MICROSOFT AZURE BATCH SERVICE SAMPLE CODE INSTRUCTIONS

INTRODUCTION

The purpose of this project is to demonstrate how to use the Microsoft Azure Batch .Net client library to interact with Azure Batch services. In this project, we'll see how to run jobs by creating a Task Pool, WorkItems, Jobs, and Tasks. We'll also see how to monitor job and task progress through the API. The application also shows how to use blob storage to stage input and output files.

If you are familiar with Azure Batch services and want to start quickly, jump to Preparation and follow the steps in that section, followed by the steps in the Configuration, Compiling and Running sections.

PREREQUISITES

NUGET

This sample solution depends on the NuGet Visual Studio extension to obtain the necessary prerequisite, dependent .Net Assemblies. You can download the NuGet extension from here. When the first build is attempted, the NuGet extension will automatically download the required assemblies in order to complete a successful build.

AZURE STORAGE MANAGEMENT TOOL

In addition, you'll need one of the 3rd party Azure storage explorers. This document uses the **Azure Management Studio** for this purpose. It's a commercial tool with a 30 day trial. But you can find other Azure Storage tools that support the use of Shared Access Signatures (SAS) which provide finer grained access to data in your storage account.

The following link points to an older list of available tools: http://blogs.msdn.com/b/windowsazurestorage/archive/2010/04/17/windows-azure-storage-explorers.aspx.

IMAGEMAGICK APPLICATION

The sample uses the ImageMagick application which is available at http://www.imagemagick.org. See the Resource section below for more information.

BATCH SERVICE ACCOUNT

You will need a Batch Service account that has Administrative privileges in order to run the ImgProc sample. It installs the ImageMagick application on the Task VM and that requires admin level access in order to complete.

WORKFLOW

The ImgProc program operates in two modes: running locally on your workstation as a client to facilitate the creation of Batch Service items (client mode) and as a task in the Batch Service to perform the conversion of a set of images into a different size. The conversion step is run as a Job in the Batch Service using a set of tasks executing in parallel, each task converting a subset of the images. To do this, the following steps will be taken:

1. Two Azure storage containers (input and resource) are created that respectively hold the images to be converted and the task execution binaries (the ImageMagick installation binary along with the ImgProc binary and its dependent files). These container names are specified in the ImgProc app configuration file.
2. Once the binaries are uploaded, ImgProc is run in client mode to create a Task Pool in the Batch Service. The pool contains multiple Task VMs that run the conversion tasks. The pool is configured with a startup task that installs the ImageMagick application on each VM in the pool. The startup task is executed after a task VM is created and has joined the pool.
3. ImgProc is run in client mode again (with different options) to create a WorkItem and the set of tasks which are added to the WorkItem. The submitted WorkItem uses the Pool created in the previous step. Each task will have a resource dependency on the binaries and the images that are associated with this task. Each task performs the following steps:
    a. All files in the resource container are downloaded to the TVM by the Batch Service before the task on that TVM starts.
    b. Each task is configured to run ImgProc which will download its set of images from the input container and start ImageMagick to process the images.
    c. When ImageMagick has finished, ImgProc will upload the results into the output container and exit which will mark this particular task as completed.
4. After submitting the WorkItem and tasks, the local ImgProc client application will query the status of the tasks in a loop and exit when all tasks have finished. Progress of the application can also be monitored by looking at the output container.

CODE STRUCTURE

The code is packaged as **BatchSamples.sln**. There are 7 projects under the solution.

The first project is **HelloWorld**. This is a very simple project demonstrating the usage of Azure Batch .Net API to submit workitem/tasks to Azure Batch services.

The 2nd project is **ImgProc**. This is both the client app that submits the tasks and the app that will run in a Task VM to process images. “ImgProc.exe” is compiled from this project.

The 3rd project is **TopNWordsSample**. This sample is a batch application in its simplest form with no customized pool setup. The program goes through specified list of files in blob storage and calculate word usage statistics and print it out on client side.

Under the folder of **Protocol**, there are 4 more projects showing the usage of the protocol layer of Azure Batch API.

The first project is **Common**. This is the common library that wraps Storage and Task related functions. “Common.dll” is compiled from this project.

The 2nd project is **ImgProc.Protocol**. This is the same as ImgProc project but uses protocol API.

The 3rd project is **LogSearchSample.Protocol**. This sample demonstrates how to use a Job Manager task to orchestrate map-reduce style workflow. This particular sample searches multiple text-based log files and summarizes the result. This sample uses protocol API.

The 4th project is **TopNWordsSample.Protocol**. This sample is the same as TopNWordsSample but uses protocol API.

PREPARATION

Before we begin, there are a couple of things we need to do - mainly getting Azure Storage ready.

To run **ImgProc**, **HelloWorld**, and **TopNWords** sample, all you need to do is fill in the StorageKey and StorageAccount field in app.config file of the project.

With other protocol samples, you might be able to specify container SAS string without releasing the storage credential. See Appendixes chapter for detail.

RESOURCES

Besides input, output, and resource containers, you'll also need to set ImageMagickExeSAS. Upload ImageMagick installation file to the resource container and update ImageMagickExeSAS field in App.Config.

The latest version of ImageMagick can be downloaded from http://www.imagemagick.org/script/binaryreleases.php#windows. We have tested with version 6.8.7 of the X64 DLL release but other versions will probably work too.

See Appendixes on how to get an SAS URL.

NOTE: When uploading ImageMagick installation package, rename it to ImageMagick.exe as that's hardcoded name used in ImgProc.

CONFIGURATION

After loading the solution file, the first thing we need to do before compiling is to update the configuration file. All the application config settings are stored in App.Config of the ImgProc project. The fields that need to be updated are:

- TaskTenantUrl, Account, Key: This is the batch account information. This is supplied by the Azure Batch Service team.
- NumTasks: # of simultaneous tasks to run against the workload.
- NumTvms: Size of the pool in terms of Task VMs. Change this based on the quota associated with your batch account.
- UploadResources: A Boolean indicating whether TVM binary dependencies need to be uploaded to the resource container. Set to true for the first time and every time you change ImgProc.
- InputDataContainerSAS (and Output/Resource/ImageMagickExeSAS): a SAS string for a container that can be read/write.
    - Input is for input images.
    - Output for results.
    - Resource for dependencies (binaries).
    - ImageMagickExeSAS for the installation package.
- LoggingLevel: currently set to None but other valid values are Verbose and Minimal. This will provide some level of logging from the TaskRequestDispatcher class in the Common assembly.

COMPILING

The solution comes with its own binary dependency (the Azure Batch client dll) so it can be built after Azure Batch SDK is installed. Just build the solution.

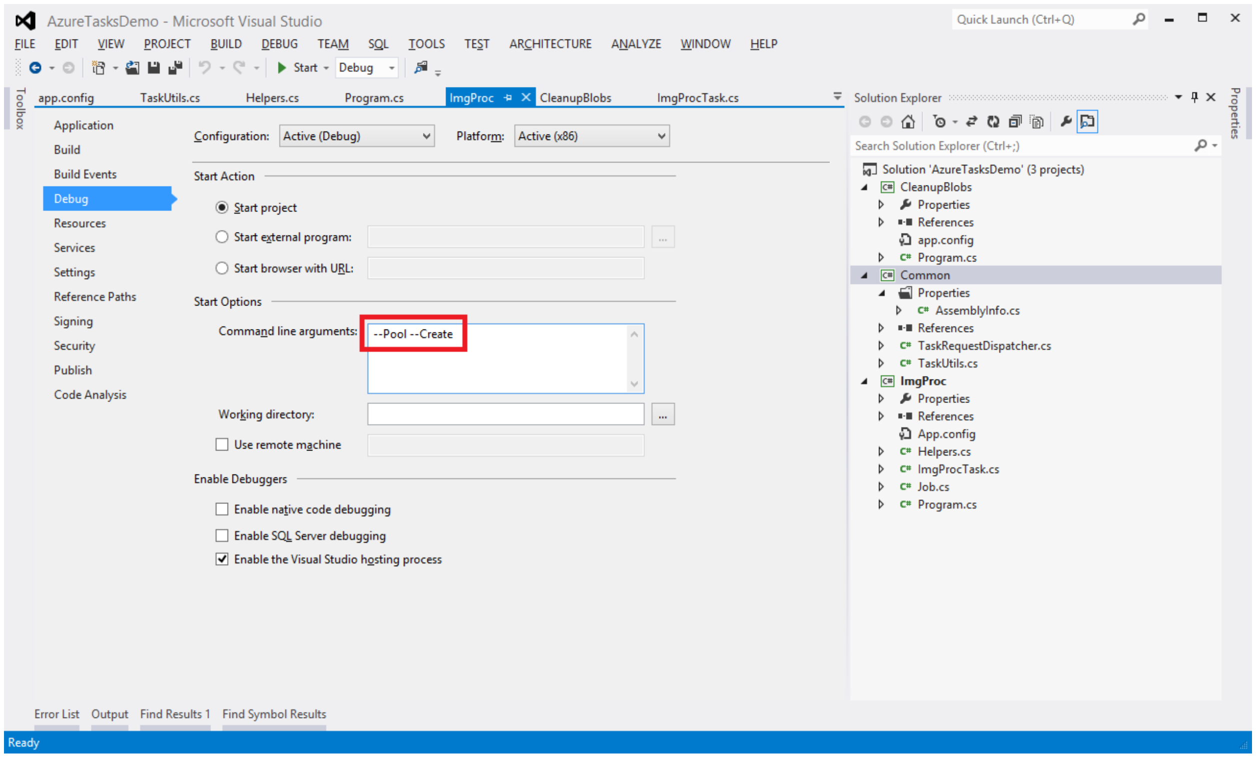
Make sure ImgProc is the startup project in the solution.

NOTE: If you get error in App.Config, note that the ampersand literal ("&") should be replaced by "&amp;" in the string and without the double quote marks. For example: "abcdef$jj!&sslk" should be "abcdef$jj!**&amp;**sslk".

RUNNING

Before the running ImgProc, upload a set of images to the input container that will be converted as described in the Input section.

The first time you run ImgProc, run with UploadResources set to true in app.config using the command line arguments "--Pool --Create". This will make sure that ImgProc.exe and its dependent binaries are uploaded and the pool is created. The command line parameters can be changed in VS as shown below in the ImgProc project properties. This step only needs to be performed once if you intend to reuse the same pool.



Configuration: Active (Debug)   Platform: Active (x86)

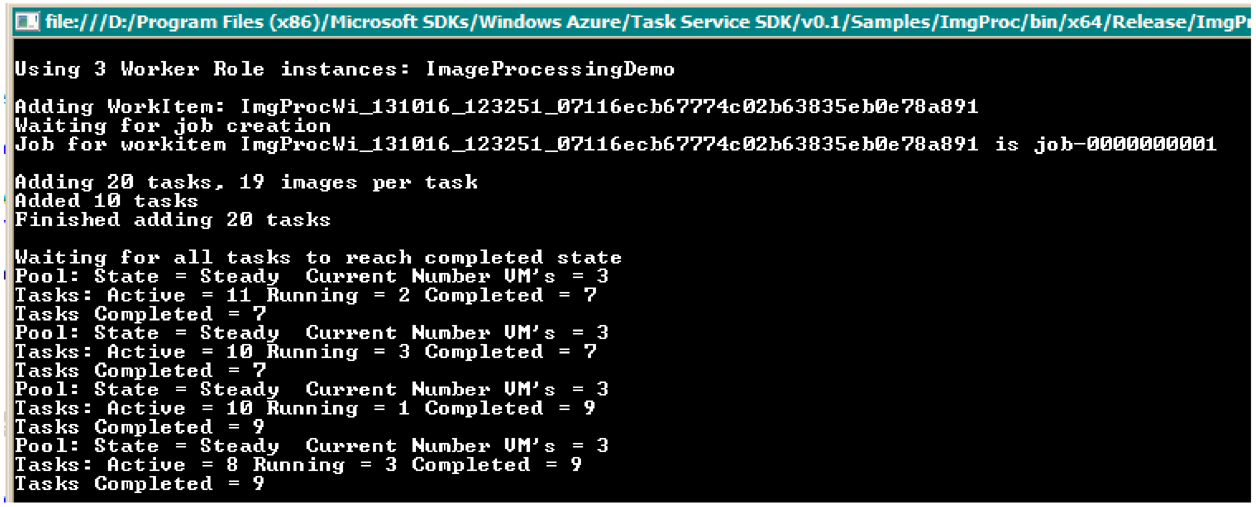
Start Action
- [✓] Start project
- Start external program:
- Start browser with URL:

Start Options
Command line arguments: --Pool --Create
Working directory:
- [ ] Use remote machine

Enable Debuggers
- [ ] Enable native code debugging
- [ ] Enable SQL Server debugging
- [✓] Enable the Visual Studio hosting process

To submit the tasks, remove the "--Pool --Create" arguments from the Command line arguments text box and run ImgProc again. ImgProc will create the WorkItems and add the tasks. After that, it will monitor the status of all the tasks in a loop until everything is done.

Here are some screenshots that shows when the ImgProc client is running properly:



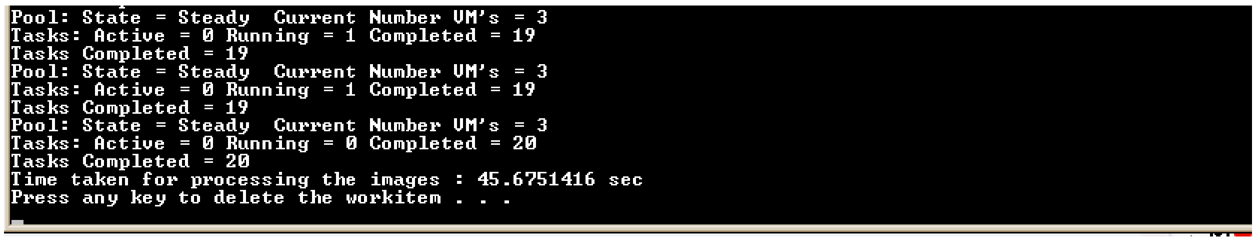
```
file:///D:/Program Files (x86)/Microsoft SDKs/Windows Azure/Task Service SDK/v0.1/Samples/ImgProc/bin/x64/Release/ImgPr

Using 3 Worker Role instances: ImageProcessingDemo

Adding WorkItem: ImgProcWi_131016_123251_07116ecb67774c02b63835eb0e78a891
Waiting for job creation
Job for workitem ImgProcWi_131016_123251_07116ecb67774c02b63835eb0e78a891 is job-0000000001

Adding 20 tasks, 19 images per task
Added 10 tasks
Finished adding 20 tasks

Waiting for all tasks to reach completed state
Pool: State = Steady  Current Number VM's = 3
Tasks: Active = 11 Running = 2 Completed = 7
Tasks Completed = 7
Pool: State = Steady  Current Number VM's = 3
Tasks: Active = 10 Running = 3 Completed = 7
Tasks Completed = 7
Pool: State = Steady  Current Number VM's = 3
Tasks: Active = 10 Running = 1 Completed = 9
Tasks Completed = 9
Pool: State = Steady  Current Number VM's = 3
Tasks: Active = 8 Running = 3 Completed = 9
Tasks Completed = 9
```



```
Pool: State = Steady  Current Number VM's = 3
Tasks: Active = 0 Running = 1 Completed = 19
Tasks Completed = 19
Pool: State = Steady  Current Number VM's = 3
Tasks: Active = 0 Running = 1 Completed = 19
Tasks Completed = 19
Pool: State = Steady  Current Number VM's = 3
Tasks: Active = 0 Running = 0 Completed = 20
Tasks Completed = 20
Time taken for processing the images : 45.6751416 sec
Press any key to delete the workitem . . .
```

CLEANING UP

By default, the WorkItem is deleted when all the tasks have completed. The pool will remain active until it is deleted which means that the task VMs associated with the pool are still being charged to your account. You can run ImgProc with "--Pool --Delete" command line arguments to delete the pool and everything associated with it.

AUTOPOOLS

Using an autopool is another way to create a pool and not worry about pool lifetimes. An Autopool is very similar to a normal Pool but its lifetime is tied to a WorkItem or a Job associated with a WorkItem.

To use Autopools, skip the creation of the normal pool and simply run ImgProc with --AutoPool (one step instead of two). A Pool is created when the WorkItem is created and will be deleted when the WorkItem is deleted.

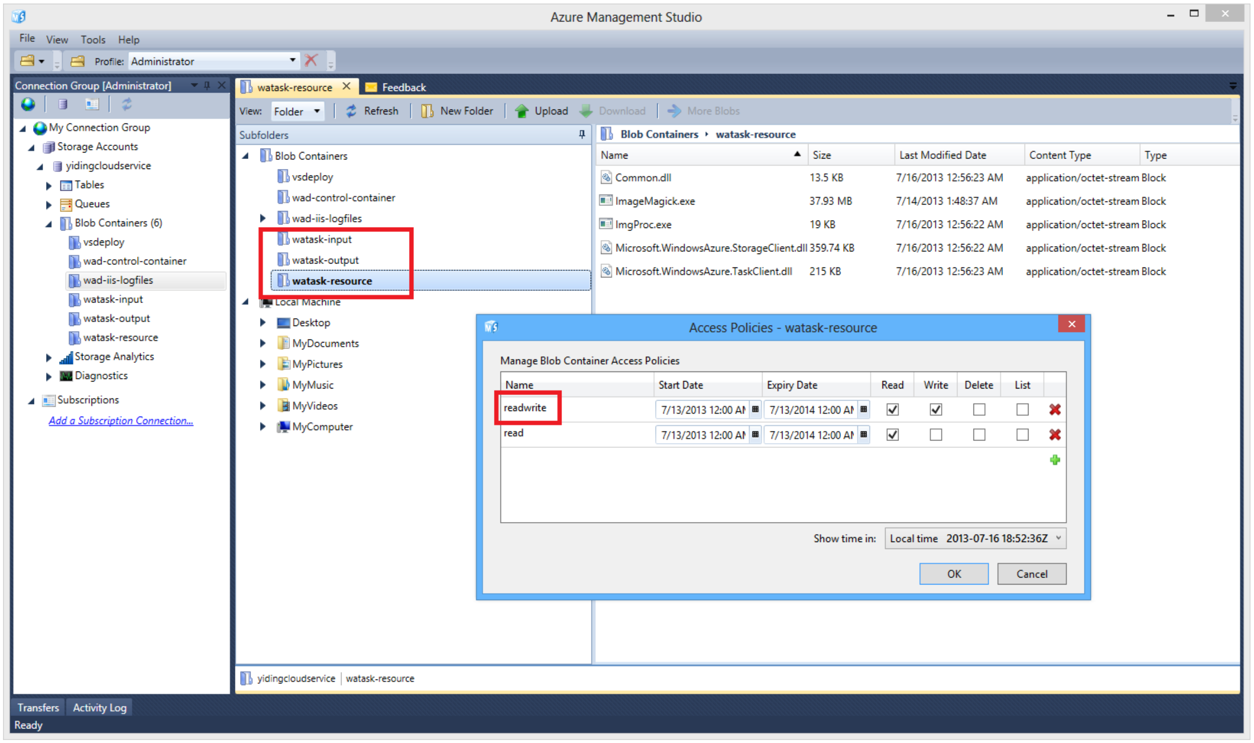
MISCELLANEOUS

Prior to General Availability of the Batch Service, additional client libraries with higher levels of abstraction will be authored and consequently, this sample may not work with these newer libraries.

APPENDIXES

CONTAINERS

As described previously, there are 3 containers needed – input, output, and resource. The input container will hold the images to be converted, the output container will hold the converted images and the resource container will hold the binary files as we discussed in the Workflow section. The following is a screenshot of Azure Management Studio showing the container organization.



Name	Size	Last Modified Date	Content Type	Type
Common.dll	13.5 KB	7/16/2013 12:56:23 AM	application/octet-stream	Block
ImageMagick.exe	37.93 MB	7/14/2013 1:48:37 AM	application/octet-stream	Block
ImgProc.exe	19 KB	7/16/2013 12:56:22 AM	application/octet-stream	Block
Microsoft.WindowsAzure.StorageClient.dll	359.74 KB	7/16/2013 12:56:22 AM	application/octet-stream	Block
Microsoft.WindowsAzure.TaskClient.dll	215 KB	7/16/2013 12:56:23 AM	application/octet-stream	Block

Access Policies - watask-resource

Manage Blob Container Access Policies

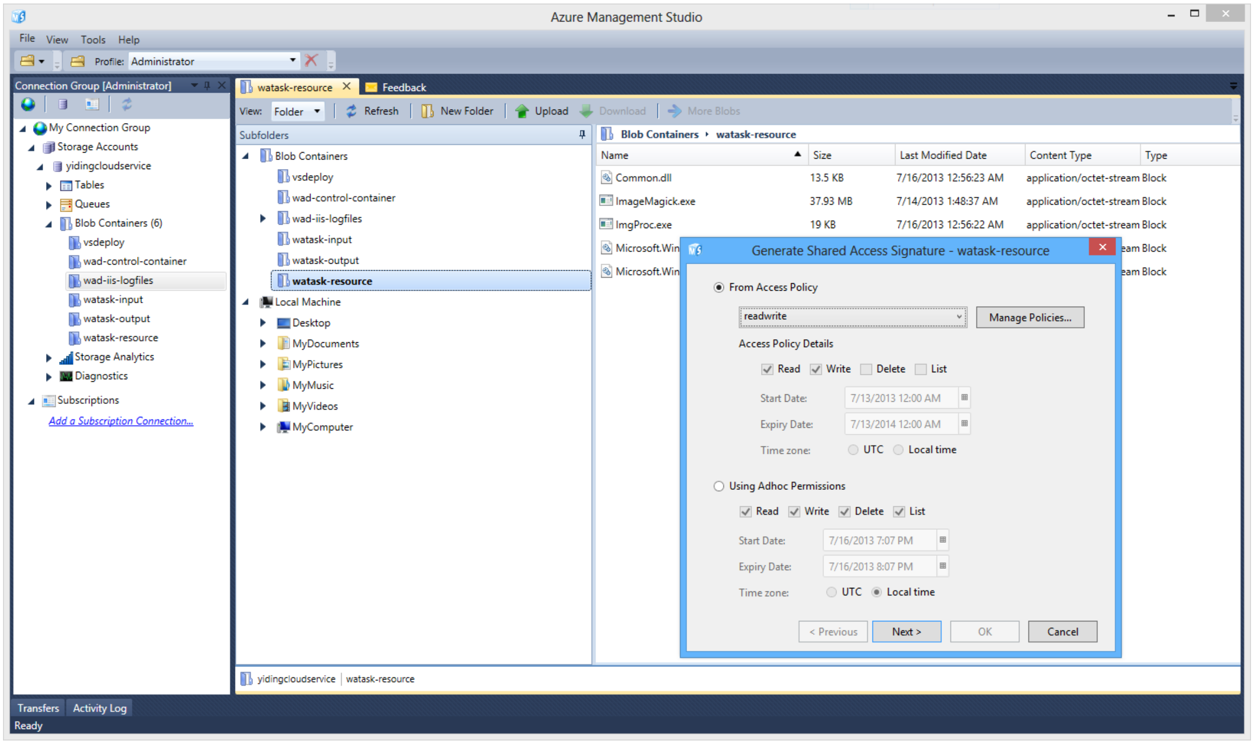
Name	Start Date	Expiry Date	Read	Write	Delete	List
readwrite	7/13/2013 12:00 AM	7/13/2014 12:00 AM	✓	✓		
read	7/13/2013 12:00 AM	7/13/2014 12:00 AM	✓			

Show time in: Local time 2013-07-16 18:52:36Z

OK Cancel

Also, we have associated “access policies” on each container. It is a best practice when using Azure Storage to assign an access policy when creating a shared access signature (SAS) intended to be used by other individuals. For this demo, we need read/write access to the output and resources folder. The Input container SAS can be read only for the demo.

After creating the containers, we need to generate a SAS for each container. Here is a screenshot of how you do that in Azure Management Studio.



Generate Shared Access Signature - watask-resource

(●) From Access Policy

readwrite   Manage Policies...

Access Policy Details

[✓] Read [✓] Write [ ] Delete [ ] List

Start Date: 7/13/2013 12:00 AM

Expiry Date: 7/13/2014 12:00 AM

Time zone: ( ) UTC ( ) Local time

( ) Using Adhoc Permissions

[✓] Read [✓] Write [✓] Delete [✓] List

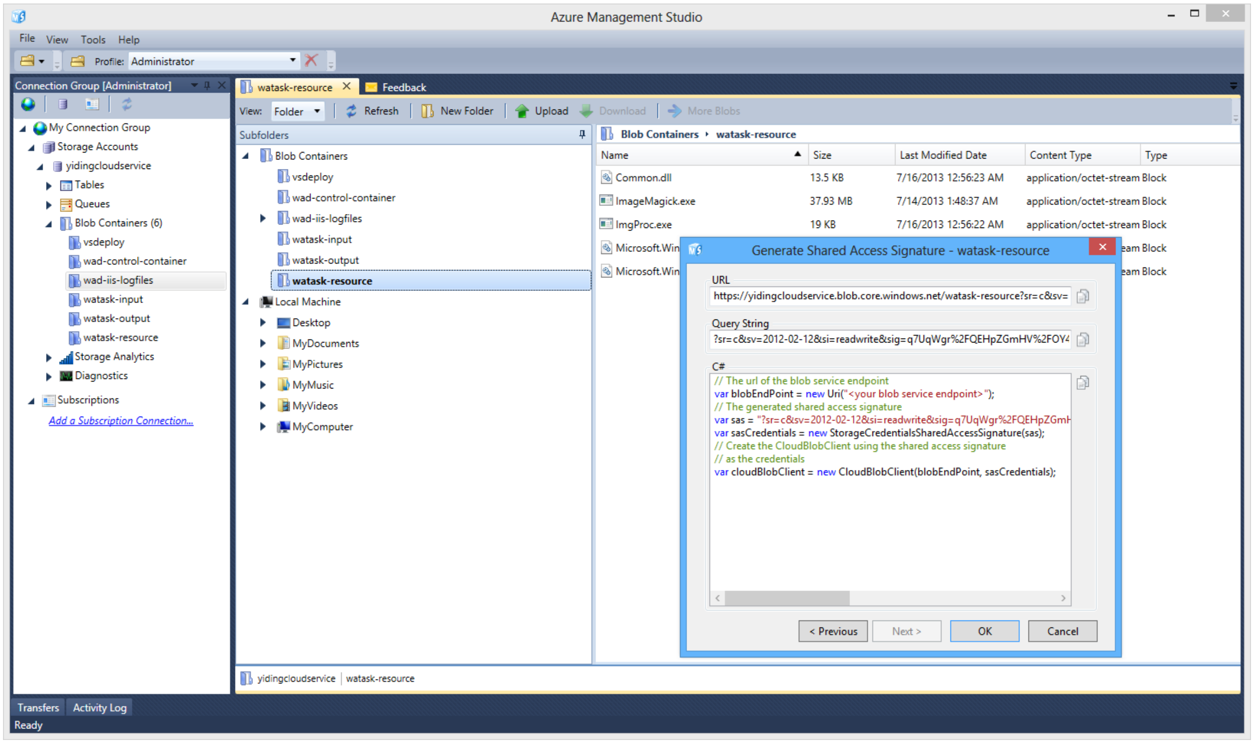
Start Date: 7/16/2013 7:07 PM

Expiry Date: 7/16/2013 8:07 PM

Time zone: ( ) UTC (●) Local time

< Previous   Next >   OK   Cancel

Just right-click the container and select “Generate Signed URL…”. Assuming you have created the appropriate access policy, select “readwrite” in the dropdown list and click Next.



Generate Shared Access Signature - watask-resource

URL
`https://yidingcloudservice.blob.core.windows.net/watask-resource?sr=c&sv=`

Query String
`?sr=c&sv=2012-02-12&si=readwrite&sig=q7UqWgr%2FQEHpZGmHV%2FOY4`

C#
```
// The url of the blob service endpoint
var blobEndPoint = new Uri("<your blob service endpoint>");
// The generated shared access signature
var sas = "?sr=c&sv=2012-02-12&si=readwrite&sig=q7UqWgr%2FQEHpZGmH
var sasCredentials = new StorageCredentialsSharedAccessSignature(sas);
// Create the CloudBlobClient using the shared access signature
// as the credentials
var cloudBlobClient = new CloudBlobClient(blobEndPoint, sasCredentials);
```

Name	Size	Last Modified Date	Content Type	Type
Common.dll	13.5 KB	7/16/2013 12:56:23 AM	application/octet-stream	Block
ImageMagick.exe	37.93 MB	7/14/2013 1:48:37 AM	application/octet-stream	Block
ImgProc.exe	19 KB	7/16/2013 12:56:22 AM	application/octet-stream	Block
Microsoft.Win				Block
Microsoft.Win				Block

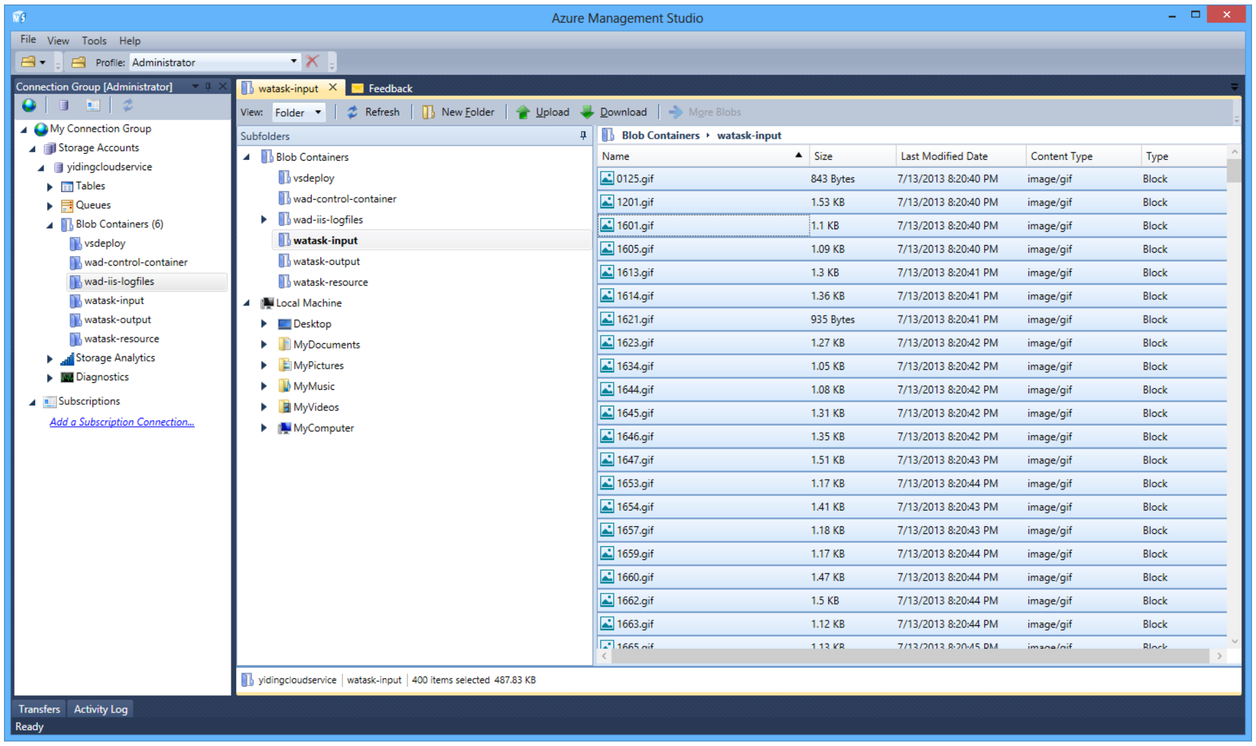
Now copy the content of the URL field and fill that in the corresponding field of App.Config file of ImgProc project (see the Configuration section below for details on how to do that).

NOTE: You will need a separate file-based SAS for the ImageMagick.exe file.

NOTE: A SAS is a URL (or rather a part of one). As the creator of a SAS, you define the validity duration of SAS and the access rights of the user in possession of this SAS to the resources protected by SAS. See this blog entry for more detail.

INPUT

After creating the input containers, we need to upload images to that container. Here is a sample of container with images.



Name	Size	Last Modified Date	Content Type	Type
0125.gif	843 Bytes	7/13/2013 8:20:40 PM	image/gif	Block
1201.gif	1.53 KB	7/13/2013 8:20:40 PM	image/gif	Block
1601.gif	1.1 KB	7/13/2013 8:20:40 PM	image/gif	Block
1605.gif	1.09 KB	7/13/2013 8:20:40 PM	image/gif	Block
1613.gif	1.3 KB	7/13/2013 8:20:41 PM	image/gif	Block
1614.gif	1.36 KB	7/13/2013 8:20:41 PM	image/gif	Block
1621.gif	935 Bytes	7/13/2013 8:20:41 PM	image/gif	Block
1623.gif	1.27 KB	7/13/2013 8:20:42 PM	image/gif	Block
1634.gif	1.05 KB	7/13/2013 8:20:42 PM	image/gif	Block
1644.gif	1.08 KB	7/13/2013 8:20:42 PM	image/gif	Block
1645.gif	1.31 KB	7/13/2013 8:20:42 PM	image/gif	Block
1646.gif	1.35 KB	7/13/2013 8:20:42 PM	image/gif	Block
1647.gif	1.51 KB	7/13/2013 8:20:43 PM	image/gif	Block
1653.gif	1.17 KB	7/13/2013 8:20:44 PM	image/gif	Block
1654.gif	1.41 KB	7/13/2013 8:20:43 PM	image/gif	Block
1657.gif	1.18 KB	7/13/2013 8:20:43 PM	image/gif	Block
1659.gif	1.17 KB	7/13/2013 8:20:44 PM	image/gif	Block
1660.gif	1.47 KB	7/13/2013 8:20:44 PM	image/gif	Block
1662.gif	1.5 KB	7/13/2013 8:20:44 PM	image/gif	Block
1663.gif	1.12 KB	7/13/2013 8:20:44 PM	image/gif	Block
1665.gif	1.13 KB	7/13/2013 8:20:45 PM	image/gif	Block

yidingcloudservice | watask-input | 400 items selected 487.83 KB

I've uploaded 400 GIFs in this case with Azure Management Studio. The sample code is currently hard coded to work on JPEG files but can be easily modified to work on other image types supported by ImageMagick. You'll need to modify the sample to use other image extensions if you are working with PNG or GIF files.