04 декабря 2021

Настройка фоновой загрузки файлов по протоколу WebDav без использования API в Azure Data Factory

Инфраструктуру Microsoft Azure используют во многих компаниях. Но не все разработчики понимают, как работать в ней максимально эффективно.

Меня зовут Алексей Жидков, я разработчик в ГК «КОРУС Консалтинг», и много работаю в облачной среде Microsoft Azure. Решил поделиться одним из кейсов, который поможет программистам обновлять большие объемы данных в Azure, когда нет возможности использовать API.

Предыстория: почему решили использовать WebDav

В одном из проектов нам приходиться ежедневно обновлять большое количество таблиц (parquet файлов), лежащих в Azure Data Lake Storage (ADLS). В этих файлах данные компании, с которой мы работаем: поставки, продукция, цены, остатки – чего только нет. Большая часть таблиц в их базе данных содержит от сотни миллионов строк. При этом в них не реализовано отслеживание изменений (даты обновления / версии строки) – данные просто затираются и заменяются на новые. А история изменений этих данных уже важна для компании.

Поэтому было решено организовать обновление данных следующим образом: мы получаем один раз полную версию таблиц клиента, а обновившиеся за день строки ежедневно выкладываются на WebDav-сервер. Мы добавляем к полным таблицам и этим обновленным строкам даты (загрузки на ADLS и выгрузки на WebDav, соответственно) и так реализуем историчность данных от момента загрузки полной версии файла.

Почему выбрали WebDav? У решения есть одно значительное преимущество: если фоновая загрузка данных по какой-то причине сломается, то можно скачать их вручную с помощью WinSCP. Этакий workaround, который можно использовать до устранения проблемы с фоновой загрузкой. Данный протокол сочетает в себе преимущества HTTP, которые не может предоставить FTP: надежная аутентификация, шифрование, поддержка прокси и кэширование. Что касается SFTP, он тоже хорош, как и WebDav, но мы используем последний, поскольку у компании, с которой мы работаем, такой сервер уже был установлен.

Но в процессе я выяснил, что в Azure Data Factory нет WebDav коннектора. Есть HTTP, FTP, SFTP и многое другое, но не WebDav.

Оказалось, что у WebDav есть множество провайдеров, которые реализуют еще большее количество стандартов. Плохая совместимость, никакой стандартизации! Я смирился с печальной действительностью и начал думать дальше.

Решил, что HTTP-коннектор тоже подойдет, ведь WebDav работает поверх HTTP-протокола. Осталось просто использовать API данного сервера и

вывести список содержащихся на нем файлов. Запросил у партнеров этот API и какую-нибудь инструкцию к нему. На что получил примерно такой ответ:

«API не наше, поэтому проконсультировать не можем. Вот вам имя хоста, логин и пароль для локального подключения»

Какие в этом случае есть варианты действий?

1.

Поплакать и уволиться. Очень соблазнительно, но непродуктивно, да и деньги нужны.

2.

Перелопатить интернет в поисках ответов, обсудить случай с коллегами и совместными усилиями прийти к неплохому решению. Так и поступили.

Решено было использовать Azure Function App, который выводил бы список содержимого на сервере в виде JSON. Собственно, этому способу и посвящен данный гайд.

Примечание: если у вас есть API нужного сервера WebDav лучше используйте именно его. Представленный ниже вариант нужен в том случае, когда API неизвестен.

Пошаговая инструкция по созданию Azure Function и ее использованию в Azure Data Factory для чайников

Шаг 1: заходим в Visual Studio Code, добавляем расширение Azure Functions, переходим в него и подключаемся.

Версия Visual Studio Code, насколько я понял, особой роли не играет. Поэтому, есть ли какая-то минимально требуемая, я вам не подскажу. Но лучше, естественно, не использовать слишком старые версии.

Шаг 2: в разделе FUNCTIONS нажимаем Create Function, выбираем HTTP-триггер и задаем для нашей функции желаемое название.

Шаг 3: заменяем сгенерированный скрипт на представленный ниже.

Здесь используется библиотека easywebdav, так как она подошла для нашего конкретного случая (провайдера). Если для вашего случая это не подходит, вы можете использовать другую библиотеку.

В данном примере подключение будет производиться по следующему адресу — https://your.server.ru/part1/part2/part3.

Скачиваемые файлы хранятся в папках с названиями таблиц, таким образом полный путь к ним выглядит следующим образом: https://your.server.ru/part1/part2/part3/path/name. Параметры path, name и create_time мы извлекаем для дальнейшего их использования в ADF Pipeline, остальные параметры – на ваше усмотрение.

Логин и пароль для подключения мы храним в Azure Key Vault, поэтому здесь используются секреты, взятые оттуда. В процессе тестирования вы можете ввести логин и пароль в сам скрипт, но в конце стоит брать из Key Vault. Инструкцию о том, как связать эти 2 сервиса вы можете прочитать по следующей ссылке.

```
import json
```

```
import logging

import os

import azure.functions as func

import easywebdav



PATH = 'part1/part2/part3/'

DESTINATION_URL = 'your.server.ru'



def main(req: func.HttpRequest) -> func.HttpResponse:

    logging.info('Python HTTP    trigger function processed a request.')



    _user = os.environ.get('USER_KV')

    _password = os.environ.get('PSWD_KV')



    try:

        webdav = easywebdav.connect(

            DESTINATION_URL,

            username = _user,
```

```
    password = _password,
    protocol = 'https',
    port = 443,
    verify_ssl = False
    )

folders = [f.name for f in webdav.ls(PATH)]

files=[]

for folder in folders[1:]:
    files.extend([{
        'path': f.name.split('/')[-2],
        'name': f.name.split('/')[-1],
        'size': f.size,
        'last_modified': f.mtime,
        'create_time': f.ctime
```

```
        } for f in webdav.ls(folder[folder.find(PATH):]) if f.name not in folders])


    response_body = json.dumps(files)


    return func.HttpResponse(
        body=response_body,
        status_code=200,
    )


except Exception as e:
    return func.HttpResponse(
        body=str(e),
    )
```

Шаг 4: добавляем в файл проекта requirements.txt используемую библиотеку и ее версию, после чего сохраняем проект. В данном случае в этот файл нужно добавить только библиотеку easywebdav, остальные используемые библиотеки уже включены в Azure Functions. Более того, попытка добавить в этот файл уже встроенные в Azure Functions библиотеки может выдать ошибку во время развертывания.

Шаг 5: заходим на портал Azure и создаем в желаемой группе ресурсов Function App. В Visual Studio Code это тоже можно сделать, но через портал надежнее – так меньше вероятность пропустить что-то важное, из-за чего это потом не будет работать.

Шаг 6: создаем для данного Function App соответствующий linked service.

Function App URL	URL - находим в созданном ранее Function App, на вкладке Overview
AKV linked service	Это название linked service вашего Key Vault, в котором хранятся логин и пароль для подключения к серверу WebDav
Secret name	Вам нужно перейти в созданный ранее Function App, на вкладку App keys. В разделе Host keys (all functions) отображаем значение для ключа «_master» и сохраняем его в Key Vault как секрет. Название секрета мы и указываем при создании linked service. Я его называю DataIntegrationFunctionKey.

Шаг 7: заливаем нашу функцию в созданный Function App и ждем окончания процесса.

С самым сложным разобрались. Теперь остается протестировать работу этой функции и настроить ADF Pipeline для загрузки файлов.

Шаг 8: создаем ADF Pipeline и добавляем в него activity Azure Function. Его задача – вывести список файлов на сервере, поэтому я назвал его просто List_of_Files.

Шаг 9: запускаем ADF Pipeline.

На выходе данной функции будет примерно следующий текст:


```
{

  "Response":

      "[

        {

          \"path\":  \"Table1\", \"name\": \"Table1_20211015.zip\",

          \"size\": 175,  \"last_modified\": \"Fri, 15 Oct 2021 00:05:07 GMT\",  \"create_time
\": \"2021-10-15T00:05:07.609Z\"

        },

        {

          \"path\":  \"Table1\", \"name\": \"Table1_20211014.zip\",

          \"size\": 175,  \"last_modified\": \"Fri, 14 Oct 2021 00:05:04 GMT\",  \"create_time
\": \"2021-10-14T00:05:04.522Z\"

        },

        {

          \"path\":  \"Table2\", \"name\": \"Table2_20211015.zip\",

          \"size\": 175,  \"last_modified\": \"Sat, 15 Oct 2021 00:05:02 GMT\",  \"create_time
\": \"2021-10-15T00:05:02.163Z\"
```

```
        },
        {
          \"path\":  \"Table2\", \"name\": \"Table2_20211014.zip\",
          \"size\": 175,  \"last_modified\": \"Sat, 14 Oct 2021 00:05:04 GMT\",  \"create_time\": \"2021-10-14T00:05:04.253Z\"
        },
        {
          \"path\":  \"Table3\", \"name\": \"Table3_20211015.zip\",
          \"size\": 175,  \"last_modified\": \"Sun, 15 Oct 2021 00:05:06 GMT\",  \"create_time\": \"2021-10-15T00:05:06.265Z\"
        },
        {
          \"path\":  \"Table3\", \"name\": \"Table3_20211014.zip\",
          \"size\": 175,  \"last_modified\": \"Sun, 14 Oct 2021 00:05:03 GMT\",  \"create_time\": \"2021-10-14T00:05:03.479Z\"
        }
      ]",
  "effectiveIntegrationRuntime": "DefaultIntegrationRuntime",
  "executionDuration": 43,
```

```
"durationInQueue":
        {
    "integrationRuntimeQueue": 0
},
 "billingReference":
        {
    "activityType": "ExternalActivity",
    "billableDuration":
                [
        {
            "meterType": "AzureIR",
            "duration": 0.016666666666666666,
            "unit": "Hours"
        }
    ]
}
}
```

Шаг 10: добавляем к выходу данной функции в ADF Pipeline фильтр по датам. Будем скачивать файлы только за последний день.

Items	@json(activity('List_of_Files').output.Response)
Condition	@greaterOrEquals(item().create_time, getPastTime(1, 'Day'))

Если вам не нравится функция getPastTime() создайте и используйте переменные DATE_FROM / DATE_TO.

Шаг 11: добавляем фильтр по имени каждой загружаемой таблицы. Это нужно, чтобы в дальнейшем для каждой из них можно было задать разные настройки. Например, сохранять их в разные хранилища, изменять для некоторых таблиц формат файлов или задавать маппинг.

Items	@activity('Select_only_new_files').output.Value
Condition	@startswith(item().name, 'Table1_')

Шаг 12: добавляем ForEach activity на выход каждого фильтра. Именно внутри этого блока мы задаем действия, которые будут совершены с загружаемыми

файлами. В нашем случае мы просто задаем для каждой таблицы типы данных и сохраняем их в формате parquet.

Шаг 13: в каждый ForEach добавляем copy activity. Для всех файлов будет достаточно создать два датасета – для Source (источник) и Sink (приемник).

Наш приемник – это WebDav (HTTP) сервер, а источник – Azure Data Lake Storage (а именно папка, которая будет содержать в себе папки Table1, Table2 и Table3).

Шаг 14: создаем Source датасет.

Для этого создаем HTTP linked service.

Также для Source датасета добавляем 2 параметра: FOLDER и NAME. Их мы используем в Relative URL.

Relative URL	@{dataset().FOLDER}/@{dataset().NAME}

Передаем выходные параметры Azure Function path и name на вход copy activity.

Так как наши файлы – это zip-архивы на вкладке Source, мы можем поставить галочку в пункте «Preserve zip file name as folder». Это позволяет нам сохранить ту же структуру папок, что и на WebDav-сервере.

В нашем случае мы добавляем в Source еще и Additional column – дату добавления/обновления строки.

На 10 шаге вы могли заметить, что имена наших файлов содержат в себе не только название таблицы, но и дату загрузки. Эта дата может отличаться от

даты создания файла (create_time) на несколько минут, поэтому мы используем дату из названия файла, чтобы проще было найти нужный файл.

Эту дату мы берем из параметра NAME, а вы можете создать и использовать для этого параметр CREATE_TIME.

Шаг 15: создаем Sink датасет.

Тут все просто – указываем адрес к папке, в которой будут храниться файлы со всеми нашими таблицами. Никаких параметров создавать и использовать не нужно.

Шаг 16: настраиваем mapping (если нужно) и повторяем это для copy activity в каждом ForEach.

Дело сделано, вы великолепны.

На мой взгляд, этот способ довольно универсален – его можно использовать для разных стандартов WebDav практически без изменений в коде функции. Это решение можно использовать в качестве шаблона – в структуре ADF Pipeline также почти не будет изменений. Построенный ADF Pipeline даже без использования API работает быстро и без проблем.

Буду рад, если это кому-нибудь пригодится.