus is to make implementing this useful behavior as straightforward as possible.

Workflow Service

Windows Workflow Foundation is a general technology for creating workflow-based applications. One classic scenario for workflow is controlling a long-running process, as is often done in enterprise application integration. More generally, WF-based applications can be a good choice for coordinating many kinds of work. Especially when the work being coordinated is located in different organizations, running the controlling logic in the cloud can make sense.

The Workflow Service allows this. By providing a host process for WF 3.5-based applications, it lets developers create workflows that run in the cloud. Figure 10 shows how this looks.

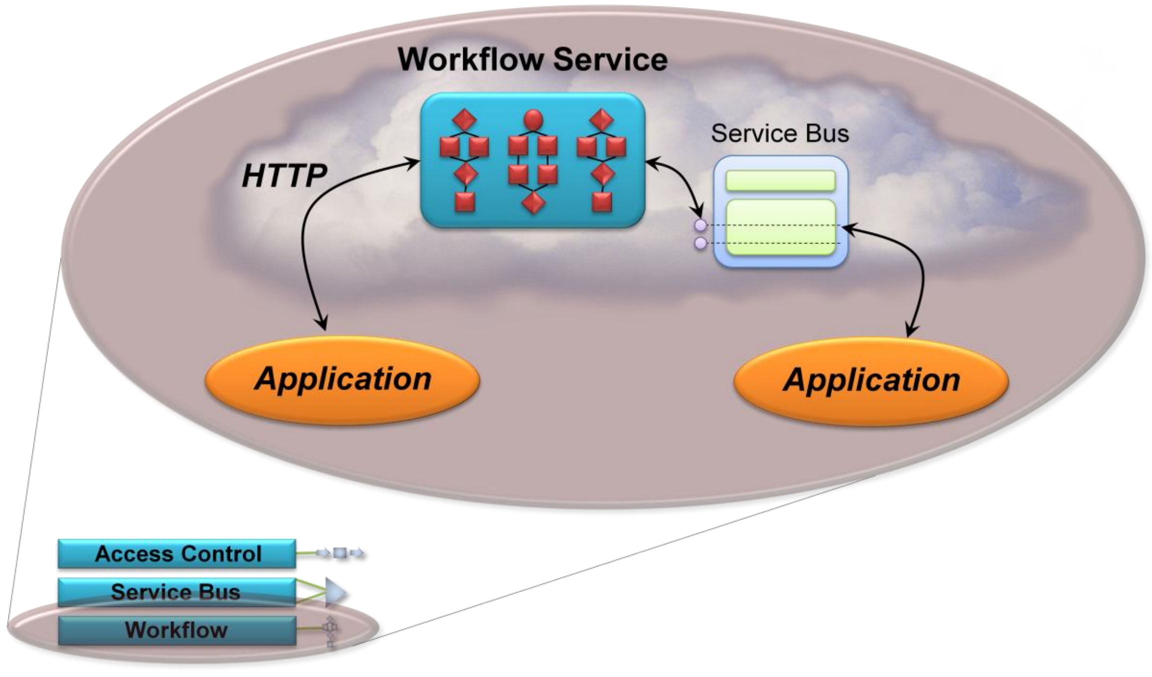

Figure 10: The Workflow Service allows creating WF-based applications that can communicate using HTTP or the Service Bus.

Every WF workflow is implemented using some number of *activities*, shown in red in the figure. Each activity performs a defined action, such as sending or receiving a message, implementing an If statement, or controlling a While loop. WF provides a standard set of activities known as the Base Activity Library

(BAL), and the Workflow Service allows the applications it runs to use a subset of the BAL. The service also provides several of its own activities. For example, the applications it hosts can communicate with other software using either HTTP or the Service Bus, as Figure 10 shows, and so the Workflow Service provides built-in activities for doing both. The Workflow Service also provides activities for working with XML messages, a common requirement for application integration.

Running in the cloud brings some limitations, however. WF-based applications running in the Workflow Service can only use WF's sequential workflow model, for example. Also, running arbitrary code isn't allowed, and so neither the BAL's Code activity nor custom activities can be used.

To create applications for the Workflow Service, developers can use Visual Studio's standard WF workflow designer. Once they're written, WF-based applications can be deployed to the cloud using a browser-based Workflow portal or programmatically using Workflow-provided APIs. Running workflows can also be managed using either the portal or these APIs. And like the Service Bus, applications that interact with the Workflow Service must first get a token from the Access Control Service—it's the only trusted STS.

WF-based applications aren't the right approach for everything. When this kind of solution is needed, however, using a workflow can make a developer's life much easier. By providing a manageable, scalable way to host WF applications in the cloud, the Workflow Service extends the reach of this useful technology.

SQL SERVICES

SQL Services is an umbrella name for what will be a group of cloud-based technologies. All of them are focused on working with data: storing it, analyzing it, creating reports from it, and more. Given that core database functions are perhaps the most fundamental of these, the first member of this family to appear is SQL Data Services.

A database in the cloud is attractive for many reasons. For some organizations, letting a specialized service provider ensure reliability, handle back-ups, and perform other management functions makes sense. Data in the cloud can also be made available to applications running anywhere, even on mobile devices. And given the economies of scale that a service provider enjoys, using a cloud database may well be cheaper than doing it yourself. The goal of SQL Data Services is to provide all of these benefits.

Yet implementing a reliable, high-performance database with Internet scalability isn't simple; tradeoffs are required. As described earlier, for example, SQL Data Services doesn't provide a standard relational database, nor does it support SQL queries. Instead, data is organized using the structure shown in Figure 11.

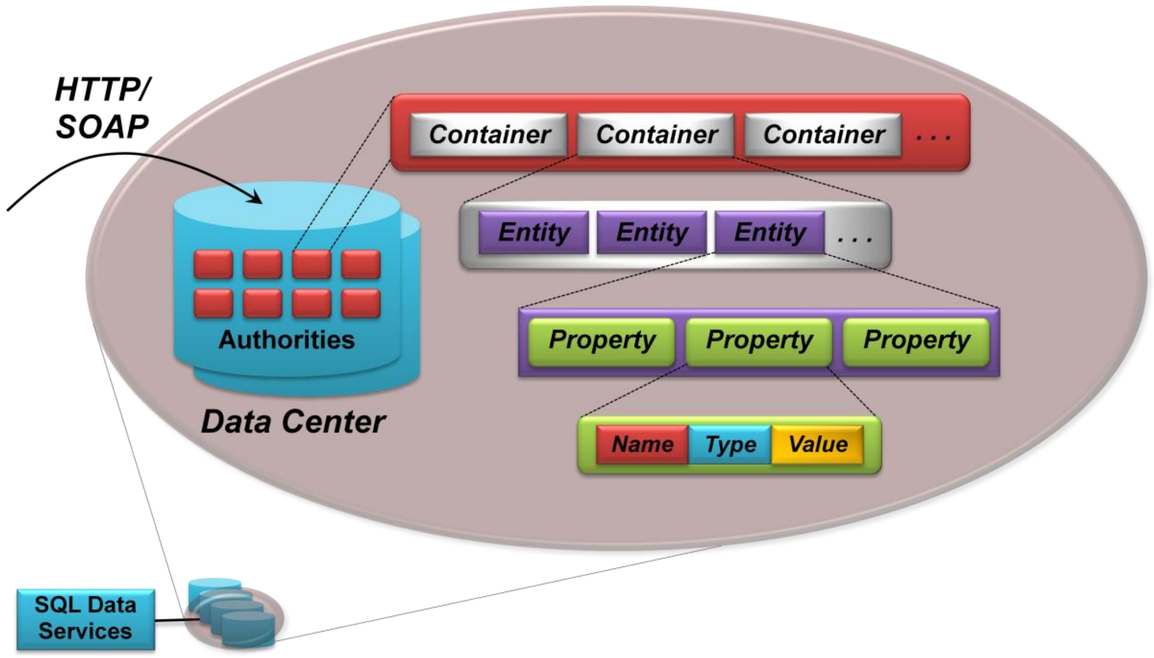

Figure 11: A SQL Data Services data center is divided into authorities, each of which holds containers, which in turn hold entities containing properties.

The information in SQL Data Services is stored in multiple data centers. Each data center contains some number of *authorities*, as Figure 11 shows. An authority is the unit of geo-location—it's stored in a specific Microsoft data center—and it has a unique DNS name. An authority holds *containers*, each of which is replicated within its data center. Containers are used for load balancing and availability: If a failure occurs, SQL Data Services will automatically begin using another replica of the container. Every query is issued against a specific container—authority-wide queries aren't allowed. Each container holds some number of *entities*, which in turn contain *properties*. Each property has a name, a type, and a value of that type. The types SQL Data Services supports include String, DateTime, Base64Binary, Boolean, and Decimal. Applications can also store blobs with MIME types.

To query this data, applications have a few options. One is to use a language that's very similar to the LINQ C# syntax, with queries sent via either SOAP or a RESTful approach. The other is to use ADO.NET Data Services, an alternative RESTful way to access data. In either case, applications issue queries against containers using operators such as ==, !=, <, >, AND, OR, and NOT. Queries can also include some SQL-like operations, such as ORDER BY and JOIN.

However queries are issued, entities, not properties, are the unit of retrieval and update. A query returns some number of entities, for example, including all of the properties they contain. Similarly, it's not possible to update just one property in an entity—the entire entity must be replaced. And because entities don't have pre-defined schemas, the properties in a single entity can have different types. The entities in a container can also be different from one another, each holding different kinds of properties.

Data in SQL Data Services is named with URIs, much like the Windows Azure storage service. The general format for a URI that identifies a particular entity looks like this:

http://*<Authority>*.data.database.windows.net/v1/*<Container>*/*<Entity>*

It's worth reiterating that nothing about SQL Data Services requires a .NET-based client running on Windows. Instead, the data it contains can be accessed via REST—that is, standard HTTP—or SOAP from any application running on any platform. Whatever platform it's running on, an application that accesses data must identify its user with a SQL Data Services-defined username and password or with a token created by the Access Control Service STS.

Going forward, Microsoft has announced plans to evolve SQL Data Services into a more relational technology. Recall that unlike Windows Azure storage, SQL Data Services is built on SQL Server, which makes this evolution more natural. Yet whatever model it provides, the technology's goal remains the same: providing a scalable, reliable, and low-cost cloud database for all kinds of applications. As SQL Services expands to include more cloud-based data services, expect to see those services rely on this first member of the family.

LIVE SERVICES

What drives the creation of new application platforms? The answer is change: changes in hardware, changes in software, and changes in how we use applications and data. Mobile phones have morphed into computers, for example, and servers in the cloud have become big parts of our lives. Applications have become more personal, as has the data we store in those applications. Combine these changes, and the stage is set for a new kind of application platform.

Live Services and the Live Framework exemplify this. Applications can use the Live Framework to access Live Services data, and they can also rely on the Live Framework to synchronize this data across desktops, laptops, and devices. Figure 12 shows how Live Services and the Live Framework fit together.

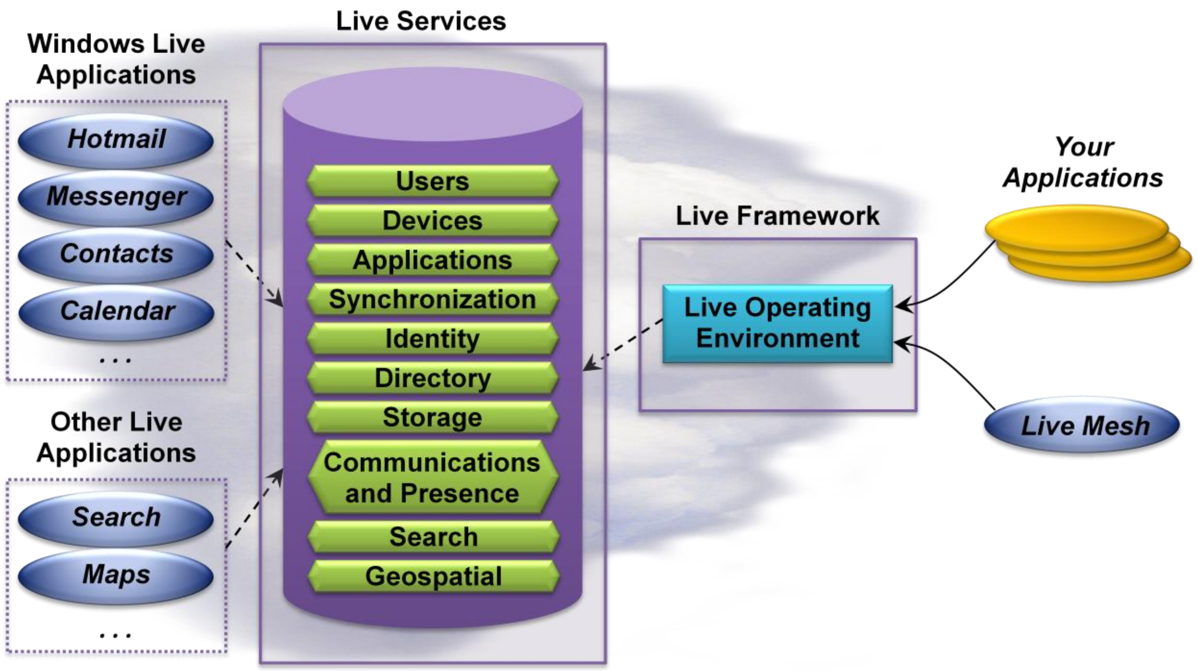

Figure 12: The Live Framework lets applications access Live Services data and more.

Live Services is broken down into several different categories, as the figure shows. Each service allows access to a particular set of resources, which can be user-specific or shared. For example, a user's contacts list is a resource provided by the Directory service, while his profile is a resource provided by the Storage service. Both of these are user-specific services, since they expose data that's associated with a particular user. The Geospatial service provides resources that contain shared data, however—maps and other geographic information—as does the Search service.

The data in Live Services is used by existing Microsoft applications in various ways, as the figure indicates. A primary goal of the Live Framework is to make it easier to create new applications that use this data. Microsoft's Live Mesh is one example of this, and ISVs and end users are free to build others. All of these applications access data through the Live Framework's primary component: the Live Operating Environment. How this looks is described next.

Accessing Data

The simplest way to access Live Services data is directly through the Live Operating Environment. Figure 13 shows how this looks.

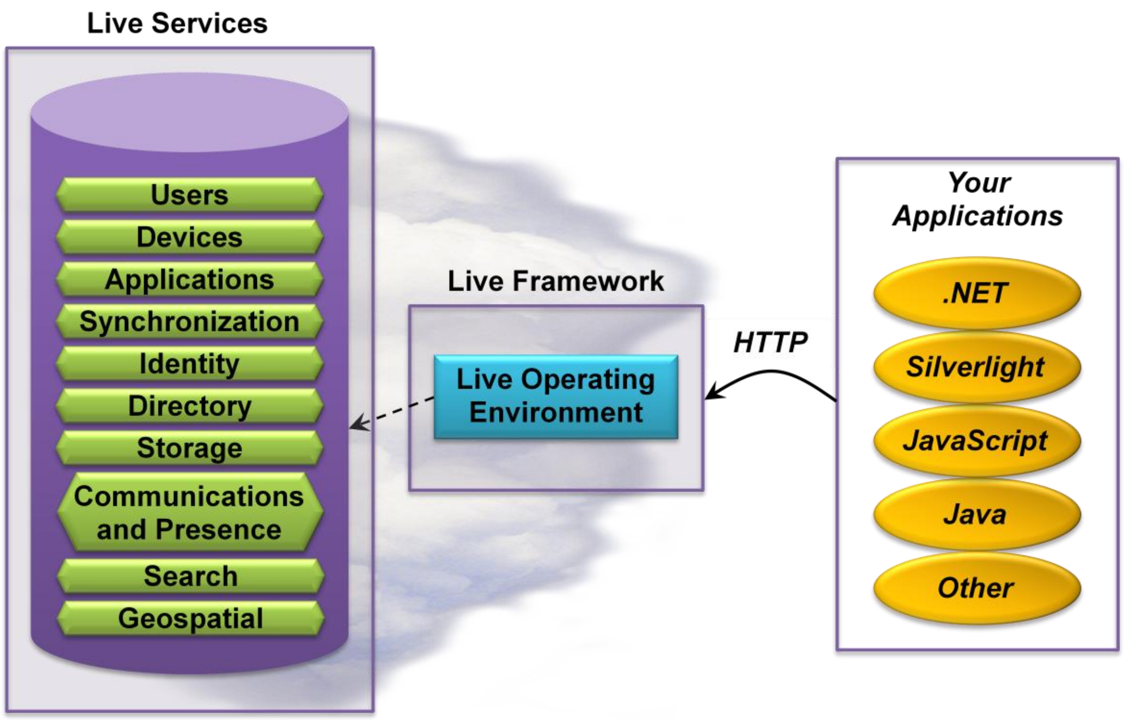

Figure 13: Because the Live Framework exposes Live Services data via HTTP, applications written using many technologies can access it.

All of the resources provided by Live Services—both user-centric and shared—are named with URIs. To access this information, an application can make RESTful requests using HTTP. Resources can also be accessed via AtomPub or in other HTTP-based ways. However it's done, information can be transferred using XML or JSON, with syndication data conveyed using RSS or Atom.

To allow a consistent approach to describing and naming Live Services data, the Live Framework defines a resource model. This model specifies types and the allowed relationships among instances of those types, along with a consistent URI naming scheme. Applications can also create custom types to store their own kinds of information. The intent is to provide enough commonality to let applications discover and navigate Live Services data while also giving application developers the flexibility they need to store diverse information. And because each user has detailed control over exactly which of her resources are exposed to which applications and for how long, no one's personal data is freely available.

It's worth pointing out that the data used by Microsoft's Live applications is exposed today through existing Live Services APIs (sometimes called the Windows Live Platform). These APIs vary significantly across applications, however. By providing common, HTTP-based access to all of this information, the Live Framework will replace this older approach with a simpler, more consistent interface.

To create an application that accesses Live Services data through the Live Framework, a developer is free to write code using a raw HTTP interface. To make this easier, however, the Live Framework also includes *Live Framework Toolkits*. These libraries provide a simpler, more natural approach for developers to build applications that access Live Services via the Live Framework. Microsoft provides toolkits for the .NET Framework, Silverlight, and JavaScript, and others are likely to emerge from the programming community. Once again, nothing about the way data is exposed by the Live Framework ties it to Microsoft technologies—Live Framework Toolkits can be created for any language or platform.

Using a Mesh

As long as it has the right permissions, any application is free to access Live Services data through the Live Framework. Optionally, though, an application might be running on a system that's been made part of a mesh. If it is, the application has a few more options.

As described earlier, each user can have her own mesh containing the systems that she uses. For instance, maybe she has a Windows XP desktop at work, a Macintosh at home, a laptop running Windows Vista, and a phone that runs Windows Mobile. All of these systems can be grouped into a mesh, as Figure 14 shows.

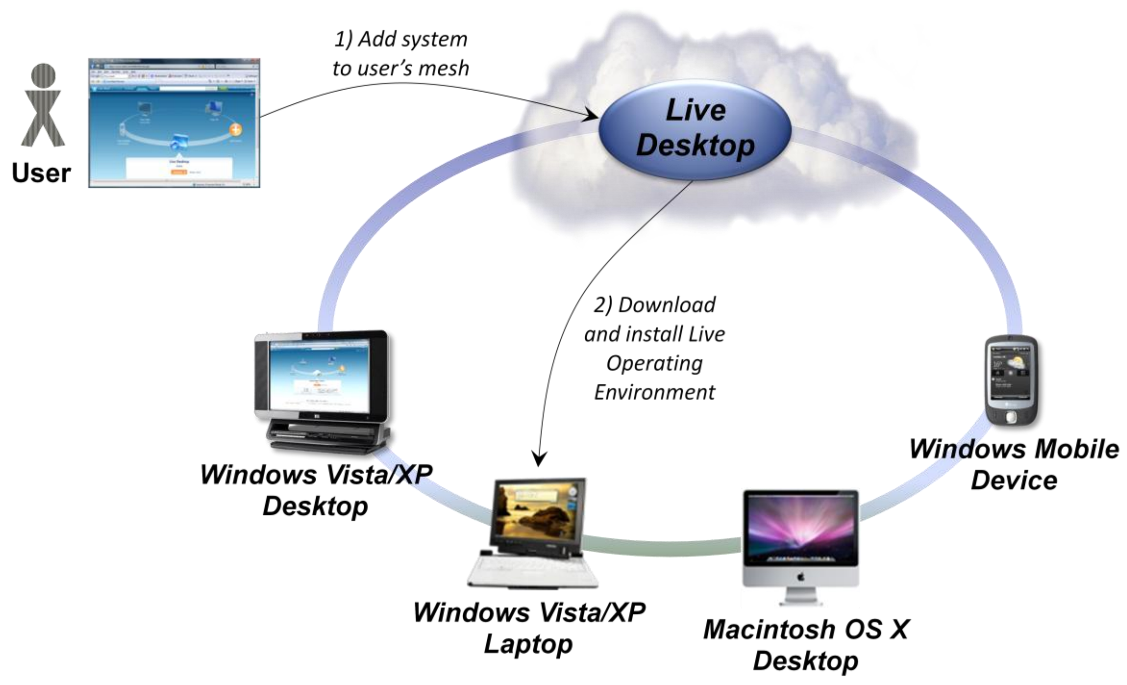

Figure 14: Adding a system to a mesh installs the Live Operating Environment on that system.

To create a mesh, a user can sign in using her Live ID, then access her own Live Desktop through her browser. She uses this cloud-based application to add systems to her mesh. As Figure 14 illustrates, the user specifies a system to add, which in this example is her laptop (step 1), and the Live Desktop adds it to her mesh. To do this, the Live Desktop in the cloud downloads and installs a copy of the Live Operating Environment onto this machine (step 2).

As described earlier, the Live Operating Environment lets applications access Live Services data via HTTP. When it's used in a mesh, however, this component also does more: It synchronizes a user's Live Services data across the cloud and all systems in the mesh. Figure 15 illustrates this idea.

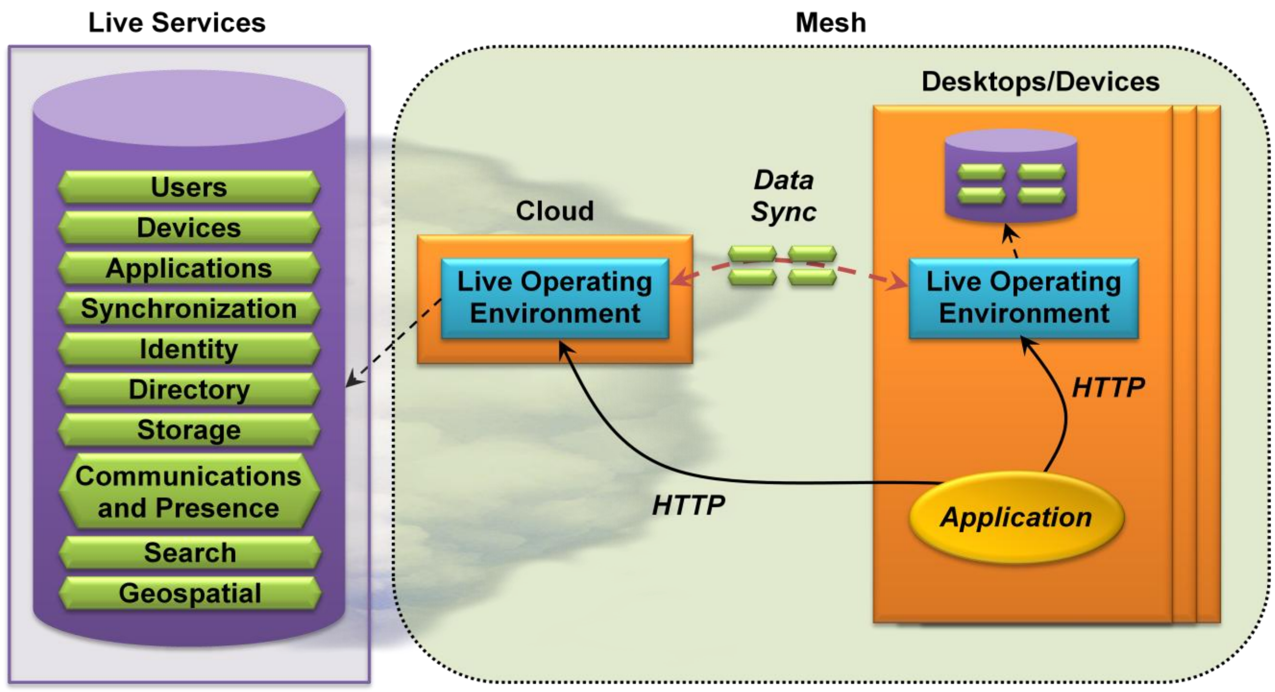

Figure 15: The Live Operating Environment keeps Live Services data synchronized across desktops, devices, and the cloud.

Users and applications can indicate what data should be included in the mesh, and the Live Operating Environment takes care of keeping that information synchronized. For example, Microsoft's Live Mesh application lets a user designate specific folders that should be part of the mesh. Once this is done, the Live Operating Environment will silently propagate changes made to data in any of these folders across all systems in the mesh. Similarly, per-user Live Services data, such as contacts and profile information, can be kept in sync across the entire mesh.

Mesh synchronization is multi-master, which means that a user can change any copy of the information on any device—there's not just one master that must be updated. The technology used to do this is FeedSync, a Microsoft-defined, publicly available protocol that relies on HTTP. Whenever possible, data is synchronized between directly connected systems—it's peer-to-peer. This isn't always an option, however, so a system can also sync with the Live Operating Environment in the cloud. This cloud-based instance can connect directly to any system in the mesh—it's everybody's peer—and so it's able to synchronize with any of them.

As always, an application running on a mesh-enabled system can access data by making HTTP requests to the Live Operating Environment in the cloud. It also has access to a local copy of all Live Services data that has been made part of this mesh, however. Rather than interacting with the remote instance of the Live Operating Environment, the application can also issue the same HTTP requests to the instance running locally, as Figure 15 shows. Except for the base URI, those requests are identical for both the local and cloud Live Operating Environment.

This symmetry lets an application work in the same way with local data and with data stored in the cloud. If an application is running on a desktop or device that's currently disconnected, for example, it can access the local copy, which acts as a cache for the last known state of the cloud data. When the device is connected again, the application can either access the cloud data directly—all that's required is changing a URI—or wait for the local copy of the data to be updated by Live Operating Environment synchronization.

Systems that don't run the Live Operating Environment can also participate in a mesh, albeit in a more limited way. Because the Live Desktop can be accessed using any browser, a user running on, say, a Linux system can use it to create a mesh with only a cloud component—the mesh contains no desktops or devices. Applications running on the Linux system can store and access data in this simple mesh just as they do any other Live Services data: using HTTP. In fact, an application can even implement the FeedSync protocol to synchronize this cloud data with a local copy. While systems that run the Live Operating Environment have more capabilities, those that don't might also find this aspect of the Live Framework useful.

Mesh-Enabled Web Applications

Any application, Windows-based or otherwise, can access Live Services data—it needn't be part of a mesh. If a developer is building an application that will always run on mesh systems, however, there's another option. He can create a mesh-enabled Web application that can be distributed and managed by the Live Framework itself. Figure 16 shows the basics of how this works.

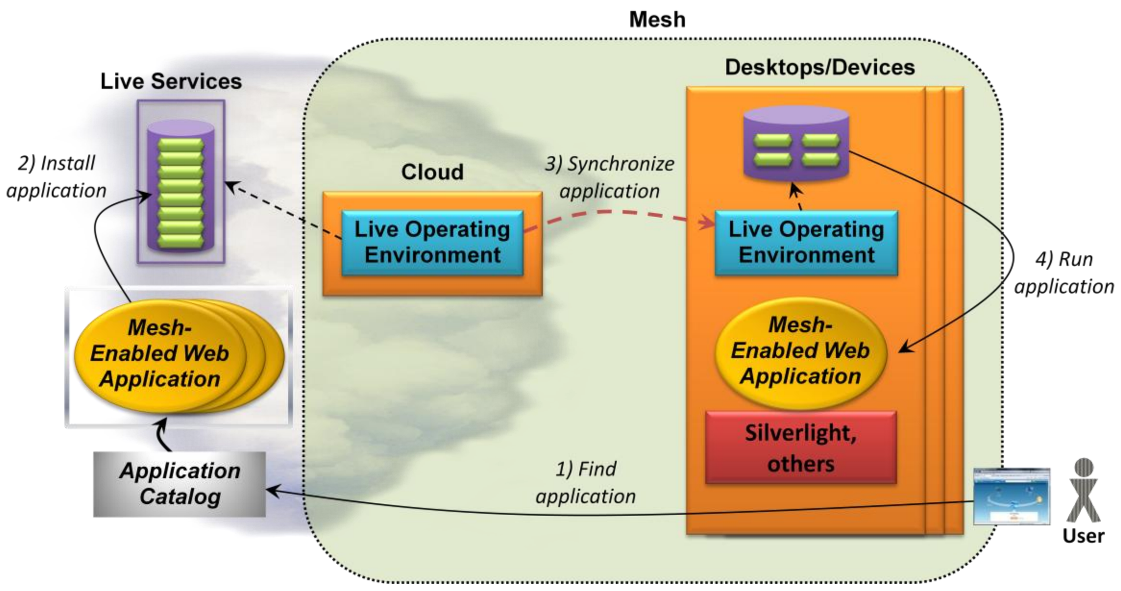

Figure 16: A user can discover a mesh-enabled Web application, then install it on his mesh.

As the figure shows, a mesh-enabled Web application can be made available through a Microsoft-provided application catalog in the cloud. A user can access this catalog to discover available mesh-enabled Web applications (step 1). Once he's chosen an application, the user can install it (step 2). Initially, this only copies the application into the user's cloud storage in Live Services. The application will be synchronized with the user's desktops and devices, though, like other mesh data (step 3). The application can now be run from the local store on any system in this user's mesh (step 4). In fact, it's not accurate to think of a mesh-enabled Web application as being installed on just one system. Instead, the application is installed on all of them—it's installed on the mesh.

A mesh-enabled Web application must be implemented using a multi-platform technology, such as Microsoft Silverlight, DHTML, or Adobe Flash. These technologies are supported on all of the operating systems that can run the Live Framework: Windows Vista/XP, Macintosh OS X, and Windows Mobile 6. Accordingly, any mesh-enabled Web application can run on any system in the mesh (although all of these options aren't supported in the Live Framework November CTP).

Because the Live Operating Environment keeps all mesh data in sync, a mesh-enabled Web application will see the same data no matter where it's running. This gives an interesting new meaning to the notion of write once, run anywhere: A mesh-enabled Web application can run unchanged on any system within a mesh, and it can also count on having the same data available no matter where it's running.

As with other kinds of Live Framework data access, a mesh-enabled Web application has access only to data that a user has specifically authorized it to work with. And like other Silverlight, DHTML, and Flash applications, a mesh-enabled Web application runs in a secure sandbox. Unless specifically allowed by a user, these applications can't directly access the local disk or the data of other mesh-enabled Web applications. A user is free to share a mesh-enabled Web application with another user's mesh, however. For example, a user could tell a mesh-enabled Web application to invite everyone in her address book to use it. Since her contacts information is directly available to the application—it's part of her mesh—having the application do this is straightforward.

To help developers create mesh-enabled Web applications, Microsoft provides project templates for Visual Studio 2008. To make updating these applications easier, a developer can upload a new version to the application catalog, then let the Live Framework automatically take care of updating that application in the mesh of every user who's installed it. And to help developers make money from their applications, Microsoft plans to allow plugging in its own adCenter or a third-party service to let a mesh-enabled Web application show ads.

To a great degree, the Live Framework is a wholly new kind of application platform. Many aspects of the environment, such as access to Live Services data and the focus on desktops and devices, make clear that a core goal for this technology is to support consumer-oriented, socially aware applications. In a very real sense, the Live Framework sits at the intersection of new technology and new kinds of human interaction.

CONCLUSIONS

The truth is evident: Cloud computing is here. For developers, taking advantage of the cloud means using cloud platforms in some way. With the Azure Services Platform, Microsoft presents a range of platform styles addressing a variety of needs:

- Windows Azure provides a Windows-based computing and storage environment in the cloud.
- .NET Services offers cloud-based infrastructure for cloud and on-premises applications.
- SQL Services provides a cloud database today through SQL Data Services, with more cloud-based data services planned.
- Live Services provides the Live Framework, which lets application access Live Services data, synchronize data across systems joined into a mesh, and more.

These four approaches address a variety of requirements, and most developers probably won’t use all of them. Yet whether you work for an ISV or an enterprise, some cloud platform services are likely to be useful for applications your organization creates. A new world is unfolding; prepare to be part of it.

ABOUT THE AUTHOR

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technology.