Microsoft

Architecting Applications to use Windows Azure AppFabric Caching

Scott Seely, Friseton

June 2011

Contents

Introduction

If you architect or develop business applications, you know that it is really hard to determine up front what the performance, scale and resilience characteristics of your applications need to be. Even if you do know, you are likely well aware that the requirements and usage does not stay constant. So you really need to craft applications that fundamentally can cope with change. That, as you know, is very hard.

One of the characteristics of the Cloud and in particular, Microsoft's bet on Platform as a Service (PaaS), is the elastic nature of the platform that manages all the moving parts of your applications, allowing them to scale quickly, be resilient to failures and perform well. This is different from Infrastructure as a Service (IaaS) where fundamentally, you are still doing all these things yourself; you just don't have to provision the physical hardware.

Does PaaS change how you build applications? Not really. It is the platform and the underlying platform infrastructure that is responsible for horizontal, elastic scale. However, you do still need to construct your applications so that the various moving parts of your applications can be scaled well.

One of the most important areas this occurs is how to scale your data. Data is at the heart of your application; it is stored in one or more application databases. You query for data, you create and store new data, you reference data sets to make choices etc. You know that the database itself is usually the biggest bottleneck in your application and so you look for strategies to offload tasks from the database and to get data as close as possible to the application components that need it.

Microsoft has introduced the Windows Azure AppFabric Caching service, a platform service to help you implement these data caching strategies.

In this article, we will take a look at what this service offers, how to get started and we will be looking at the different types of cache patterns the Caching service supports. We will examine the key issues introduced by a cache that you need to consider as you architect applications to take advantage of caching.

Windows Azure AppFabric

Before we dive into the AppFabric Caching service, let's briefly take a look at the broader AppFabric middleware platform. Microsoft's strategy is to create a single set of platform services that are symmetrical between on-premise servers and Windows Azure. Not only will this platform be symmetrical from a development perspective, but the product engineering behind it will be a single platform that will offer PaaS both on traditional servers as well as in the cloud.

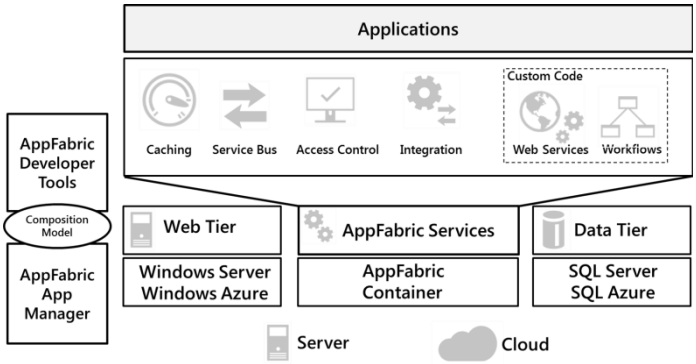

Figure 1 - AppFabric - Next Generation Middleware Platform

Figure 1 represents the complete set of capabilities being created and that are being released initially on Windows Azure. These services will be delivered in quick iterations over the next several months and hardened on Windows Azure. This will be followed by an on-premises version of the platform. You can expect that the on-premises release to be consistent with other server releases with a major version every 2-3 years.

Here's a brief description of the key capabilities:-

- **Caching:** An explicit in-memory .NET object cache to store data that can improve application performance and scale.
- **Service Bus:** Provides messaging services such as queuing, eventing, and publish/subscribe to connect your application components as well as to connect to other systems in the cloud or on-premise.
- **Access Control Service:** A secure token service, relying party, and identity provider all rolled into one. The service allows applications to integrate social and enterprise claims based identity.
- **Integration:** A set of components that provide "last mile" connectivity, transformation and adapters to LOB systems, alongside specific B2B and trading partner capabilities. This service also provides broader business process centric components for business rules and process analytics.
- **Custom Code:** Along with infrastructure services, you need to run custom code that you create. AppFabric provides runtime services for both .NET 4 code as well as workflows (using Windows Workflow Foundation 4). The platform provides all the scaling, resilience and state management for your components.
- **AppFabric Developer Tools and App Manager:** Enhancements in Visual Studio provide a composition environment that will support all the granular components the individual services expose. This will allow the developer to focus on connecting the services needed with business logic in the middle-tier as well as with the web and data tiers of an application. The App Manager can take the Composition Model and use this to provision all parts of the application across the platform as well as manage the application as a single entity.

So, that's the whole AppFabric platform. It is going to be interesting to see this platform evolve and start to use the great capabilities that will be provided. Now, you don't have to wait for some of them, they are already here and in production ready to be used in your applications, like the AppFabric Caching, Access Control, and Service Bus services. The rest of this paper is focused on the AppFabric Caching service, so let's dive in and start by taking a look at the architecture and what you can use caching for.

AppFabric Caching Service Architecture Overview

AppFabric Caching is a service for your applications to use. Your application decides what goes in the cache, your application code retrieves data from the cache. Figure 2 illustrates how the Caching service fits into a typical Windows Azure application. The application's executable code lives in Windows Azure Roles. The executable code decides what data ought to always come from the authoritative store and what data can be remembered.

Sometimes, the data comes from an authoritative source, sometimes not, all depending on the application design. All communication between data sources and the cache goes through the application.

The Caching service maintains a set of key/value pairs in memory. Applications, which are typically worker and web roles, access the cache over a secure

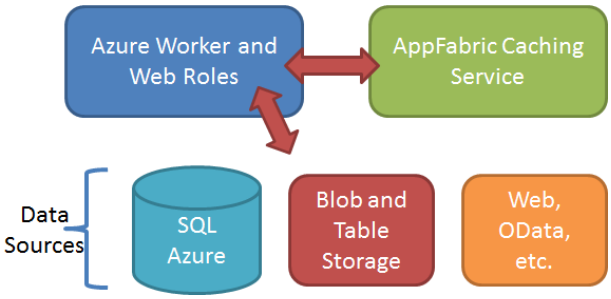

Figure 2 - Pieces of the Windows Azure Data Story

connection. Some values may be accessed in rapid succession or may change very slowly. To take advantage of this, the AppFabric Caching service libraries can also use memory within the local application to create a second level cache. Figure 3 illustrates these components and their relationships.

Integrating the Caching service into an application involves a number of architectural decisions. Many of these decisions relate to what type of data you have in your applications and its suitability to be cached. Let's take a quick look at these types next.

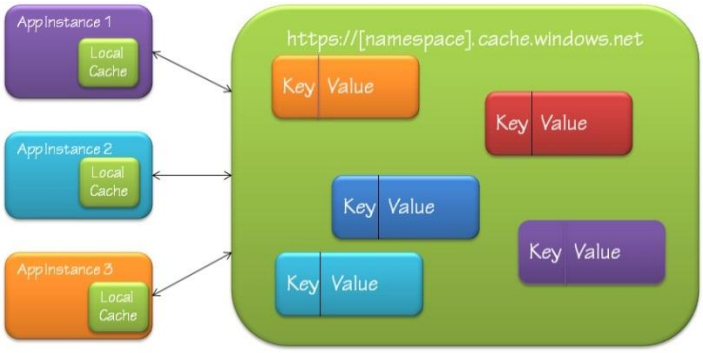

Figure 3 - Caching Service Acts as a Volatile Data Store

Types of Cached Data

Three types of data frequently find their way into a caching solution and do so for different reasons:

- **Reference data:** Data such as country names, store locations, and product catalog data all change fairly slowly. This data has a high volume of reads, almost no writes. This data is a great candidate for caching.
- **Activity data:** Data that is primarily created as a result of an individual user's activity. Typically this is user state data, like a shopping cart. This data may or may not find its way to the database, for example if a user commits to an order. This data is also great for caching until it needs to be committed to the database.
- **Resource data:** Resources are things in your applications that are consumed by users, like a seat on a place, a book ordered from your store, an item in an auction. Typically there is contention for these resources; you cannot allow multiple users to have the same instance. However, even here, you can improve application scale by changing how you think about this contention. For example, you could always take orders for books without regard to whether you have the inventory to ship. Since you don't need to check inventory and manage contention, you can scale much better. For orders that cannot be shipped immediately, you inform the user about the out of stock item, allow them to cancel etc. This would allow you to cache resource data but to also to cope with some staleness in the inventory data.

Reference data and Activity data are the two types of data that are most suitable for caching and have the greatest immediate impact on application performance and scale. By moving reference data into the Cache, you take unnecessary load off the database – allowing the database to focus on resource data. You could use the cache for resource data, and the cache contains the necessary concurrency and locking semantics, but it's harder and this workload should usually be left to the database.

Before we talk any more about architecture, it's time to take our first look at the Cache and get the basics over with so that you can understand how to "get one" and how to use it.

The Caching Service "*Hello, World!*"

When introducing a new technology, it often helps to start with a simple example. In this section, we will write an application that reads a number of stock symbols from the command line and displays their values. Stock data changes many times per second. For our purposes, we will tolerate data that has aged for up to a minute. The application itself will try to get the stock data from the cache and if the data is not present, will put the data into the cache from an external stock feed. This will illustrate the most fundamental caching pattern, often called "cache-aside".

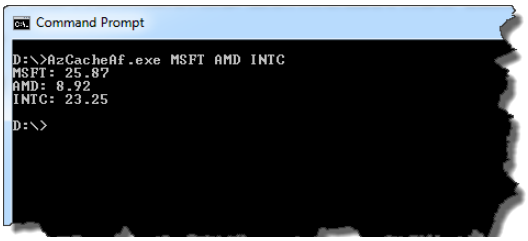

Figure 4 - Application interaction

Figure 4 shows the desired look and feel of the command line application.

Provisioning the Cache

The application will store the value of each stock in the Caching service using the stock ticker symbol as a key. The first thing we will do is provision a cache using the following steps:

1. Navigate to https://windows.azure.com.
2. Go to the *Service Bus, Access Control & Caching* management view. (Figure 5)
3. From here, select *New Namespace.*
   A namespace is used to uniquely identify your cache and of course you will need to use this namespace in your applications so that you can interact with you" cache. By default, the dashboard provisions services for the Caching service, Access Control service, and Service Bus. You can have multiple caches and therefore multiple namespaces in an application.

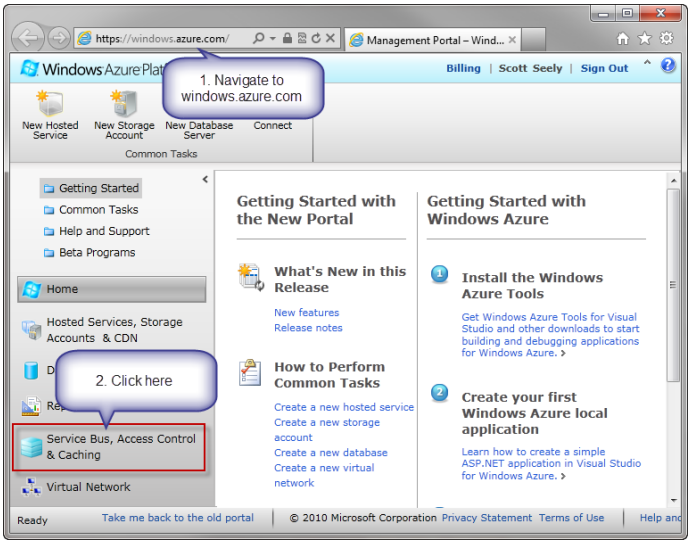

Figure 5 - Azure Portal

   Pick a Service Namespace, a Region, and a Cache Size. The available cache sizes are 128MB, 256MB , 512MB, 1GB, 2GB and 4GB (Figure 6).
4. Press Create Namespace.

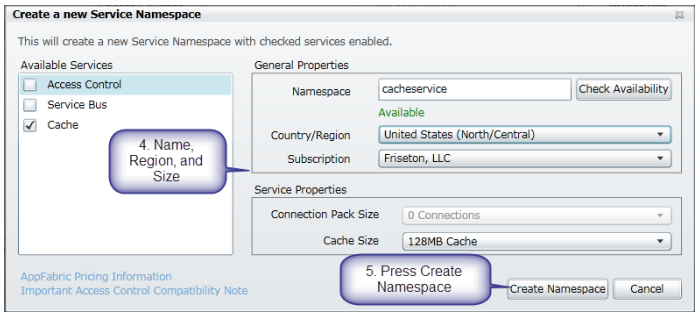

Figure 6 - Configure the New Caching Service

At this point, we have a cache to use. Just to illustrate how to actually use it, we are going to create a simple console application that interacts with the cache. This console app will be running on our local machine and connect to a cache in Windows Azure. Of course in the real world, you would probably not go across this boundary due to the latency, so the Windows Azure AppFabric

Caching service should mainly be used from Applications running in Windows Azure.

Writing the Application

The application itself is quite simple. Here is a flowchart of what needs to happen, the basic flow of the cache-aside pattern. Cache-aside simply means that the cache sits by the side of whatever the data source is - all my clients check if the data is in the cache first. If the desired data it is not in the cache, the requesting client will get the data from the data source and then store it in the cache, making it available to other clients on subsequent requests.

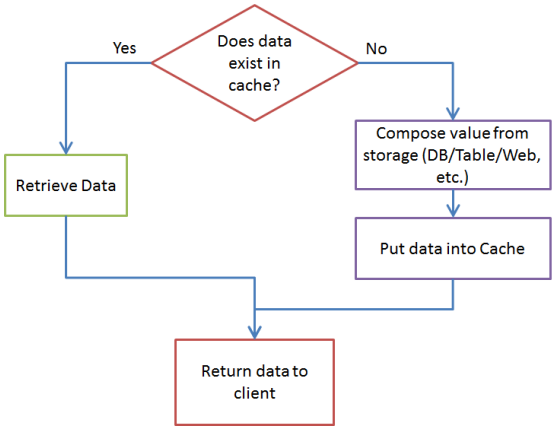

Figure 7 - Cache Aside Flowchart

In our application, we look in the cache for a stock symbol and if it's not present we go get the stock symbols and price from a stock service, and then store the data to the cache.

Before we actually write any code though, we have to tell our application about the Caching service. We do this by copying the configuration that was created on the cache portal into our app.config file. Additionally we also need to reference the AppFabric Caching DLL's in our console application.

First, create the Console Application in Visual Studio and make sure that the Target Framework is set to .NET Framework 4. (By default, console applications target the .NET Framework 4 Client Profile and this application will need access to the full profile because of the caching DLLs.) Next, we add an application configuration file and add the following lines:

```
<configSections>
  <!-- Append below entry to configSections. Do not overwrite the full section. -->
  <section name="dataCacheClients" type="Microsoft.ApplicationServer.Caching.DataCacheClientsSection,
Microsoft.ApplicationServer.Caching.Core"
     allowLocation="true" allowDefinition="Everywhere"/>
</configSections>
<dataCacheClients>
  <dataCacheClient name="default">
    <hosts>
      <host name="cacheservice.cache.windows.net" cachePort="22233" />
    </hosts>

    <securityProperties mode="Message">
      <messageSecurity
         authorizationInfo="[key elided]">
      </messageSecurity>
    </securityProperties>
  </dataCacheClient>
</dataCacheClients>
```

We then add the following two DLL's to our project references, which can typically be found under C:\Program Files\Windows Azure AppFabric SDK\V2.0\Assemblies\Cache.

- Microsoft.ApplicationServer.Caching.Client
- Microsoft.ApplicationServer.Caching.Core

Our code will be reading some HTML. In order to process the HTML, we will use the HtmlAgilityPack. Assuming you have Visual Studio 2010 SP1 installed, you can add this library to your project through NuGet. Just right click on your project and select Add Library Package Reference. Then, search for HtmlAgilityPack, as shown in Figure 8.

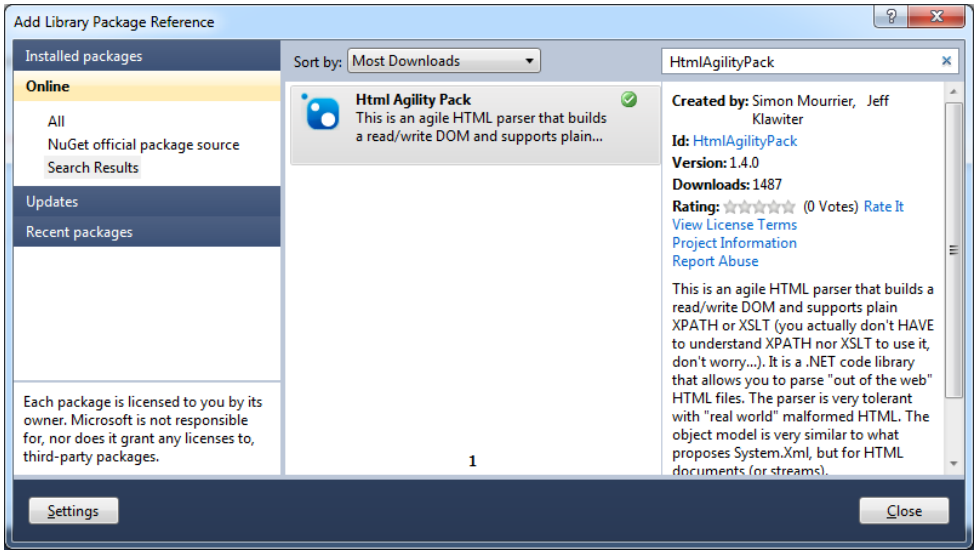

Figure 8 Adding the HtmlAgilityPack to the Project

Finally, we write some code to read the data from the command line. The code is going to perform the following steps:

1. Initialize the cache factory and connect to the default cache.
2. Loop through each symbol.
3. Print the value of each symbol to the Console.

```
class Program
{
  static DataCacheFactory _factory;
  static DataCache _cache;
  static void Main(string[] args)
  {
    // Setup the connection to the cache.
    _factory = new DataCacheFactory();
    _cache = _factory.GetDefaultCache();
    foreach (var symbol in args)
    {
      // Lookup the stock price.
      Console.WriteLine("{0}: {1}", symbol, GetStockPrice(symbol));
    }
  }

  static decimal GetStockPrice(string symbol)
  {
    var retval = new decimal(-1);

    // Check the cache
    var data = _cache.Get(symbol);
    if (data == null)
    {
      // Data wasn't found. Go to the source.
      var client = new WebClient();
      var html = client.DownloadString(
        "http://investing.money.msn.com/investments/stock-price/?Symbol=" + symbol);
      var doc = new HtmlDocument();
```

```
      doc.LoadHtml(html);
      var stock = doc.DocumentNode.SelectSingleNode(@"//span[@class = 'lp']/span");
      if (stock != null)
      {
        decimal temp;
        if (decimal.TryParse(stock.InnerText, out temp))
        {
          retval = temp;

          // Populate the cache.
          _cache.Put(symbol, retval, TimeSpan.FromMinutes(1));
        }
      }
    }
    else
    {
      retval = (decimal)data;
    }
    return retval;
  }
}
```

The `GetStockPrice` method uses the cache-aside pattern we mentioned earlier. The code looks to see if a key/value pair is in the cache. If the data is present, the code uses the cached value; otherwise, the code constructs the value by requesting the quote from a web page and then reading the value of the stock. That value is then stored in the cache so that the next check for the stock symbol will find the value.

The final thing you might notice about the code is that the stock value is added to the cache along with a TimeSpan value. The TimeSpan indicates when the value should be evicted from the cache, freeing up the memory and letting a cache user know that it is time to get a fresh value from the MSN web site.

Implicit and Explicit Caching

Whenever code knows about the cache, we call that explicit caching because any code must be aware of the fact that a cache actually exists and be written to read and write data from the cache. Sometimes, an application just needs to transparently use the cache as a storage service. When this feat can be achieved via configuration without rebuilding the application, we call it implicit caching. The Hello World example showed explicit caching in action.

ASP.NET enables implicit caching through the SessionState provider and output caching. As a cache user, you add the following configuration to your web.config file within <system.web> to use the appropriate Session Store provider and Output Cache provider in order to store Session State and output cache in the Caching service.

```
<sessionState mode="Custom" customProvider="AppFabricCacheSessionStoreProvider">
  <providers>
    <add name="AppFabricCacheSessionStoreProvider"
         type="Microsoft.Web.DistributedCache.DistributedCacheSessionStateStoreProvider, Microsoft.Web.Distribute
dCache"
         cacheName="default"
         useBlobMode="true"
         dataCacheClientName="default" />
  </providers>
</sessionState>
```

```
<caching>
  <outputCache defaultProvider="DistributedCache">
    <providers>
      <add name="DistributedCache"
           type="Microsoft.Web.DistributedCache.DistributedCacheOutputCacheProvider, Microsoft.Web.DistributedCache"
           cacheName="default"
           dataCacheClientName="default" />
    </providers>
  </outputCache>
</caching>
```

Implicit and explicit caching is all about the level of effort a developer must expend in order to use the cache. In the explicit model, the developer directly interacts with the cache, implementing any and all code that interacts with the cache as shown in Figure 9. In the implicit model, the developer interacts with an extensibility mechanism. The session state and caching configuration above uses the ASP.NET provider model to inject a new cache mechanism into the application. In implicit caching, the application code uses the cache almost as a side effect; the act of using the extensibility point implicitly states that the Caching service will be used. Figure 10 illustrates this mechanism, showing that the implicit model adds a layer of abstraction to hide the caching details from the user.

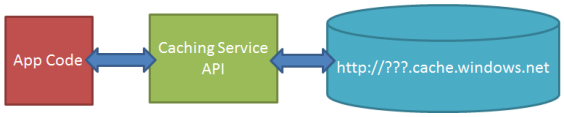

Figure 9 Explicit Caching Interaction

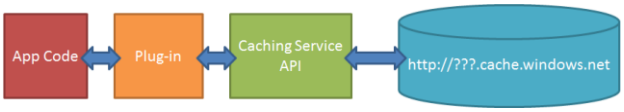

Figure 10 Implicit Caching Interaction

Windows Azure AppFabric Caching Service Concepts

When working with the Windows Azure AppFabric Caching service, you will need to understand some of the cache's basic features. In this section, we cover how objects move between applications and the Caching service, how the Caching service recovers memory when the cache fills up, and how to make use of local cache and avoid calling out to the Caching service for frequently accessed data.

Object Serialization

The underlying bits in the Caching service implementation use the `NetDataContractSerializer` to add objects to the cache. Each object in the cache must have a serialized form that is under 8 MB in size. Finally, objects in the cache must have one of the following traits:

- Marked with the `[Serializable]` attribute.
- Marked with the `[DataContract]` attribute.
- Implement `ISerializable`.

This gives the type fidelity and easy type matching when reading objects to and from the cache. This also requires all clients using the cache to reference the same assemblies for any common objects. Any assembly differences will

cause issues in reading and writing objects. Use clean caches when updating the assemblies that reference common cache objects. Applications can accomplish this in a couple of different ways:

1. Create a new set of caches for the new implementation. Run the applications on different caches, eventually phasing out the old caches and versions as the application rolls out.
2. Declare a maintenance window on the application. Then, shut down all running applications, restart the caches, and bring the new instances online.

Whatever the application does, it should not try to run two different versions of the same objects in the same caches using shared key structures. This can result in deserialization issues as objects move from cache to application. The issues will only show up on the applications that retrieve objects.

Expiration and Eviction

The `DataCache`'s `Put`, `PutAndUnlock`, `Unlock`, and `Add` methods have overloads that take an explicit `TimeSpan` value. This value allows the clients to declare how much longer a value can live in the cache before automatically expiring. By default, objects in the Caching service never expire. When adding an object to the cache, consider how long the object should be allowed to remain in the cache and set the expiration when Adding or Putting any data. Objects are automatically removed from the cache when they reach their expiration time.

When the cache becomes full, the cache evicts items using a least recently used policy. Items that have been accessed least recently leave the cache first. The cache considers itself full when it memory usage reaches a high watermark, as determined by the Caching service. Upon hitting the threshold, the cache evicts items until it reaches its low watermark.

The cache cannot evict a locked object until after the system removes the lock. This allows the cache and cache clients to behave in a predictable fashion.

Using Local Memory for Caching

The Caching service can make use of local memory and avoid hitting the remote servers most of the time by adding the following:

```
<dataCacheClient name="cacheservice-1">

  <localCache isEnabled="true" ttlValue="300" objectCount="10000"/>
  <!-- remainder elided -->
</dataCacheClient>
```

Adding that bit of configuration will dramatically improve response times for frequently accessed data, but will allow the local cache to get out of sync with the hosted Windows Azure AppFabric Cache. By default, objects live for 5 minutes in the local cache and the local cache supports up to 10000 unique objects. You can tune these values to meet the application's requirements.

To manage cache settings, I recommend keeping a spreadsheet around that has a table like this with a row for each cached item:

Cache Namespace: *contoso*

.NET Type	Key Pattern	Average Size (bytes)	Time to Live (s)
Contoso.Person	Person.[Id]	150	60
Contoso.CountryList	Countries	4000	86400
Contoso.Order	Order.[Id]	11000	5

The table documents quite a bit:

- Link between objects and names.
- Size of objects.
- Time to live.

When setting up the `ttlValue`, use a value that is equal to or less than the smallest time-to-live setting for all objects. In our case, the `ttlValue` should be set to 5 because `Contoso.Order` cannot be more than 5 seconds old. It is possible, however, to use multiple data cache clients, each with their own local cache settings. This can be done programmatically or by using a new dataCacheClients configuration section. See the MSDN documentation (http://msdn.microsoft.com/en-us/library/gg278356.aspx) for more information.

The other value you can tune is how many objects you should keep in the local cache. For an application where most pages are unique per user, the calculation would involve finding the following values:

- `ttlValue` time period.
- Number of objects in cache that will be served by pages in that time period. We have no idea which pages will be served, but we have an idea about the number of concurrent users during that time period. Recall that this assumes a worst case where all cached objects are user-specific. For 25 unique users and three pages, the table looks like this:

Page	Number of Cached Objects	Objects in cache for ttlValue
Home/Index	3	3 * 25 = 75
Home/About	1	1 * 25 = 25
Home/Other	10	10 * 25 = 250

The above implies that the local cache for this small application should set the the local cache configuration as:

```
<localCache isEnabled="true" ttlValue="5" objectCount="350"/>
```

These calculations are just guidelines; you should inform the settings with real world data. Watch other metrics like page response times to adjust these settings as needed.

Cache Security

To access the cache, a caller needs two pieces of information: the Caching service URL and the cache Authentication Token. Because of the simple security placed on the cache, only trusted applications on trusted

devices should have access to the Authentication Token. Applications that you own and that run in Windows Azure should have access to the Authentication Token.

However, applications running on user devices should never have access to this information. Why? Access to the application running on Windows Azure has a number of safeguards in place that you can trust and that the attacker will find extraordinarily difficult to compromise. On the other hand, a phone, laptop, web browser, or other device gives the attacker too much access because they can simply dig into the code and other artifacts to discover the key. Although you can use the Windows Azure AppFabric Caching service from an external non-Windows Azure application, this is not a supported design. The Windows Azure AppFabric Caching service is designed to be used only with Windows Azure applications.

For objects that live in the cache, make sure that the business logic layer understands how to guard access to what gets returned. This often means thinking about the granularity of the objects in the cache and how they are retrieved. For example, the application may need to include security artifacts in the cached objects. Before adding data about a cached object to a response, make sure that the current user has the correct rights on the object.

Another approach is naming the objects. A single instance may have many different representations depending on who asked for the data. For example, users with more rights might see a more completely filled-in object. Imagine a `Person` object stored in the cache with a tag naming pattern of `Person.[Id].[SecurityModifier]`. For a user in the Users security group, they might ask for `Person.4.Users` and only see full name and email address. A user in the Admin security group would ask for the same ID as `Person.4.Admin` and see home address, phone numbers, and other personal data, even though both users were looking at the `Person` object whose ID is 4.

The approach you choose depends on the needs of the application.

Managing the Cache

The Caching service imposes a set of throttles on cache usage to make sure that everyone using a cache gets decent service. The throttles cover four aspects of cache usage: memory, transactions, bandwidth, and connections. The throttles assign quotas to each aspect. Whenever you find that you are regularly exceeding one of these quotas, you can either change your code or change the size of the Caching service you have provisioned. The Caching service refreshes the quota counts on an hourly basis.

The Caching service has you select the cache based on size. When you are selecting a size, you are also selecting the quota values for your cache instance. Microsoft is continuously monitoring and reviewing the cache quotas and cache usage in general as the service gets real world usage. You should check the current production quota limits here: http://msdn.microsoft.com/en-us/library/gg602420.aspx#CACHING_FAQ.

Of course, you need to figure out how these limits apply to your application. Some of the calculations are easy, others are not. One thing to note is that when your application exceeds a quota, the application will have to handle an exception. That exception indicates which throttle was just exceeded. Applications should track these exceptions and use them as a tool to figure out when to upgrade to a larger cache. In this section, we will look at how each of these quotas is calculated.

Memory

As the cache is used, the memory quota is consumed. You should first estimate how much memory you might need. Step two of the capacity planning guide for Windows Server AppFabric contains a lot of good information about sizing your cache. Note that this is the on-premises caching solution, so some of the content does not apply to caching in Windows Azure. To estimate the size of an object for that spreadsheet, you can serialize the object and key using the `NetDataContractSerializer` class. Measure the bytes consumed and then add 1% extra for metadata associated with the object. Finally, round this value off to the nearest 1024 bytes to figure out the total size.

Once you deploy your application, you can compare your estimates to reality. Use the management dashboard to see how much cache you are using. The dashboard displays three key statistics:

1. Current Size: Number of MB in use now.
2. Peak Size (this month): The largest the cache has been in the current month.
3. Peak Size (last year): The largest the cache has been in the last calendar year.

You will want to monitor the Current Size and Peak Size to make sure you have not over or under committed cache. An under committed cache would see low usage compared to the current quota, indicating that you should move to a smaller sized cache.

An over committed cache is a different story. A cache is over committed when the current and peak size statistics are near the limit of the cache. If you are using explicit caching, this may mean several things:

1. Not enough of your objects have reasonable expiration times. Review the code and make sure that expirations are set everywhere. Only accept the default, never expire setting on reference data.
2. Your cache is too small for your current load. Consider moving to a larger cache instance.

Over commitment in implicit scenarios should be considered normal. The data may never time out of the Session State or the page output caches in ASP.NET applications. In this case, you want to make sure that sessions are not timing out too early. So long as you have enough cache to satisfy active users, all should be fine. If your data are being removed too soon, your application needs to provision a larger cache instance.

Transactions

Another quota for the Caching service is the number of calls, or “transactions”, to the cache. When an API call causes the quota to be exceeded, the caller sees an exception. Not all calls count as a transaction. Unauthenticated and exception generating calls to your instance of the Caching service do not count against the quota. Use of the local cache in your configuration can help reduce the number of API calls that actually call to the Caching service since many of those requests might be able to be served locally.

Bandwidth

Every request to the Caching service uses bytes in the bandwidth quota. This includes the actual cached object as well as metadata about the object. Metadata includes things like the size of the key, the name of the cache, and so on. When the bandwidth is exceeded, an exception indicating this fact will be raised in the client application. Like with the transaction quota, use of the local cache can reduce your bandwidth usage; a request serviced by the local machine consumes no bandwidth.

Connections

Each DataCacheFactory can create a number of connections. By default, a single DataCacheFactory will only create one connection. If your application will be in a high throughput scenario, accessing the cache tens or hundreds of times per second per client, you will want to add extra connections to the cache server to improve messaging performance. You can create extra connections by changing the MaxConnectionsToServer property on the DataCacheFactoryConfiguration class or by setting the property in configuration:

```
<dataCacheClient name="default" maxConnectionsToServer="2">
```

Periodically, the Caching service will check the connection count and drop connections until the count is within the quota. The connections are not dropped based on any algorithm, meaning that an old connection is as likely to be dropped as a brand new connection. Once the limit is reached, the Caching service refuses any new connections.

Updating the Cache

Resource and activity data changes over time. When making changes to the values, code has to be careful to make sure that the updates are accurate. This means avoiding scenarios where the last write to the cache wins. To handle these scenarios, the Caching service supports two modes of updates: optimistic and pessimistic updates. The primary difference between these two options is that optimistic updates avoid the use of locks and pessimistic updates rely upon locks. In this section, we will explore both options for updates. The section demonstrates both update patterns through management of a list of the most recent IP addresses that have tried to use the application.

Optimistic Cache Update

When making changes to an item in the cache, one needs to make sure that the changes are happening to the latest version of that object. Without this guarantee, it is possible that an update would put the application into an invalid state. The Caching service does support versioning on the objects; the version is actually part of the object metadata. To retrieve the object with metadata, code should call `DataCache.GetCacheItem`. This method returns a `DataCacheItem`, providing the caller with the value, version information, when the object expires, and any tags associated with the item.

You can implement an optimistic update by following one of two paths, depending on whether the updates from the current operation can merge with other updates. If the update cannot merge, the path is:

1. Read the `DataCacheItem`.
2. Compare the version of the `DataCacheItem` with the version in local memory.
    a. If the versions are different, fail.
    b. If the versions are the same.
        i. Update the value locally.
        ii. Save the value to the store, passing in the version code knows about. `DataCache.Put` has a number of overloads that accept a `DataCacheItemVersion`.
        iii. If an exception occurs, fail the update.

If updates can merge, as in the case of updating the most recent list of visitors, the pattern looks more like this:

1. Read the `DataCacheItem`.
2. Merge known changes into `DataCacheItem.Value`.
3. Save the value to the store, passing in the version code knows about.
4. If no exception occurs, update succeeded.
5. If an exception occurs, return to 1. Constrain number of repeats to avoid endless cycles.

Let's apply this to a cached list of IP addresses. These IP addresses represent the most recent machines to visit a page on our web site. Updating the cached list of recent IP visits has a small set of rules:

1. Any new request moves the IP to the front of the list.
2. The list only has 10 items.
3. The list is updated on every page visit.

Using the rules for an optimistic cache update plus the algorithm for updating recent IP visitors yields the following method, `InsertMostRecentIpAddress`. Note that this function tries to take advantage of the fact that the `MostRecentIpAddresses.get` method already has information on the `DataCacheItem` and passes the value along to minimize calls to the Caching service. This code also assumes a `Utility` class that has maintains a reference to a `DataCache`.

```
private static List<string> InsertMostRecentIpAddress(string userAddress,
  DataCacheItem ipAddressesItem)
{
  var cache = Utility.DataCache;
  List<string> retval = null;

  // Limit retries to 5, then give up.
  for (var i = 0; i < 5; ++i)
  {
    if (ipAddressesItem == null)
    {
      // Should only be here if the Add failed above. Get the item that caused
      // DataCache.Add to fail because it already existed.
      ipAddressesItem = cache.GetCacheItem(MostRecentIpAddressesIpAddressesKey);
      if (ipAddressesItem == null)
      {
        // This can only happen if a remove happened after an add failed or if a
        // remove happened while the version was changing. In that case, we throw
        // up our hands and give up, but return a list with the current client
        // so any foreach loops can stay happy.
        break;
      }
    }

    // Update the list.
    retval = ipAddressesItem.Value as List<string>;

    if (retval == null)
    {
      // type cast failure...
      break;
    }
    if (retval.Contains(userAddress))
    {
      // move the address to the front by removing from current list
      retval.Remove(userAddress);
    }
    retval.Insert(0, userAddress);
    const int maxItems = 10;
    if (retval.Count > maxItems)
```

```
    {
      retval.RemoveRange(maxItems, retval.Count - maxItems);
    }
    try
    {
      // Make sure that the update is only applying to the version we know about.
      // If the current version in cache is newer, cause this to fail and add again.
      cache.Put(MostRecentIpAddressesIpAddressesKey, retval,
        ipAddressesItem.Version);
      break;
    }
    catch (DataCacheException)
    {
      ipAddressesItem = cache.GetCacheItem(MostRecentIpAddressesIpAddressesKey);
    }
  }

  // If retval is null, must have hit one of the failure modes.
  // Just return something to keep the caller happy enough.
  return retval ?? (new List<string> {userAddress});
}
```

Depending on the use cases for the application, the types of data being updated, and more, an optimistic update might not be appropriate. How would one guarantee that upon entering some state, the code can cleanly update a value? For that to happen, one needs to use locks.

Pessimistic Cache Update

A pessimistic update assumes that there will be many entities vying to update some object. Instead of relying on versioning, code can also lock a cached object. Locking the object gives ownership to the owner of the lock handle until the lock is released. Locking code follows these steps:

1. Get and lock the object.
2. If lock fails because the object is already locked, wait, then return to step 1. Set a failure threshold so that the code does not loop forever. Also, consider an increasing back off of some sort.
3. If lock is owned, update object.
4. Update and release the lock.

This pattern works great when the code must update the underlying resource and cannot work with merging data. `DataCache` supports locking updates via the `GetAndLock`/`PutAndUnlock` pair of methods. `GetAndLock` retrieves a value and locks the value for some time duration. This allows the lock to expire naturally if the locking code fails in some unexpected way. `PutAndUnlock`, as well as the `Unlock` method, releases the lock explicitly. Coding in this style requires code to acquire and release the lock as fast as possible so as to avoid blocking other participants.

```
private static List<string> InsertMostRecentIpAddressLocking(string userAddress)
{
  var cache = Utility.DataCache;
  List<string> retval = null;

  // Limit retries to 5, then give up.
  for (var i = 0; i < 5; ++i)
  {
    DataCacheLockHandle lockHandle = null;
    try
    {
```

```
      retval = cache.GetAndLock(MostRecentIpAddressesIpAddressesKey,
                    TimeSpan.FromSeconds(30),
                    out lockHandle) as List<string>;
    }
    catch (DataCacheException ex)
    {
      if (ex.ErrorCode == DataCacheErrorCode.KeyDoesNotExist)
      {
        // Shouldn't be here if key doesn't exist. Go to fallback behavior.
        break;
      }
      if (ex.ErrorCode == DataCacheErrorCode.ObjectLocked)
      {
        // back off, then try again
        Thread.Sleep(TimeSpan.FromSeconds(i));
        continue;
      }
      // Anything else, let someone higher up the stack handle it.
      throw;
    }

    if (retval == null)
    {
      // type cast failure...
      break;
    }
    if (retval.Contains(userAddress))
    {
      // move the address to the front by removing from current list
      retval.Remove(userAddress);
    }
    retval.Insert(0, userAddress);
    const int maxItems = 10;
    if (retval.Count > maxItems)
    {
      retval.RemoveRange(maxItems, retval.Count - maxItems);
    }

    // Update the cache and unlock the key
    cache.PutAndUnlock(MostRecentIpAddressesIpAddressesKey, retval, lockHandle);
    break;
  }

  // If retval is null, must have hit one of the failure modes.
  // Just return something to keep the caller happy enough.
  return retval ?? (new List<string> { userAddress });
}
```

When implementing a pessimistic lock, the developers using the cache have to enforce proper usage. The Caching service has no ability to make sure that locks get used all the time and that the code manages these locks correctly; that is strictly up to code reviews. So long as locks are used consistently, the Caching service will protect the data.

Summary

The Windows Azure AppFabric Caching service contains a lot of great features and capabilities. It will make applications running in Windows Azure respond faster because those applications will be able to fetch data from memory in the datacenter instead of having to always go to a database or other storage medium. When building applications, developers and architects need to decide what kind of update behavior should be implemented.

Most applications can benefit from some level of cache usage. Web applications can store session information, lookup tables, and commonly generated HTML. Taken together, these actions greatly improve the responsiveness of web applications as well as scale by distributing load across both data and cache tiers. Background processes that refer to common data can also benefit for the same reasons: once an oft-used value is known, the cache can hold that information for any sibling workers toiling on similar problem sets.

Perhaps the biggest benefit of the Windows Azure AppFabric Caching service is that it exists as an already installed, easily provisioned service on the Windows Azure platform. Developers will find that including the service in Windows Azure applications is very simple. Many will start with implicit caching by using the Caching service for Session State and Output Caching with ASP.NET and Web roles, achieving improvements in page load times with no code changes. Then with small code changes they can start to move to explicit use of the Caching service where appropriate for even greater gains.

Resources

- Scaling .NET Web Applications with Microsoft's Project Code-named Velocity. http://www.griddynamics.com/solutions/appfabric-white-paper-2011.html

- Windows Azure Platform Training Kit. http://www.microsoft.com/downloads/en/details.aspx?FamilyID=413E88F8-5966-4A83-B309-53B7B77EDF78&displaylang=en

- Windows Azure AppFabric Caching on Channel 9: http://channel9.msdn.com/posts/Karandeep-Anand-Windows-Azure-AppFabric-Caching

- Self-Paced Training Class - Developer Introduction to Windows Server AppFabric - (Part 2): Caching Services: http://msdn.microsoft.com/en-us/windowsserver/gg675186

- Windows Azure Caching MSDN Documentation

- Differences between the Windows Server AppFabric Cache and Windows Azure AppFabric Caching service
    - http://go.microsoft.com/fwlink/?LinkId=216934
    - http://go.microsoft.com/fwlink/?LinkId=216935

Appendix A: Named Caches and Regions

The Windows Server AppFabric Cache supports two features that will find their way to the Windows Azure AppFabric Cache in a future release: named caches and regions. The Caching service, named cache, and region all represent different levels of management. Figure 11 illustrates the containment model.

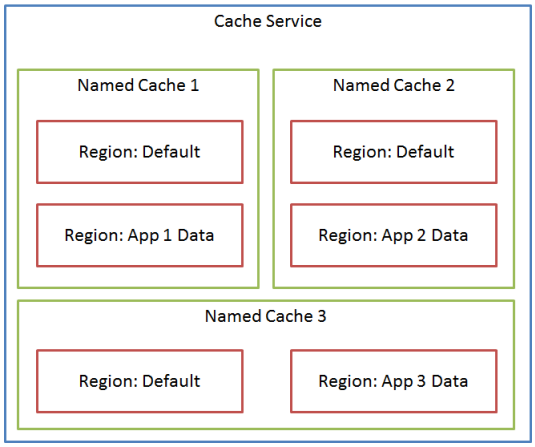

Figure 11 Relations Between Top Level Cache Entities

The Caching service manages the overall amount of memory available for caching. Since the Caching service manages all memory, it manages cache eviction policies. Whenever the percentage of memory usage exceeds the high water mark, the Caching service causes all caches and regions to evict objects until the cache reaches the low water mark percentage. Administrators manage the high water and low water mark values. The Caching service also maintains statistics about the Caching service: total size, number of objects in the cache, information on regions and named caches, number of requests, and number of cache misses.

The caches manage specifics about object lifetimes within the cache. Each cache can have a different policy from other caches. A cache can enable notifications, allowing interested applications to monitor cache item lifecycle events. The cache also controls the eviction policy. The policy may be least recently used (`LRU`) where the least recently accessed items get evicted first. The policy may also be `None`. The `None` value prohibits objects from being ejected and may cause the Caching service to run out of memory. The `None` value works great for sets of values that change rarely. Values within a cache may expire if the cache has enabled expiration. By default, caches enable expiration. If an object is inserted into the cache, the cache controls the default time to live. Putting data like country lists, states, and other data that change extremely slowly work well within a cache that never evicts or expires objects.

Regions define collections of objects. Each cache has a default region and zero or more named regions. In a multi-tenant application, regions can serve to keep the objects from Tenant A separated from those owned by Tenant B. Code can query the collection of objects contained within a region based on keys or tag values, looking for objects that have all tags in some set or one tag out of many. This design can then be used to forcibly evict items that have been affected by a change to some other data. For example, an application may have a Person object whose key is `Person.4`. Any other objects effected by changes to `Person.4` get tagged. Later, when `Person.4` changes, the code looks for related items in the region and either removes those items or updates them to reflect changes to the state of the application.

About the Author

Scott Seely is a cofounder of Friseton (freez-ton), a leading .NET development shop based in Chicago, Illinois. Scott is the author of many books and articles. Scott has produced several courses for Pluralsight, including courses on Windows Communication Foundation, Windows Azure AppFabric, and SQL Azure. Scott has two decades of experience as a software developer and architect, focusing on large scale distributed systems. You can find him

throughout the year speaking at conferences, user groups, and on his blog. You can reach Scott at any time at scott.seely@friseton.com.

Acknowledgements

Wade Wegner and Tony Meleg deserve significant credit for the content found in this whitepaper. Wade helped provide a lot of direction through blog entries, Channel 9 videos, and conversations. Tony had a keen eye on how to tell the caching story and did an excellent job making sure I understood it too. Thanks also to Kent Brown, Jason Roth, and Itai Raz in making sure that the contents in here are correct. Of course, any errors are still my own.