Public Notes: Running Python in Azure Batch on Windows using the Azure Portal

TLDR; This post explains how to setup an Azure Batch node which uses Windows using installers. It also explains how to use application packages to preload Azure Batch nodes with utilities so that tasks can just be command lines. The explanation use case is “run a Python script”, but should apply more broadly to “install tools, distribute stuff, and run command lines.”

When I start experimenting with something, I do not start out with writing code to automate everything. Sometimes, I try to use a GUI to bootstrap my process, then automate when setup is correct. Why? A lot of environments, like Azure, will allow for fast cycles from the UI tools. My latest adventure took a bit of time, so I’m documenting what I did.

Here’s the context: I am developing a mid-sized example project for scalability. If everything goes to plan, the demo will show how to solve the same problem with Azure Batch and the Azure Kubernetes Service. The demo is targeting a special kind of data scientist: an actuary. Actuaries frequently write code in one of three languages: APL, Python, and R. I’m focusing on Python.

My goals:

1. Configure everything through the Azure portal.
2. Install Python on the batch node at startup.
3. Use the application packages feature to deliver the Python virtual environment prior to any tasks running.
4. Run a command line without resources to make sure I can run a script in the Python virtual environment.

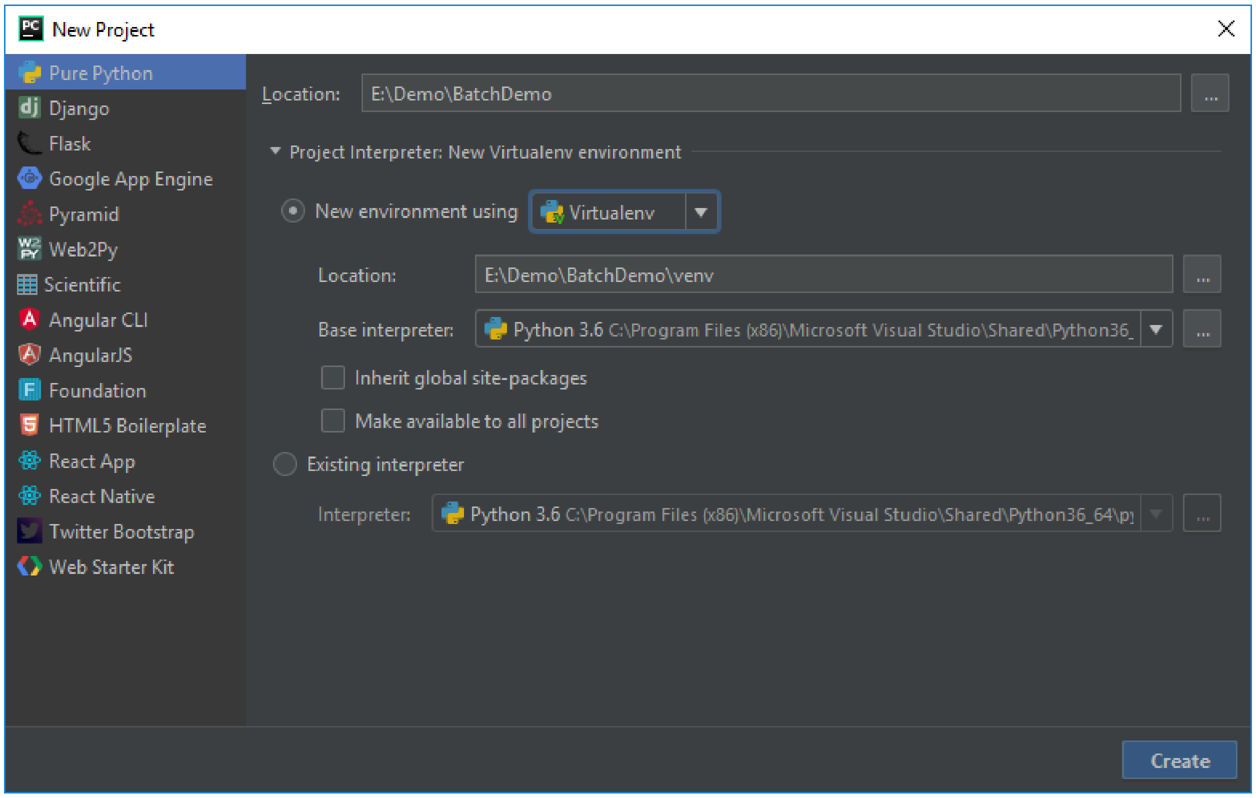
What follows is essentially a lab for you to follow and do the same things I did. As time marches forward, this lab’s correctness will degrade. Hit me up on LinkedIn if you catch this and I may go back and update the details.

For all the Azure pieces, try to keep things in the same region. I’ll be using East US. This isn’t necessary, but is helpful for the parts that transfer files. Staying in the same region gives better speed.

1 Create a new project in PyCharm

a. Open up your editor; I’m using PyCharm. Details for other editors will differ. Set the location to wherever you like. I’m naming the project *BatchDemo*.
b. Setup a New environment using Virtualenv. The location will be in the venv directory of your project. For the base interpreter, use the one installed on your machine already.

For me, the dialog looks like this:


New Project

Location: E:\Demo\BatchDemo

Project Interpreter: New Virtualenv environment

New environment using: Virtualenv

Location: E:\Demo\BatchDemo\venv

Base interpreter: Python 3.6 C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_

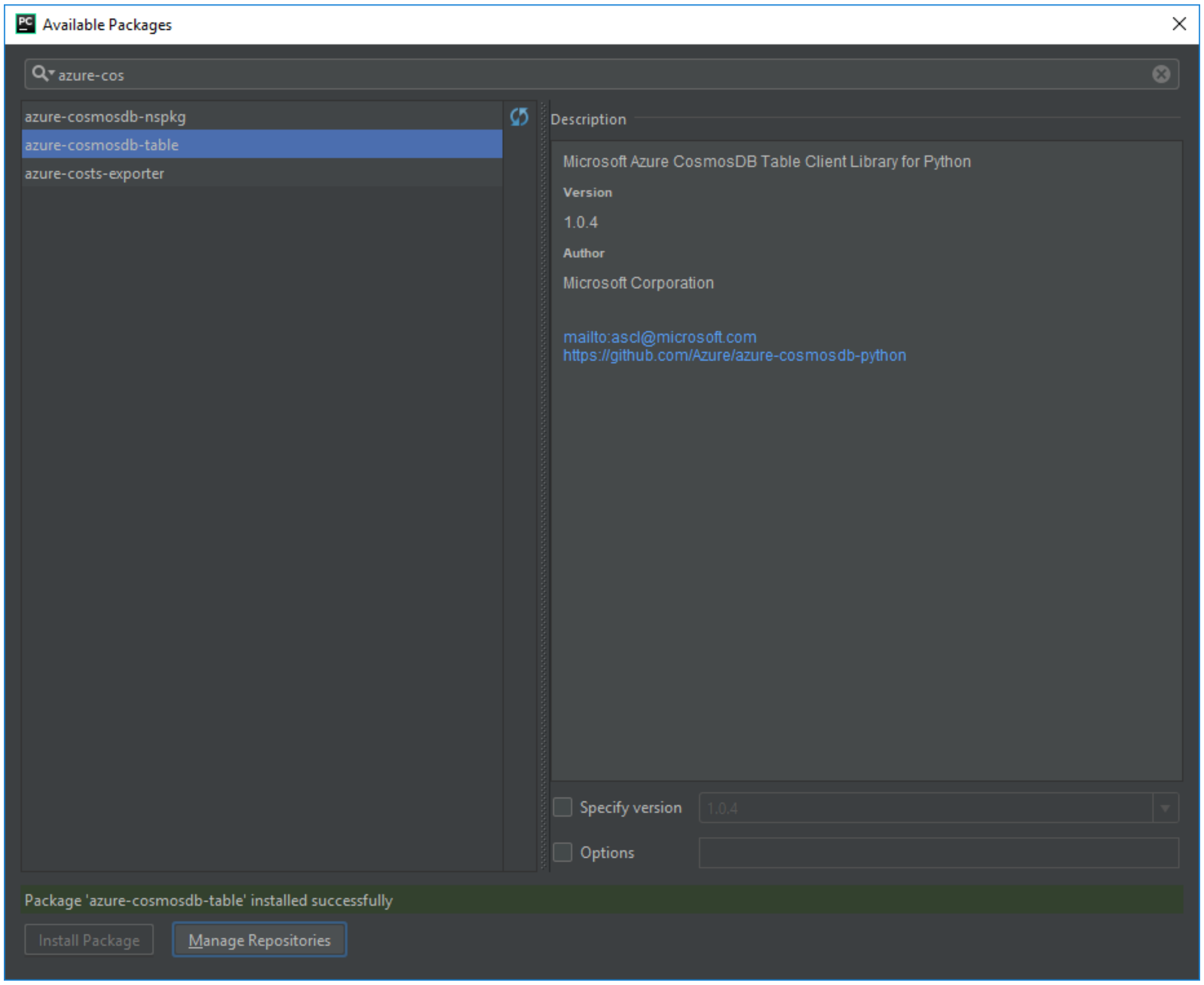
Inherit global site-packages [ ]

Make available to all projects [ ]

Existing interpreter

Interpreter: Python 3.6 C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\py

c. Click on *Create*.
d. Add a new file, addrow.py, to the project in the BatchDemo directory.
e. Add the library to use table storage.
    a. Select File→Settings→Project: BatchDemo→Project Interpreter.
    b. In the list of packages, you'll see a plus sign '+'. Click on that.
    c. Select azure-cosmosdb-table. Click on *Install Package*.
    d. Close the Available Packages window once done. My machine looked like this before I closed the window:


Available Packages

azure-cos

azure-cosmosdb-nspkg
azure-cosmosdb-table
azure-costs-exporter

Description

Microsoft Azure CosmosDB Table Client Library for Python

Version

1.0.4

Author

Microsoft Corporation

mailto:ascl@microsoft.com
https://github.com/Azure/azure-cosmosdb-python

Specify version 1.0.4

Options

Package 'azure-cosmosdb-table' installed successfully

f. Click OK on the Settings Window. You should now have a number of packages installed.

g. Add the following code to the *addrow.py* file. The code is fairly “Hello, world!”-ish: add a row to a Table Storage table (using libraries from the azure.cosmosdb namespace, but interacts with Storage, so no, not intuitive). The script is simple and adds one row to a table named *tasktable*:

```
from azure.cosmosdb.table.tableservice import TableService
import datetime

def main():
    table_name = 'tasktable'
    table_service = TableService(
        account_name="<your Azure Storage Account Name>",
        account_key="<your Azure Storage Account Key>")
    if not (table_service.exists(table_name)):
        table_service.create_table(table_name)
    task = \
        {
            'PartitionKey': 'tasks',
            'RowKey': str(datetime.datetime.utcnow()),
            'description': 'Do some task'
        }
    table_service.insert_entity(table_name, task)
```

```
if __name__ == "__main__":
    main()
```

For the highlighted code, take the name of one of your Azure Storage Accounts and a corresponding key, then plug in the proper values. If you need to create a storage account, instructions are here. To get the keys, look in the same doc (or click here) and follow the instructions. If you create a new storage account, use a new resource group and name the resource group *BatchPython*. We'll use that group name later too.

One last comment here: for a production app, you really should use Key Vault. The credentials are being handled this way to keep the concept count reasonably low.

h. Test the code by running it. You should be able to look at a table named *tasktable* in your storage account and see the new row. The RowKey is the current timestamp, so in our case it should provide for a unique enough key.

Once you have all this working and tested, let's look at how to run this simple thing in Azure Batch. Again, this is for learning how to do some simple stuff via a *Hello, World*.

2 Create a batch account

In this step, we are going to create a batch account which I'll refer to as *batch_account* in here; your name will be different. Just know to substitute a proper string where needed.

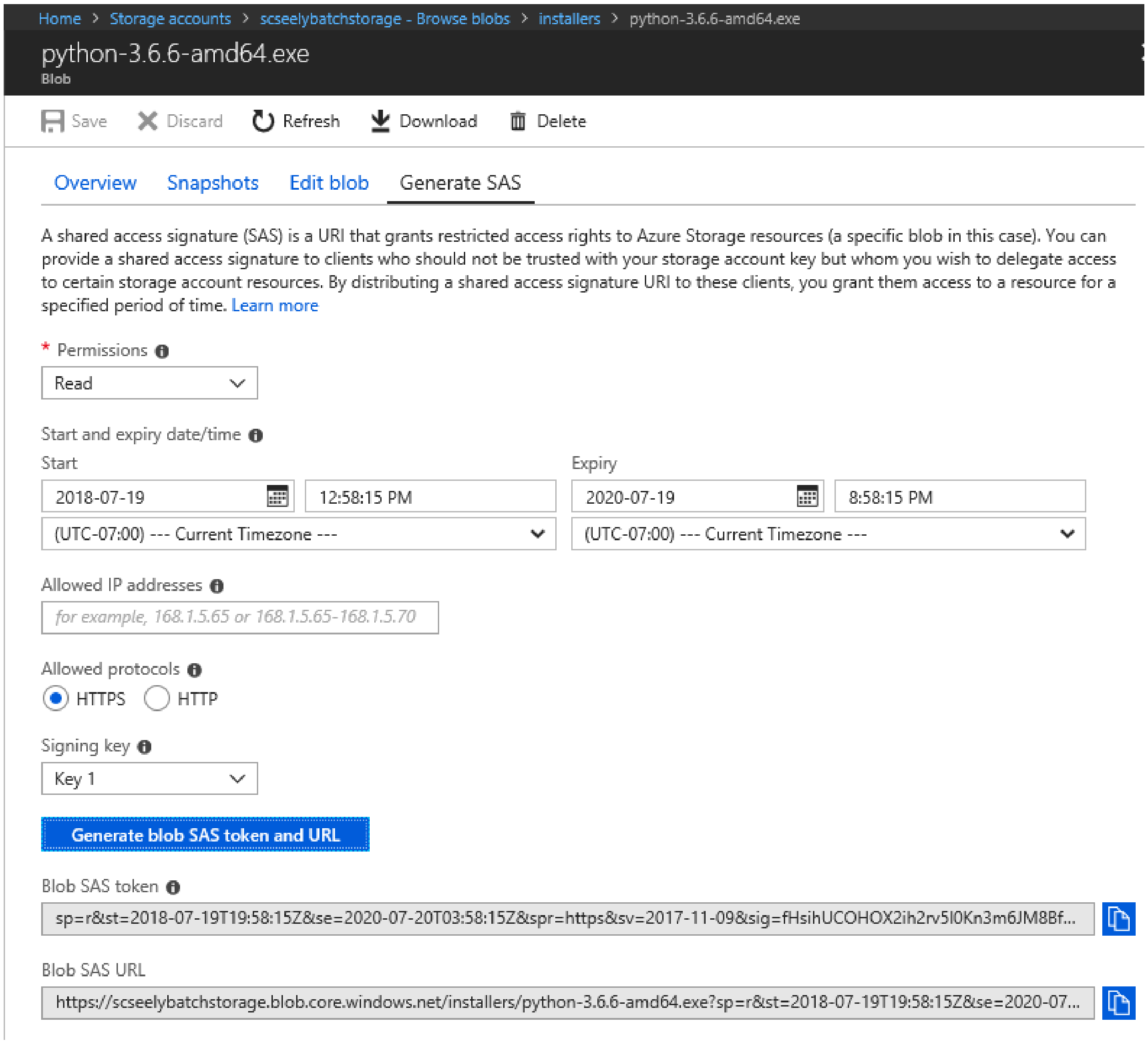
1. In the Azure portal, click on *Create a resource*.
2. Search for *Batch Service*. Click on *Batch Service*, published by *Microsoft*.
3. Click on *Create*.
   - Account name: For the account name, enter in *batch_account* [remember, this is a string you need to make up, then reuse. You're picking something unique. I used *scseelybatch*]
   - Resource Group: Select *BatchPython*. If you didn't create this earlier, select *Create New*.
   - Select a storage account to use with batch. You can use the same one you created to test the table insertion.
   - Leave the other defaults as is.
   - Click on *Create*.

3 Upload the Python Installer

Upload the Python installer which you want to use. I used the Windows x86-64 executable installer from here.

1. In your storage account, create a container named *installers*.
   a. In the Azure Portal, navigate to your Storage Account.
   b. Select Blob Service→Browse Blobs
   c. Click on *+ Container*.
   d. Set the Name to *installers*.
   e. Click on *OK*.

2. Once created, click on the *installers* container.
3. Upload the Python installer from your machine.
    a. Click on *Upload*.
    b. In the *Upload blob* screen, point to your installer and click on *Upload*.
    c. Wait until the upload completes.
4. Get a SAS URL for the installer.
    a. Right click on the uploaded file.
    b. Select *Generate SAS*.
    c. Set the expiration of the token to some time in the future. I went for 2 years in the future.
    d. Click on *Generate blob SAS token and URL*


Home > Storage accounts > scseelybatchstorage - Browse blobs > installers > python-3.6.6-amd64.exe
python-3.6.6-amd64.exe
Blob
Save
Discard
Refresh
Download
Delete
Overview
Snapshots
Edit blob
Generate SAS
Generate blob SAS token and URL

A shared access signature (SAS) is a URI that grants restricted access rights to Azure Storage resources (a specific blob in this case). You can provide a shared access signature to clients who should not be trusted with your storage account key but whom you wish to delegate access to certain storage account resources. By distributing a shared access signature URI to these clients, you grant them access to a resource for a specified period of time. Learn more

* Permissions: Read

Start and expiry date/time

Start: 2018-07-19 12:58:15 PM (UTC-07:00) --- Current Timezone ---

Expiry: 2020-07-19 8:58:15 PM (UTC-07:00) --- Current Timezone ---

Allowed IP addresses: for example, 168.1.5.65 or 168.1.5.65-168.1.5.70

Allowed protocols: [✓ HTTPS] HTTP

Signing key: Key 1

Blob SAS token

```
sp=r&st=2018-07-19T19:58:15Z&se=2020-07-20T03:58:15Z&spr=https&sv=2017-11-09&sig=fHsihUCOHOX2ih2rv5I0Kn3m6JM8Bf...
```

Blob SAS URL

```
https://scseelybatchstorage.blob.core.windows.net/installers/python-3.6.6-amd64.exe?sp=r&st=2018-07-19T19:58:15Z&se=2020-07...
```

    e. Copy the *Blob SAS URL*. Store that in a scratch area. You'll need it in a bit.

4 Create a Batch Application

1. Going back to your machine, go to the BatchDemo directory which contains your addrow.py file along with the virtual environment. Zip up BatchDemo and everything else inside into a file called *BatchDemo.zip*. [Mine is about 12MB in size]
2. Open up your list of Resource Groups in the portal. Click on *BatchPython*.
3. Click on your Batch account.
4. Select *Features*→*Applications*
5. Click on *Add*.
    a. Application id: BatchPython
    b. Version: 1.0
    c. Application package: Select BatchPython.zip
    d. Click on *OK*.

    The file will upload. When complete, you'll have 1/20 applications installed.

6. Click on *BatchPython*.
7. Set Default Version to 1.0.
8. Click on *Save*.

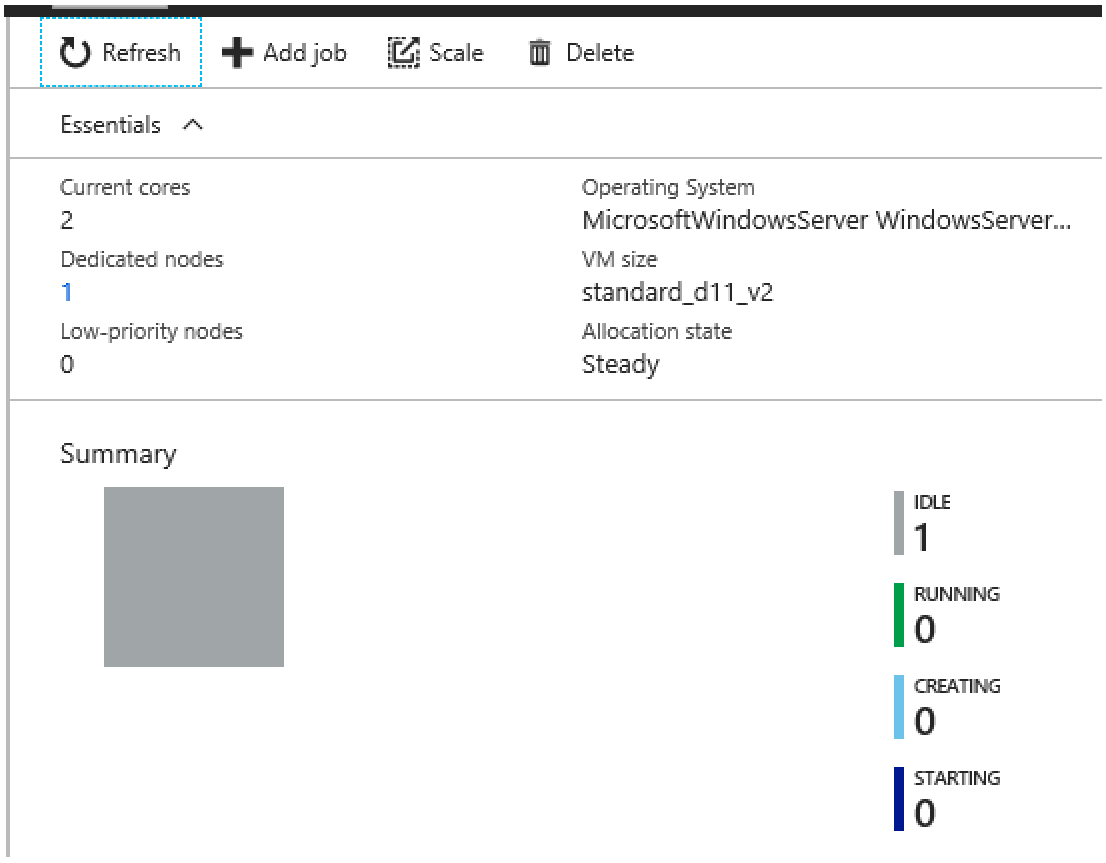
5 Create a Batch Pool

1. Open up your list of Resource Groups in the portal. Click on *BatchPython*.
2. Click on your Batch account.
3. Select on *Features*→*Pools*
4. Click on *Add*.
    a. Pool ID: Python
    b. Publisher: MicrosoftWindowsServer
    c. Offer: WindowsServer
    d. Sku: 2016-Datacenter
    e. Node pricing tier: Standard D11_v2 [Editorial: When experimenting, I prefer to pick nodes with at least 2 cores. 1 for the OS to do its thing, 1 for my code. I'll do one core for simple production stuff once I have things working. This is particularly important to allow for effective remote desktop/SSH. The extra core keeps the connection happy.]
    f. Target dedicated nodes: 1
    g. Start Task/Start task: Enabled
    h. Start Task/Command Line: We want this installed for all users, available on the Path environment variable, and we do not want a UI.
    python-3.6.6-amd64.exe /quiet InstallAllUsers=1 PrependPath=1 Include_test=0
    i. Start Task/User identity: Task autouser, Admin
    j. Start Task/Wait for success: True
    k. Start Task/Resource files:
        i. Blob Source: Same as the URL you saved from the piece labeled "Upload the Python Installer". The SAS token is necessary.
        ii. File Path: python-3.6.6-amd64.exe
        iii. Click on *Select*

l. Optional Settings/Application packages
    i. Click on Application Packages.
    ii. Application: BatchPython
    iii. Version: Use default version
    iv. Click on *Select*

m. Click on *OK*.

The pool will be created. In my experience, creating the pool and getting the node ready can take a few minutes. Wait until the node appears as *Idle* before continuing.



Current cores
2
Dedicated nodes
1
Low-priority nodes
0
Operating System
MicrosoftWindowsServer WindowsServer...
VM size
standard_d11_v2
Allocation state
Steady

Summary

IDLE
1
RUNNING
0
CREATING
0
STARTING
0

6 Run a job

1. Open up your list of Resource Groups in the [portal](). Click on *BatchPython*.
2. Click on your Batch account.
3. Select on *Features→Jobs*
4. Click on *Add*
    a. Job ID: AddARow
    b. Pool: Python
    c. Job manager, preparation, and release tasks:
        i. Mode: Custom
        ii. Job Manager Task:
            1. Task ID: AddARowTask
            2. Command line:

```
cmd /c %AZ_BATCH_APP_PACKAGE_BATCHPYTHON%\BatchDemo\venv\Scripts\python.exe
%AZ_BATCH_APP_PACKAGE_BATCHPYTHON%\BatchDemo\addrow.py
```

Note on the environment variable: Application packages are zip files. Batch puts the location of the unzipped application package into an environment variables in one of two ways, depending on if you select the default version or a specific version.

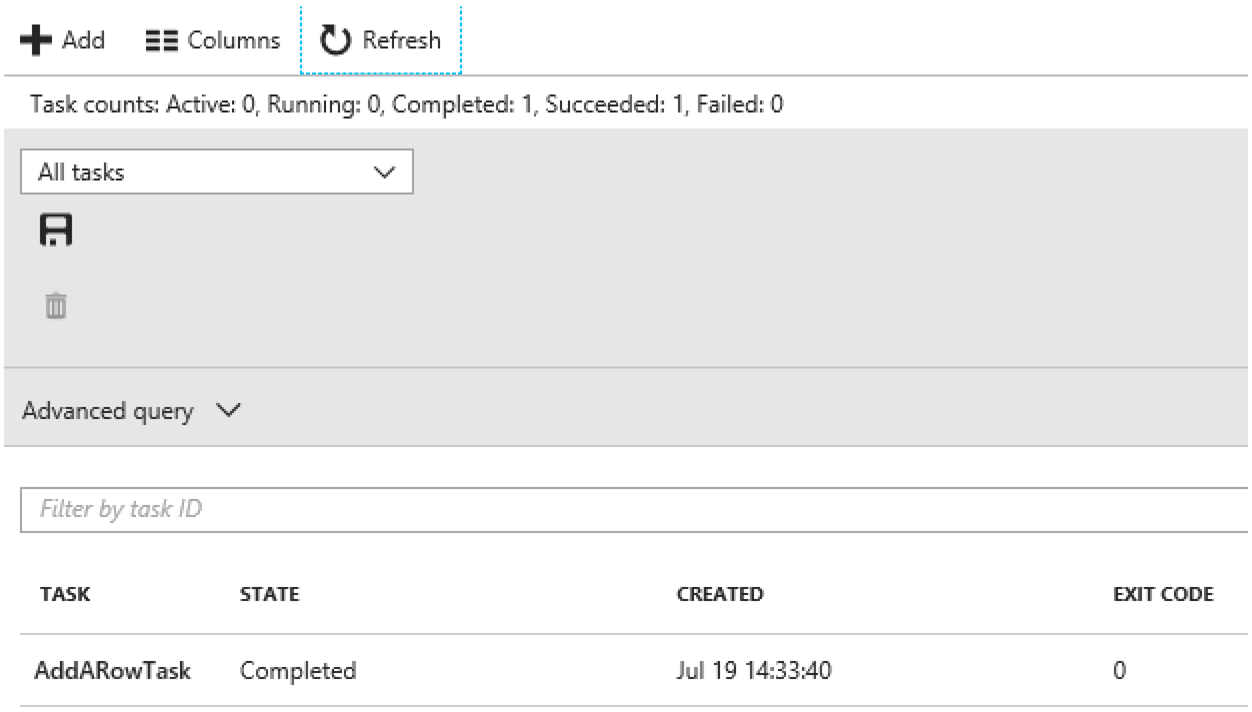
Default: AZ_BATCH_APP_PACKAGE_<Package name, upper case>

Versioned: AZ_BATCH_APP_PACKAGE_<Package name, upper case><Version number>

iii. Click on *Select*

d. Click on *OK*

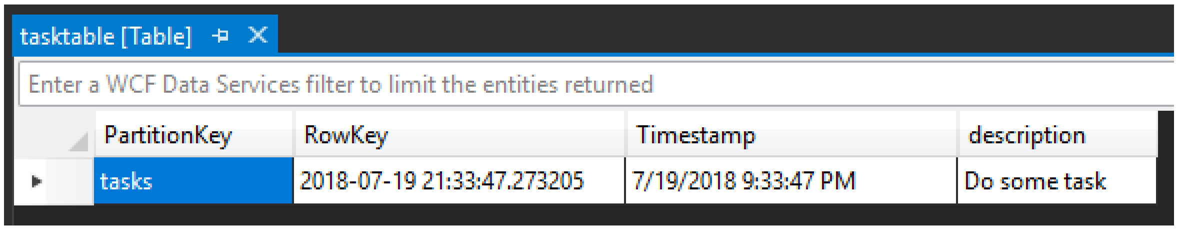
The job should start running immediately. Because it's a short task, it'll finish quickly too. Click on Refresh and you'll probably see that the AddARowTask has completed.



Task counts: Active: 0, Running: 0, Completed: 1, Succeeded: 1, Failed: 0

TASK	STATE	CREATED	EXIT CODE
AddARowTask	Completed	Jul 19 14:33:40	0

You can then verify the output by opening up the table and looking at the rows. A new one should be present. I'll expect a row that completed near 21:33 on July 19; the time will be recorded as UTC, and I'm in Redmond, WA, USA, which is 7 hours behind UTC time.


tasktable [Table]
Enter a WCF Data Services filter to limit the entities returned

PartitionKey	RowKey	Timestamp	description
tasks	2018-07-19 21:33:47.273205	7/19/2018 9:33:47 PM	Do some task

That view is courtesy of the Azure tools present in Visual Studio 2017.

7 So, what next?

Now that you've done all this, what does it mean? For your batch pools, you could preload them with a common toolset. The resource files you pass in to a job can be files to operate on, independent of binaries. Your tasks start times can be greatly reduced by loading the prerequisites early. Could you do this with custom VMs? Sure, but then you need to keep the VMs patched. This mechanism allows you to use a patched VM and just install your few items.

This is definitely a toy example, meant to show how to do the initial setup in the portal. Here's what you want to do for automation purposes:

1. Script all of this.
   a. For the Python piece, add a mechanism to create the zip file after you have a successful build and test.
   b. Script the management of the binaries, creating the Batch Service, and configuring the pools and application package(s).
2. Add an integration test to validate that the pool is ready to run.
3. Minimize the number of secrets in the code to 0 secrets. Use Key Vault to manage that stuff.