Synchronizing Files to Windows Azure Storage Using Microsoft Sync Framework

*I wanted to kick off this post by introducing myself. I’m Michael Clark. I lead the Sync Framework team. I don’t normally post to this blog (Liam Cavanagh does a fantastic job of that) but wanted a forum to get the information below out for now and decided that blog would be a good venue. One of the things that we’d like to know is if the community would like to see more posts of this nature. If so let us know by giving us some feedback and we’ll figure out some appropriate venue to get this kind of info out on a more regular basis.*

[[](http://microsoftpdc.com/)](http://microsoftpdc.com/)At PDC09 we talked quite a bit about how to synchronize data to the cloud. Most of this revolved around synchronizing structured data to and from SQL Azure. If you are interested in that, check out our [Developer Quick Start](http://www.microsoft.com/windowsazure/developers/sqlazure/datasync/), which includes a link to download [Microsoft Sync Framework Power Pack for SQL Azure November CTP](http://www.microsoft.com/downloads/details.aspx?FamilyID=bce4ad61-5b76-4101-8311-e928e7250b9a&displaylang=en).

For this post, however, I want to augment that information to answer another question that we are frequently asked and were asked a number of times at PDC, “How can I synchronize things like files with Azure Blob Storage?” The answer at this point is that you’ll have to build a provider. The good news is that it’s not too hard. I built one that I’ll describe as part of this post.

So how does it work? The sample itself consists of three major portions: the actual provider, some wrapper code on Azure Blob Storage, and a simple console application to run it. I’ll talk in a bit more depth about both the provider and the Azure Blob Storage integration. On the client side the sample uses Sync Framework’s file synchronization provider. The file synchronization provider solves a lot of the hard problems for synchronizing files, including moves, renames, etc., so it is a great way to get this up and going quickly.

The azure provider is implemented as a FullEnumerationSimpleSyncProvider, using the [simple provider](http://msdn.microsoft.com/en-us/library/dd937763(SQL.105).aspx) components of Sync Framework. Simple providers are a way to create Sync Framework providers for data stores that do not have built-in synchronization support.

The provider itself derives from the [FullEnumerationSimpleSyncProvider](http://msdn.microsoft.com/en-us/library/microsoft.synchronization.simpleproviders.fullenumerationsimplesyncprovider(SQL.105).aspx) class, which is used for stores that don’t support any form of change detection whatsoever. It’s extremely useful because this is actually the category that most off-the-shelf stores fall into (another great example of this type of store is the FAT file system). Sync Framework also contains the notion of an [AnchorEnumerationSimpleSyncProvider](http://msdn.microsoft.com/en-us/library/microsoft.synchronization.simpleproviders.anchorenumerationsimplesyncprovider(SQL.105).aspx) for stores that have the ability to enumerate changes based on some type of anchor (timestamp, tick count, opaque blob of goo, whatever). But for now I’m going to focus on full enumeration as that is what is required for Azure Blob Storage at this point.

The basic idea behind a full enumeration synchronization provider is that you need to tell Sync Framework some basic information about the items you'll be synchronizing, how to identify them and how to detect a version change, and then give Sync Framework the ability to enumerate through the items looking for changes. To tell Sync Framework about the items, override the MetadataSchema property of the FullEnumerationSimpleSyncProvider class. When you build the metadata schema you’ll specify a set of custom fields to track, and an IdentityRule. Together these things make up the set of data required to track and identify changes for objects in the store. For the Azure Blob synchronization provider this property looks like this:

public override ItemMetadataSchema MetadataSchema

{

get

{

CustomFieldDefinition[] customFields = new CustomFieldDefinition[2];

customFields[0] = new CustomFieldDefinition(ItemFields.CUSTOM\_FIELD\_NAME, typeof(string), AzureBlobStore.MaxFileNameLength);

customFields[1] = new CustomFieldDefinition(ItemFields.CUSTOM\_FIELD\_TIMESTAMP, typeof(ulong));

IdentityRule[] identityRule = new IdentityRule[1];

identityRule[0] = new IdentityRule(new uint[] { ItemFields.CUSTOM\_FIELD\_NAME });

return new ItemMetadataSchema(customFields, identityRule);

}

}

Next, in order to let Sync Framework detect changes you need to override the EnumerateItems method. In your EnumerateItems implementation you’ll create a List of ItemFieldDictionary objects to tell Sync Framework about all of the metadata properties you have specified in your MetadataSchema. Sync Framework uses this information to track the state of objects in the store, looking for adds, updates, and deletes and then produces the proper synchronization metadata for those changes so that they can be synchronized with any Sync Framework provider. The implementation for EnumerateItems in this sample looks like this:

// Enumerate all items in the store

public override void EnumerateItems(FullEnumerationContext context)

{

List<ItemFieldDictionary> items = DataStore.ListBlobs();

context.ReportItems(items);

}

There is obviously some hidden detail here because I’ve put some of the work in the store wrapper that I mentioned previously. I’ll talk more about the store wrapper in a moment but I’ll include its ListBlobs method here since it is relevant.

internal List<ItemFieldDictionary> ListBlobs()

{

List<ItemFieldDictionary> items = new List<ItemFieldDictionary>();

BlobRequestOptions opts = new BlobRequestOptions();

opts.UseFlatBlobListing = true;

opts.BlobListingDetails = BlobListingDetails.Metadata;

foreach (IListBlobItem o in Container.ListBlobs(opts))

{

CloudBlob blob = Container.GetBlobReference(o.Uri.ToString());

ItemFieldDictionary dict = new ItemFieldDictionary();

dict.Add(new ItemField(ItemFields.CUSTOM\_FIELD\_NAME, typeof(string), o.Uri.ToString()));

dict.Add(new ItemField(ItemFields.CUSTOM\_FIELD\_TIMESTAMP, typeof(ulong), (ulong)blob.Properties.LastModifiedUtc.ToBinary()));

items.Add(dict);

}

return items;

}

What this ListBlobs implementation does is simply walk through all of the blobs in the container (container is an Azure Blob Storage concept; in this sample I treat it like a root directory). Then, for each blob get out the information that was specified for the MetadataSchema and build a list of that information to give back to Sync Framework.

The last crucial bit for the providers is to give Sync Framework a way to add, update, or delete items from the store. To do this, override [InsertItem](http://msdn.microsoft.com/en-us/library/microsoft.synchronization.simpleproviders.simplesyncprovider.insertitem(SQL.105).aspx), [UpdateItem](http://msdn.microsoft.com/en-us/library/microsoft.synchronization.simpleproviders.simplesyncprovider.updateitem(SQL.105).aspx), and [DeleteItem](http://msdn.microsoft.com/en-us/library/microsoft.synchronization.simpleproviders.simplesyncprovider.deleteitem(SQL.105).aspx) respectively. I won’t include the source for those methods here for brevity but you can check them out in the sample.

There are a couple of other things that are needed for the synchronization provider but those are mostly bookkeeping and I’ll let you look at the sample to get the details.

At this point, I want to talk briefly about the store wrapper class. The store wrapper class utilizes objects in the Microsoft.WindowsAzure.StorageClient namespace. To get that you’ll need to download [Windows Azure Tools for Microsoft Visual Studio](http://www.microsoft.com/downloads/details.aspx?FamilyID=6967ff37-813e-47c7-b987-889124b43abd&displaylang=en). I’m not going to go into too much detail about the store wrapper itself. It has most of the methods you’d expect, such as the ListBlobs method seen above, and the corresponding methods for adds, updates, and deletes. I do want to talk a little bit about one important detail with the store and that is *optimistic concurrency*. Optimistic concurrency is the thing that will allow multiple synchronization clients to work with the store at the same time without overwriting each other unwittingly and corrupting data. The Sync Framework simple providers are designed to work well with stores that support optimistic concurrency so that they can provide correct synchronization.

The great news is that Windows Azure Blob Storage supports optimistic concurrency well. If you are using the StorageClient API, it does this by using a BlobRequestOptions object. You can see an example of how this works in the DeleteFile method of the store wrapper:

internal void DeleteFile(

string name,

DateTime expectedLastModified

)

{

CloudBlob blob = Container.GetBlobReference(name);

try

{

blob.FetchAttributes();

}

catch (StorageClientException e)

{

// Someone may have deleted the blob in the meantime

if (e.ErrorCode == StorageErrorCode.BlobNotFound)

{

throw new ApplicationException("Concurrency Violation", e);

}

throw;

}

BlobProperties blobProperties = blob.Properties;

...

BlobRequestOptions opts = new BlobRequestOptions();

opts.AccessCondition = AccessCondition.IfNotModifiedSince(expectedLastModified);

try

{

blob.Delete(opts);

}

catch( StorageClientException e )

{

// Someone must have modified the file in the meantime

if (e.ErrorCode == StorageErrorCode.BlobNotFound || e.ErrorCode == StorageErrorCode.ConditionFailed)

{

throw new ApplicationException("Concurrency Violation", e);

}

throw;

}

}

Note that by specifying the AccessCondition property of the BlobRequestOptions object, the code tells Azure Blob Storage not to touch the file if the file has been modified since the last time we looked at it. If the file has been touched, the CloudBlog object from the StorageClient library throws StorageClientException. For a couple of specific errors, the store wrapper converts that into ApplicationException to let other parts of the code know that they should treat this as a temporary error and temporarily skip it from the perspective of synchronization. That code is in the DeteleItem method of the provider and looks like this:

try

{

DataStore.DeleteFile(name, expectedLastUpdate);

}

catch (ApplicationException e)

{

recoverableErrorReportingContext.RecordRecoverableErrorForChange(new RecoverableErrorData(e));

}

What this does is cause Sync Framework to temporarily exclude that particular item from synchronization. The item will be picked up again later.

Now, to run this sample you’ll need to download the [Microsoft Sync Framework v2 SDK](http://www.microsoft.com/downloads/details.aspx?FamilyID=89adbb1e-53ff-41b5-ba17-8e43a2e66254&displaylang=en) as well as the [Windows Azure Tools for Microsoft Visual Studio](http://www.microsoft.com/downloads/details.aspx?FamilyID=6967ff37-813e-47c7-b987-889124b43abd&displaylang=en). The sample synchronizes to a Windows Azure Blob Storage project so you’ll need to make sure that you have an account and project set up. To get started on that, go to <http://www.microsoft.com/windowsazure/>. Finally, when running the sample you’ll need to modify the app.config file for the sample and specify the AccountName and AccountSharedKey properties for the storage project that you created. When you actually run it from the command line, the application expects to be given a container name (this is an Azure Blob Storage concept mentioned previously) and a local path. It will synchronize everything from the local path, including subdirectories, up to the container specified.

So that’s basically it. Please give this a try, check out the code, and let us know what you think by sending mail to [syncfdbk@microsoft.com](mailto:syncfdbk@microsoft.com?subject=Azure%20Blob%20Provider%20feedback). If you do this with photos you can easily view them in your web browser by going to https://<accountName>.blobs.core.windows.net/<containerName>/<fileName>.jpg.

One final note that is worth covering: A great question to ask here is, “Why not just use Live Mesh to synchronize the files?” The answer to that is that if Live Mesh fits your scenario and requirements, then you absolutely should use it. Live Mesh, part of the Windows Live Platform, is a great product and will allow you to synchronize files between any set of PCs and mobile devices. It is perfect for a lot of synchronization scenarios involving files. Most of the customers that have asked us about how to accomplish this with Sync Framework need something special. For instance, they are creating an end-to-end application and want explicit control over everything (instead of leaving that up to the end user) including where the files are synchronized, etc. Other examples are customers who specifically want the files in Azure Blob Storage so that they can use them with their Azure Web Application. The bottom line is that if Live Mesh meets your needs, then great. If not, Sync Framework is a perfect alternative for meeting the synchronization needs of your application.

So there it is. This is a simple example of how to synchronize files with Azure Blob Storage. You can take this a lot further by for instance hosting in a Windows Azure Web Role and storing the item metadata directly in the file metadata. But using the method described above is fast to get up and running and performs well for a number of scenarios. Let us know if you have any questions or comments.