Microsoft® SQL Services

Microsoft

SQL Data Services – Developer Focus

Writer: Jason Lee (Content Master)

Technical Reviewer: Mohan Vanmane (Microsoft)

Published: October 2008

Applies to: SQL Server 2008

Summary: Microsoft® SQL Data Services (SDS) offers highly scalable, Internet-facing, enterprise-class database and advanced query processing for customers who want to build new applications or extend existing investments into the cloud. This paper explains the key features and architecture of SDS, and describes how you can start programming with SDS in your own applications.

Copyright

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Contents

Introduction

In most organizations, Web applications rely on an accompanying database infrastructure to store data and service queries. This infrastructure requires constant management by IT Professionals. Before you deploy an application, you must consider whether you have sufficient capacity for your data, whether the infrastructure has enough redundancy to provide the level of availability you need, and how variable usage and load might affect performance.

The onsite, server-side data platform also has a number of limitations for evolving development patterns. For example, Web applications that rely on Microsoft Silverlight™ or Flash to drive presentation and interactivity don't work well with the traditional post-back model for data population, and are better suited to a service-oriented architecture. Similarly, mashup applications must be able to consume data as a service, rather than through a tightly-coupled connection to a back-end database server.

Cloud-based data platforms such as SDS have evolved to provide answers to these constraints and limitations, thereby giving you more time and freedom to innovate with data-driven solutions. Over the course of this paper we'll take a closer look at how SDS works and how you can use it in your own applications.

What Is SDS?

SDS is an Internet-facing database that provides a robust, secure, and flexible alternative to traditional onsite database infrastructures. SDS encapsulates a set of services you can use to store, retrieve, and manipulate any amount of data, from a few kilobytes to several terabytes.

These services are exposed through standards-based interfaces such as Simple Object Access Protocol (SOAP) and Representation State Transfer (REST), making them accessible from virtually any programming language and development environment. SDS also provides a flexible entity-based data model, so you can map any data source to the cloud regardless of schematization or structure.

Why Use SDS?

SDS is a truly enterprise-class database service that you can use to extend your on-premise applications or to develop new and innovative cloud-based solutions. Let's start by looking at some of the features that SDS can offer you as a developer. There are three key areas in which SDS provides advantages:

Flexibility and scale

When you use SDS, your applications can scale to any size without hitting infrastructure limits. Support for data partitioning means you can perform parallel operations against SDS and achieve extremely high throughput rates. You can also access your data at any time and from any device that has an Internet connection and support for HTTP. From a business perspective, the "pay as you grow" service model helps to keep your start-up costs low and ensures that you only pay for the storage you use, which results in a lower total cost of ownership (TCO).

Business-ready reliability and security

SDS is built on tried and tested Microsoft Windows Server® and SQL Server® technologies, which together with published service level agreements (SLAs) can help ensure enterprise-class performance and reliability. SDS stores and manages multiple, geo-redundant copies of your data to protect against loss, and ensures that transactional consistency is maintained across these copies. You can also define the geographic locations in which your data is stored to help reduce latency and increase response times.

The confidentiality and privacy of your data is of paramount importance. SDS provides secure login access with support for Secure Sockets Layer (SSL), and you can implement further authentication and authorization at each level of the data model.

Developer agility

SDS currently exposes Web services over the SOAP and REST protocols, with support for other protocols such as JavaScript Object Notation (JSON) and Atom Publishing Protocol (AtomPub) expected soon. As such, you can program for SDS from virtually any language or environment – all you need is access to either a SOAP stack or an HTTP stack. You can query your data by using a simple text-based query syntax, based on the C# Language Integrated Query (LINQ) pattern.

The flexible data model does not require schemas, which means you can rapidly provision your data without first creating tables, columns, and relationships. SDS supports familiar **String**, **Decimal**, **Boolean**, **DateTime**, and **Binary** property data types, and you can also store virtually any type of content as a binary large object (BLOB).

Getting Started

Registering for the Service

You can try out the private beta version of SDS for free. To register, visit the SDS home page (http://go.microsoft.com/?linkid=9373222) and follow the instructions there.

Before you start programming with SDS, it's important to get an understanding of the basic architecture. The SDS data model consists of just three hierarchical levels of containment: *authorities*, *containers*, and *entities*, known collectively as the ACE model.

The ACE Model

In SDS, an authority is the top-level object. An authority can contain any number of containers, and a container can contain any number of entities.

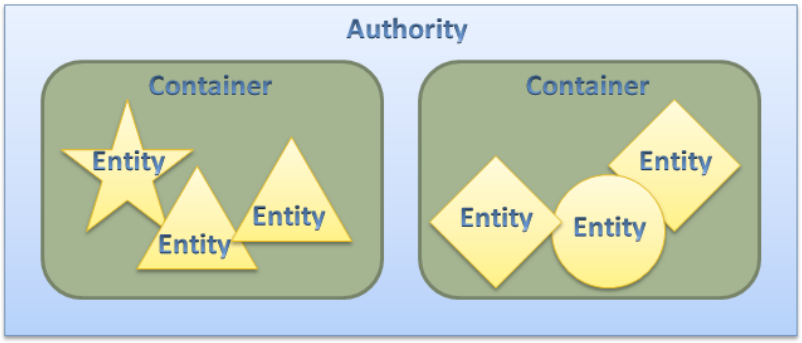

Figure 1: The ACE model

The following table shows the definition and purpose of authorities, containers, and entities.

Containment Layer	Definition	Purpose	SQL Server Analogy
Authority	Set of containers	Groups containers for accounting, security, and co-location	A SQL Server instance
Container	Set of entities	Groups entities for content and queries	An individual database (heterogeneous storage) or a database table (homogeneous storage)
Entity	Scalar property bag	A unit of storage	An individual record

Table 1: Definition and purpose of authorities, containers, and entities

In fact, authorities and containers are also types of entity. You'll see why as we move through the paper – you can use the same programming patterns to interact with SDS at any level of the data model.

A traditional relational database table has a schema that describes the row structure. In contrast, containers in SDS are not schematized. There are no columns, and you can add different types of entity, each with a different set of properties, to the same container. We'll discuss authorities, containers, and entities in more detail as we move through the paper.

The Programming Model

As mentioned earlier, you can use either SOAP or REST to perform CRUD (Create, Retrieve, Update, and Delete) operations against your data in SDS. The programming model is broadly similar in each case:

1. Set the scope of your operation (service, authority, container, or entity).
2. Perform the operation (create, retrieve, update, or delete).

If you use the SOAP protocol, you create a Scope object and then set properties on this object to target specific authorities, containers, or entities. For example, you can use the following C# code to create an authority:

```
using (SitkaSoapServiceClient proxy = new SitkaSoapServiceClient())
```

```
{
        proxy.ClientCredentials.UserName.UserName = "myUserName";
        proxy.ClientCredentials.UserName.Password = "myPassword";

        //Use the empty scope to target the service level
        Scope myScope = new Scope();

        //Create the authority
        Authority myAuthority = new Authority();
        myAuthority.Id = "exampleID";

        //Pass the scope and authority objects to the Create method
        proxy.Create(myScope, myAuthority);
}
```

Note: *The next release of SDS will include multiple SOAP endpoints for different authentication methods. In this case, you will need to modify the using statement to specify which endpoint the proxy object should use.*

REST, on the other hand, is a very simple, HTTP-based protocol that relies on URIs and basic HTTP verbs to set scopes and define operations. The scope of an operation is determined by the URI to which you address the HTTP request. Every time you create an authority, SDS creates an associated DNS record. For example, suppose you create an authority named **miami**. To scope an operation to the **miami** authority, you would address an HTTP request to the following URI:

```
https://miami.data.beta.mssds.com/v1/
```

You can build on this root URI to scope operations to containers or entities within your authority, as in the following examples:

```
https://miami.data.beta.mssds.com/v1/<container-id>
```

```
https://miami.data.beta.mssds.com/v1/<container-id>/<entity-id>
```

Having established the scope of your request, you simply call the HTTP method that corresponds to the database operation you want to perform. The following table illustrates how the principal HTTP verbs map to the core CRUD operations in SDS:

HTTP Verb	SDS Operation
POST	Create
GET	Fetch, Query
PUT	Update
DELETE	Delete

Table 2: Mappings between HTTP verbs and SDS operations

In the remainder of this paper, we'll focus on how to use SDS with the REST protocol, because REST offers broader platform support. However, the key concepts are the same regardless of your choice of protocol.

Note: *For the public beta version, the service URI is expected to change from* ***https://data.beta.mssds.com*** *to* ***https://data.database.windows.net****. Furthermore, the service will only be available over https.*

Provisioning and Adding Your Data

After you've registered an account for SDS, your first task will be to provision your data source and start adding data. Suppose that you are hired to create a data hub for Coho Winery, a prestigious wine merchant. The hub will include listings for red wines, white wines, and Champagnes, and will also include video clips of taste tests and reviews. The following diagram illustrates how you might map this to the ACE model.

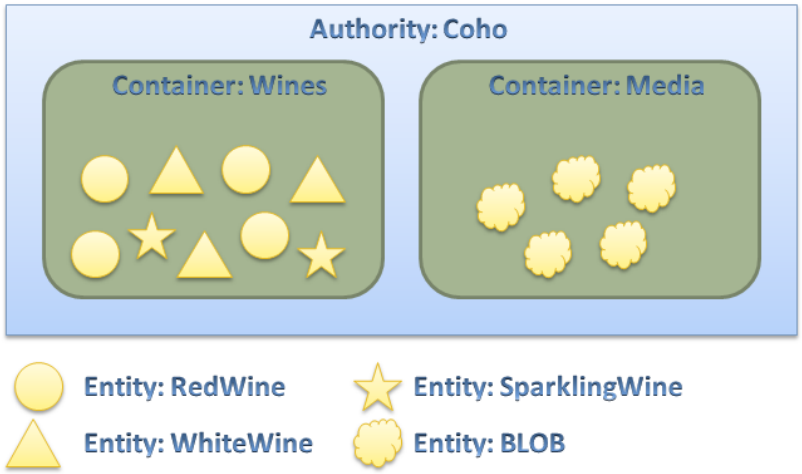

Figure 2: Example data structure for Coho Winery

As you can see, we will create a single authority with an ID of **Coho**. We will add a container named **Wines**, which we will use for heterogeneous storage of **RedWine**, **WhiteWine**, and **SparklingWine** entities. We will add a second container named **Media** to store taste tests and reviews as BLOBs.

Now that you've established your data structure, your first development task will be to create the authority.

Creating Authorities

As discussed earlier, an authority is the top layer in the SDS data model. Every authority has a DNS name assigned to it, which effectively makes the authority a unit of geographical location. Because of this, every authority ID must be globally unique.

To create an authority using the REST interface, you first create an XML payload that describes your authority:

```
<s:Authority xmlns:s='http://schemas.microsoft.com/sitka/2008/03/'>
   <s:Id>coho</s:Id>
</s:Authority>
```

The **Id** property is compulsory, and can only contain lowercase letters, numbers, or dashes. In case you're wondering, "Sitka" is the pre-release codename for SDS.

Remember that to create an authority, you must scope your operation at the service level. As such, your next step is to send an HTTP request to the service:

```
https://data.beta.mssds.com/v1/
```

Because you are performing a create operation, you use the HTTP POST method to stream your XML payload to the service as a byte array. Set the **Content-Type** header to **application/x-SDS+xml**, and set the **Content-Length** header to the length of your XML payload in bytes. The following C# code sample illustrates this process. This assumes that you have defined a string named **myPayload** that contains your XML payload, and a string named **serviceUri** that contains the URI of the service as provided in the previous paragraph.

```
HttpWebRequest request = (HttpWebRequest)HttpWebRequest.Create(serviceUri);

//Set the request headers
request.Credentials = new NetworkCredential(myUserName, myPassword);
request.Method = "POST";
UTF8Encoding encoding = new UTF8Encoding();
request.ContentLength = encoding.GetByteCount(myPayload);
request.ContentType = "application/x-ssds+xml";

//Write the XML payload to the request stream
using (Stream myStream = request.GetRequestStream())
{
        myStream.Write(encoding.GetBytes(myPayload),
          encoding.GetByteCount(myPayload));
}
```

When the operation has completed, SDS will send an HTTP response. The response includes an HTTP status code that indicates whether or not your operation was successful (a status code of 201 indicates that a resource was created).

Note: *For full end-to-end code examples in C#, Java, and Ruby, together with SOAP examples in C#, visit the SDS Primer site on MSDN at http://msdn.microsoft.com/en-us/library/cc512417.aspx.*

Creating Containers

You can use containers to store homogeneous data (entities of the same kind) or heterogeneous data (entities of different kinds). If you use a container to store homogeneous data, the container is analogous to an individual database table. If you use a container to store heterogeneous data, the container is analogous to an entire database. Whereas a traditional database uses tables and schemas to organize data, containers impose no such restraints. Instead of tables and schemas, you use entity

metadata to structure your data storage and retrieval. We'll cover this in more detail when we discuss entities.

Going back to the Coho Winery example, our next step is to create containers to store wine listings and media files. Because containers do not require schemas, the creation process is exactly the same for both containers: you don't need to create columns or specify what type of entity the container will hold.

You can create a container in a very similar way to how you create an authority. First, you assemble an XML payload that describes your container:

```
<s:Container xmlns:s='http://schemas.microsoft.com/sitka/2008/03/'>
   <s:Id>Wines</s:Id>
</s:Container>
```

To create a container, you must scope your operation at the authority level. To do this, you address your HTTP request to the authority URI:

```
https://coho.data.beta.mssds.com/v1/
```

As before, you use the HTTP POST method to perform a create operation, and you send your XML payload to SDS over the request stream as a byte array. When the operation has completed, check the status code of the response to verify that your container was created successfully.

Creating Entities

When you work with flexible entities (in other words, non-BLOB entities), you need to be aware of two different types of properties. Flexible properties are defined by the user and stored in a simple string object dictionary. You can add as many or as few flexible properties to each entity as you want. Currently, you can define property values in **String**, **Base64Binary**, **Boolean**, **Decimal**, and **DateTime** data types.

Metadata properties are fixed properties that provide information on the entity itself. Each entity (with the exception of BLOB entities, which we'll cover later) has three metadata properties.

Metadata Property	Purpose
ID	Provides a unique string identifier for each entity.
Version	Maintains a system-generated version number for each entity. Enables synchronization between data sources and provides support for conditional updates. Be aware that version numbers are not incremental and do not start at 1.
Kind	Assigns an arbitrary type to each entity, such as **RedWine** or **WhiteWine**. Enables you to query specific kinds of entity from heterogeneous containers.

Table 3: Entity metadata properties

Generally speaking, you will only create authorities and containers when you provision or reorganize your data. By contrast, creating entities is a frequent operation in most applications that use SDS as entities to represent your data. As with authorities and containers, when you create an entity using the REST protocol your first step is to create an XML payload that describes your data:

```
<RedWine xmlns:s='http://schemas.microsoft.com/sitka/2008/03/'
      xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
      xmlns:x='http://www.w3.org/2001/XMLSchema' >
      <s:Id>CohoSauvignon05</s:Id>
      <vineyard xsi:type='x:string'>Coho Vineyard</vineyard>
      <vintage xsi:type='x:string'>2005</vintage>
      <grape xsi:type='x:string'>Cabernet Sauvignon</grape>
      <region xsi:type='x:string'>New Zealand</region>
      <description xsi:type='x:string'>Full-bodied and fruity</description>
      <thumbnail xsi:type='x:base64Binary'>/9j/4AAQSkZJ… </thumbnail>
      <price xsi:type='x:decimal'>16.95</price>
 </RedWine>
```

The name of the parent element defines the **Kind** metadata property value of the entity, and the **Id** child element specifies the obligatory ID value. You can add as many flexible properties as you want by adding further child elements, with an appropriate **type** attribute in each case. Note that in this example we store a thumbnail image as a base64Binary encoded value.

To create an entity, you must scope your operation to the parent container. To do this, you address your HTTP request to the container URI:

```
https://coho.data.beta.mssds.com/v1/wines/
```

Again, as with authorities and containers, you use the HTTP POST method to perform a create operation, and you send your XML payload to SDS over the request stream as a byte array. When the operation has completed, check the status code of the response to verify that your entity was created successfully.

Creating BLOB Entities

In SDS, BLOBs are stored as entities. You can add BLOB entities to containers in much the same way that you would add any other entity. However, the metadata properties of BLOB entities differ from those of regular entities in two key ways. First, the value of the **Kind** property is always **Entity** and cannot be changed at present. Second, BLOB entities have an additional metadata property named **Content** that includes the following attributes.

Content attribute	Purpose
content-type	Specifies the MIME type of the BLOB content.
content-length	Specifies the size of the BLOB.
content-disposition	Optionally specifies how the client should handle the BLOB. For example, Web browsers use the **ContentDisposition** header to suggest a filename

	when you save the BLOB as a file.

Table 4: Attributes of the Content metadata property

As with regular entities, to create a BLOB entity you must scope your operation to the parent container. To do this, you address your HTTP request to the container URI:

```
https://coho.data.beta.mssds.com/v1/media/
```

Next, you add the metadata properties to the request header. You use the **Slug** header to define the ID value you want to apply to the entity. As with all "create" operations, you use the HTTP POST method. The following C# code sample illustrates this process.

```
HttpWebRequest request = (HttpWebRequest)HttpWebRequest.Create(containerUri);
request.Method = "POST";
request.ContentLength = 1234567;
request.ContentType = "video/x-msvideo";
request.Headers["Content-Disposition"] =
      "attachment; filename='Coho1.avi'";
request.AllowWriteStreamBuffering = false; //enable full streaming
request.Headers["Slug"] = "CohoSauvignon05TastingNotes";
```

Finally, you use an asynchronous operation to write your data to the request stream. To do this, you typically open a stream to the object you want to upload and copy your data across in small chunks, such as 64 kilobytes at a time.

Remember, for full end-to-end code examples you can visit the SDS Primer site on MSDN at http://msdn.microsoft.com/en-us/library/cc512417.aspx.

Retrieving Your Data

When you use SDS, you can retrieve your data in two ways. You can retrieve authorities, containers, and entities directly by URI. Alternatively, you can scope queries at the service, authority, or container levels to retrieve direct descendants subject to the query criteria and conditions.

Direct Retrieval

Let's start by looking at how to retrieve authorities, containers, and entities directly. To retrieve an item, simply address an HTTP GET request to the appropriate URI. For example, to retrieve the **coho** authority, address your HTTP request to the authority URI:

```
https://coho.data.beta.mssds.com/v1/
```

The service responds with an XML payload that contains the authority metadata:

```
<s:Authority xmlns:s="http://schemas.microsoft.com/sitka/2008/03/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:x="http://www.w3.org/2001/XMLSchema">
      <s:Id>coho</s:Id>
      <s:Version>3593083</s:Version>
```

```
</s:Authority>
```

Remember that the **Version** property value is a non-incremental system-generated number. In other words, if you've just created an entity, don't assume that the version number will be 1.

To retrieve the **CohoSauvignon05** entity, address your HTTP request to the entity URI:

```
https://coho.data.beta.mssds.com/v1/wines/cohosauvignon05
```

Note: *In this URL,* ***coho*** *represents your authority,* ***wines*** *represents your container, and* ***cohosauvignon05*** *represents your entity.*

The service responds with an XML representation of the entity:

```
<RedWine xmlns:s='http://schemas.microsoft.com/sitka/2008/03/'
      xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
      xmlns:x='http://www.w3.org/2001/XMLSchema' >
      <s:Id>CohoSauvignon05</s:Id>
      <s:Version>210441</s:Version>
      <vineyard xsi:type='x:string'>Coho Vineyard</vineyard>
      <vintage xsi:type='x:string'>2005</vintage>
      <grape xsi:type='x:string'>Cabernet Sauvignon</grape>
      <region xsi:type='x:string'>New Zealand</region>
      <description xsi:type='x:string'>Full-bodied and fruity</description>
      <thumbnail xsi:type='x:base64Binary'>/9j/4AAQSkZJ… </thumbnail>
      <price xsi:type='x:decimal'>16.95</price>
 </RedWine>
```

You'll notice that this matches the XML payload we used earlier to create the entity, except that this also includes a **Version** element.

While these basic fetch operations are fine if you know exactly what you're looking for, in most cases you will want to specify queries to provide more sophisticated data retrieval.

Building a Query

SDS supports a simple, text-based query syntax, based on the C# Language Integrated Query (LINQ) pattern. Each query takes the following format:

```
from e in entities [where condition] order by [property] select e
```

The **from e in entities** format applies regardless of whether you are retrieving authorities, containers, or entities. In SDS, authorities, containers, and entities are all types of flexible entity. The scope of your query determines the type of flexible entity (authority, container, or entity) that your query will return. For example, if you scope a query to an authority, your query will return containers. Similarly, if you scope a query to a container, your query will return entities.

You can use comparison operators to add conditions to your queries. You can also use Boolean operators to concatenate or further manipulate your conditions. The current beta version of SDS supports the following operators.

Comparison operators	> (greater than) >= (greater than or equal to) < (less than) <= (less than or equal to) == (equal to) != (not equal to)
Boolean operators	&& (logical AND) || (logical OR) ! (logical NOT)

Table 5: Supported comparison and Boolean operators

Let's look at a few examples. In each case, assume that we have scoped each query to the **Wines** container. We'll look at how to scope and submit a query in the next section.

First, suppose you want to retrieve all red wines from the **Wines** container. You would structure your query as follows:

```
from e in entities where e.Kind == "RedWine" select e
```

Because **Kind** is a strongly-typed metadata property, you use the dot notation in comparisons. In contrast, you use an indexer notation to query flexible properties. Suppose you want to refine your query to return only red wines from New Zealand:

```
from e in entities where e.Kind == "RedWine" && e["region"] == "New Zealand"
select e
```

You can use brackets to explicitly set the order of comparison for more complex queries. Suppose you want to broaden your query to return red wines from either New Zealand or South Africa:

```
from e in entities where e.Kind == "RedWine" && (e["region"] == "New Zealand"
|| e["region"] == "South Africa") select e
```

Finally, to show how you might use other comparison operators, suppose you want to return red wines where the grape is anything but Merlot and your price limit is $15:

```
from e in entities where e.Kind == "RedWine" && e["grape"] != "Merlot" &&
e["price"] <= 15 select e
```

Note: *The public beta version of SSDS will also support* ***join*** *queries.*

Now that you know how to build a query, let's look at how to scope and submit your queries to SDS.

Submitting a Query

To submit a query by using the REST protocol, you must:

1. Get the URI of the authority or container (or the root service URI) you want to scope your query to.
2. Append the query text to the scope URI as a query string.

3. Send an HTTP GET request to the resulting URI.

This makes the query process very straightforward – the URI specifies the scope, and the query string specifies the query.

For example, to submit a query to the **Wines** container, you would append your query to the container URI:

```
https://coho.data.beta.mssds.com/v1/wines/?='from e in entities…'
```

The service returns an XML payload that contains the collection of flexible entities that match your query conditions:

```
<s:EntitySet xmlns:s="http://schemas.microsoft.com/sitka/2008/03/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:x="http://www.w3.org/2001/XMLSchema">
      <RedWine>
      …
      </RedWine>
      <RedWine>
      …
      </RedWine>
</s:EntitySet>
```

This pattern applies regardless of the scope at which you submit your query. The service always returns an **EntitySet** element that contains all the child entities that match the query condition. These child entities could be authorities, containers, or entities, depending on the scope of the query. If no matching entities are found, the service returns an empty **EntitySet** element.

Finally, you can submit an empty query to return all the entities within a specified scope. For example, if you send an HTTP GET request to the following URI, the service will return an **EntitySet** that contains every entity in the **Wines** container:

```
https://coho.data.beta.mssds.com/v1/wines/?=''
```

Retrieving BLOB Data

To retrieve a BLOB entity, send an HTTP GET request to the entity URI. You cannot use queries to return BLOB entities, as the query results will include only the BLOB metadata.

In some applications, you may want to retrieve both the BLOB entity itself and its associated metadata. When you build the HTTP request, you can set the **Accept** header to specify whether the service should return the BLOB data itself or an XML document that represents the BLOB metadata:

- If you set the **Accept** header to the MIME (Multipurpose Internet Mail Extensions) type of your BLOB data (**video/x-msvideo** in the AVI example we used earlier), the service will return the BLOB data in the response body.
- If you set the **Accept** header to **application/x-ssds+xml**, the service will return the entity metadata in the response body, in the same format as any other entity.

Updating Your Data

So far we've covered how to provision your data source, add data, and retrieve data. Now let's look at how to update existing data on SDS.

Conceptually, to update an entity in SDS you simply retrieve your entity, make your changes, and send the updated entity back to SDS. When you use the REST interface, this process is very similar to creating an entity. However, whereas you use the HTTP POST method to create a new entity, you use the HTTP PUT method to replace an existing entity.

Suppose that you want to update the **price** property of the **CohoSauvignon05** entity that we created earlier.

1. Send an HTTP GET request to **https://coho.data.beta.mssds.com/v1/wines/cohosauvignon05**.
2. Retrieve the XML payload returned by the GET request, and modify the value of the **price** element.
3. Send an HTTP PUT request to **https://coho.data.beta.mssds.com/v1/wines/cohosauvignon05**, and add the modified XML payload to the request stream.
4. Check the status code of the response to ensure that your entity was updated successfully.

When you update an entity, SDS will automatically update the **Version** metadata property of the entity. We'll look at how versioning can be used in conditional operations later in this document.

Updating BLOB Data

You can update BLOB entities in the same way that you update regular entities. Use the HTTP PUT method to stream your updated BLOB data to the entity URI. SDS will replace the entire BLOB entity with the new content.

Conditional Operations and Versioning

You can take advantage of the **Version** metadata property to perform conditional retrieve, update, and delete operations on entities. For example, you might want to:

- Retrieve an entity only if your local copy is out of date.
- Update an entity only if the server version matches the client version.
- Delete an entity only if the server version matches an expected value.

When you use the REST interface, you can make retrieve, update, and delete operations conditional by setting the **If-Match** or **If-None-Match** request headers to a specific version number. As you might expect, if you set the **If-Match** header, the operation will only complete if the version number of the target entity on SDS matches the value of the **If-Match** header. Similarly, if you set the **If-None-Match** header, the operation will only complete if the version number of the target entity on SDS does not match the value of the **If-None-Match** header.

The following C# code sample illustrates how to construct an HTTP request for a conditional GET operation. The service will only return the entity if the version number is not 210441:

```
WebRequest request = HttpWebRequest.Create(entityUri);
request.Credentials = new NetworkCredential(myUserName, myPassword);
request.Method = "GET";
request.Headers[HttpRequestHeader.IfNoneMatch] = "210441";
request.ContentLength = 0;
request.ContentType = "application/x-ssds+xml";
```

If the target entity does not match your versioning criteria, the service will instead return a **Not Modified** HTTP status code (HTTP 304).

Note: *If you need to establish the version number of an entity on which you have performed an operation, retrieve the* ***ETag*** *response header returned by the service.*

Deleting Your Data

We've now looked at how to provision your data source and how to add, retrieve, and update data. In this final section, we'll describe how you can delete entities from SDS.

To delete an entity in SDS, you set the scope to the entity you want to delete, and then you perform the delete operation. The process is exactly the same regardless of whether you want to delete a container or an entity (standard or BLOB). However, you should be aware that the current version of SDS will not allow you to delete an authority.

When you use the REST protocol, all you need to do is to send an HTTP DELETE request to the URI of the entity you want to delete. For example:

- To delete the **CohoSauvignon05** entity, send an HTTP request that specifies the DELETE method to **https://coho.data.beta.mssds.com/v1/wines/cohosauvignon05**.
- To delete the entire **Wines** container, send an HTTP request that specifies the DELETE method to **https://coho.data.beta.mssds.com/v1/wines**.

The service will respond with an HTTP status code that indicates whether the delete operation was successful.

Current Limitations

The current beta version of SDS has various limitations that you should be aware of before you start developing your own applications. These limitations are likely to change as the service evolves.

When you create an authority, the authority ID can contain only lowercase letters, numbers, or dashes. Currently, you cannot delete authorities. Each authority can contain up to 1,000 containers.

In the current beta version, each container can store a maximum of 100MB of flexible entities and 1GB of BLOB entities. Flexible entities are limited to 2MB, and BLOB entities are currently limited to 100MB. You cannot update BLOB metadata properties; if you need to change BLOB metadata, you must delete

and recreate the BLOB. Currently, you must use the REST interface, rather than SOAP, to handle BLOB entities.

Conclusion

In this paper, we've introduced SQL Data Services (SDS) as a cloud-based alternative to traditional onsite data infrastructures. We've examined some of the advantages that the service can offer in terms of flexibility and scalability, reliability and security, and developer agility, and we've explained the key features of data model and the application programming interface (API). Finally, we've described how to start using SDS and how you can create, retrieve, update, and delete your data.

For more information, you can also find links to videos, blogs, and other information on the SDS Web site (http://www.microsoft.com/sql/dataservices). For end-to-end code examples in C#, Java, and Ruby, please visit the SDS Developer Center on MSDN (http://msdn.microsoft.com/en-us/library/cc512417.aspx). You can also download the SDS SDK (http://msdn.microsoft.com/en-us/sqlserver/dataservices/cc512120.aspx).