Documentation for Incremental Data Extraction from an API Endpoint

Purpose

This script is intended for incremental data extraction from an API endpoint. It compares the data from the API endpoint to data from a local file and updates the file with any changes in the data since the last time it was run.

Prerequisites

- The script requires a valid API endpoint URL and the local file name as input parameters.
- The script assumes that the local file is a JSON file and contains the same data structure as the data returned by the API endpoint.
- The script requires the `Invoke-RestMethod` cmdlet to retrieve data from the API endpoint and the `Get-FileHash` cmdlet to calculate the hash values of the data. These cmdlets are part of the PowerShell core and should be available in any modern version of PowerShell.

Inputs

- `$apiEndpoint`: A string that specifies the URL of the API endpoint. It should start with `http://` or `https://`.
- `$localFileName`: A string that specifies the name of the local file (without the file extension). The script assumes that the file is located in the current directory.
- `$algorithm`: A string that specifies the hashing algorithm to use for calculating the hash values. The following algorithms are supported: MD5, SHA1, SHA256, SHA384, SHA512.

Outputs

- The script outputs a message indicating whether the data from the API endpoint and the local file are equal or not.
- If the data is not equal, the script outputs the changed data in a table format and updates the local file with the changed data.

Usage

```
@echo off

set cwd=%CD%
set apiEndpoint="https://api.coindesk.com/v1/bpi/currentprice.json"
set localFileName="bpiUSD"
set algorithm="SHA256"

pwsh.exe -File "%cwd%\Extract-IncrementalDatav1.2.ps1" -apiEndpoint %apiEndpoint%
-localFileName %localFileName% -algorithm %algorithm%
```

```
timeout 10
```

Notes

- The `Get-DataHash` function is intended to calculate the hash value of data retrieved from a specified data source, which can be either an API endpoint or a file. The function takes two mandatory parameters, the data source and the algorithm to be used to calculate the hash value. The data source must be a string value that either represents an API endpoint starting with "http" or "https" or a file path. The algorithm parameter must be a string value that specifies the hashing algorithm to be used. It must be one of the following values: "MD5", "SHA1", "SHA256", "SHA384", "SHA512".
  The function first checks whether the data source is an API endpoint or a file. If it is an API endpoint, it uses the `Invoke-RestMethod` cmdlet to retrieve the data from the endpoint and convert it to a JSON string. It then converts the JSON string to a byte array using the UTF8 encoding method. If the data source is a file, the function reads the contents of the file into a string and converts it to a JSON string. It then converts the JSON string to a byte array using the UTF8 encoding method.
  The function then creates a new MemoryStream object with the capacity of the byte array and writes the byte array to the object. It uses the `Get-FileHash` cmdlet to calculate the hash value of the data streaming object using the specified algorithm and returns the calculated hash value.

- The script compares the hash values of the data from the API endpoint and the data from the file. If the hash values are not equal, it compares the reference data with the difference data using the `Compare-Object` cmdlet to determine which data has changed and displays the changed data. The script compares data based on the 'code', 'symbol', 'rate', 'description', and 'rate_float' properties.

- The script updates the local CSV file with the changed data from the API endpoint and replaces the local JSON file with the API JSON data.

Potential Performance Issues

1. **Network performance:** The script retrieves data from an API endpoint using the `Invoke-RestMethod` cmdlet. If the network connection is slow or the API endpoint is experiencing high traffic, it could take a long time to retrieve the data.

2. **Memory usage:** The script converts the data from the API endpoint and the local file to JSON strings and then to byte arrays. If the data is large, it could consume a significant amount of memory, which could lead to performance issues.

3. **File I/O performance:** The script reads the contents of a local file into memory and writes to the file multiple times. If the file is large or the disk is slow, it could take a long time to read and write the data, which could impact performance.

4. **Hash calculation performance:** The script calculates the hash value of the data from the API endpoint and the local file using the `Get-FileHash` cmdlet. Depending on the size of the data and the chosen hashing algorithm, this could take a long time and impact performance.

5. **Data comparison performance:** The script uses the `Compare-Object` cmdlet to compare the data from the API endpoint and the local file and determine which data has changed. If the data is large, this

could take a long time and impact performance.

Improvement Suggestions

Here are a few suggestions that might help to improve the performance of the script:

1. **Caching:** One way to improve performance is to cache the data that is being fetched from the API endpoint. This can be done by storing the data in a variable or in a temporary file, and checking if the data has changed before making a new request to the API. This will help to reduce the number of requests being made to the API, which can be resource-intensive.

2. **Asynchronous requests:** Instead of using the `Invoke-RestMethod` cmdlet, you can use the `Invoke-WebRequest` cmdlet, which allows you to make asynchronous web requests. This can improve the performance of the script, especially if the API endpoint is slow or the network connection is not reliable.

```
# Data source is an API endpoint
# Use Invoke-WebRequest to retrieve the data from the API
try {
    $response = Invoke-WebRequest -Uri $DataSource -Method Get -
UseBasicParsing
    $data = $response.Content | ConvertFrom-Json
}
catch {
    Write-Host "Program failed to retrieve data at the API endpoint"
}
```

3. **Optimize the data processing:** The script converts the data from the API endpoint to a JSON string and then to a byte array, and then it creates a MemoryStream object with the byte array and calculates the hash value of the data. This can be a resource-intensive process, especially if the data being fetched from the API is large. To optimize the data processing, consider using a streaming approach, where the data is processed in chunks rather than all at once. This can help to reduce the amount of memory being used by the script. For example, in the following line of code:

```
$dataStream = (Invoke-RestMethod -Uri $DataSource -Method Get -OutFile
$null).bpi.USD
```

The `Invoke-RestMethod` cmdlet sends a `GET` request to the specified API endpoint and returns the response to the `$dataStream` variable. The `-OutFile $null` parameter specifies that the cmdlet should not create a file with the response.

This approach allows you to process the data in a streaming fashion, rather than reading and storing the entire response in memory before processing it. This can be more efficient, especially if the response is large or if you need to process the data as it is received. By using `-OutFile $null`, you can avoid creating temporary files to store the output and reduce the overhead of writing and reading from the file system.

4. **Use a faster hashing algorithm:** If the performance of the script is still an issue, you may want to consider using a faster hashing algorithm. For example, the MD5 algorithm is generally faster than the SHA256 algorithm, so using the MD5 algorithm might help to improve the performance of the script.

   > *For security reasons, MD5 and SHA1, which are no longer considered secure, should only be used for simple change validation, and should not be used to generate hash values for files that require protection from attack or tampering.*

   > *Performance-wise, a SHA-256 hash is about 20-30% slower to calculate than either MD5 or SHA-1 hashes.*

5. **Use parallel processing:** If the data being fetched from the API is very large, you may want to consider using parallel processing to improve the performance of the script. This can be done using the `Start-Job` cmdlet or the `System.Threading.Tasks.Parallel` class. Parallel processing allows you to run multiple tasks simultaneously, which can significantly improve the overall speed of the script.

```
# Start jobs to calculate hash values in parallel
$apiJob = Start-Job -ScriptBlock {
    & ".\Calculate-DataHash.ps1" -DataSource
"https://api.coindesk.com/v1/bpi/currentprice.json" -Algorithm SHA256
    }
$fileJob = Start-Job -ScriptBlock {
    & ".\Calculate-DataHash.ps1" -DataSource ".\bpiUSD.json" -Algorithm
SHA256
    }

# Wait for the jobs to complete
Wait-Job -Job $apiJob, $fileJob

# Get the results of the jobs
$apiDataArray = Receive-Job -Job $apiJob
$fileDataArray = Receive-Job -Job $fileJob

# Remove the jobs from the session
Remove-Job -Job $apiJob, $fileJob
```

   > This will start two separate jobs in parallel, --one for the API endpoint data and one for the local file data-- and wait for both of them to complete before retrieving the results and removing the job objects.

6. **Use Hashing Algorithm for data comparison:** One approach you could take to improve the performance of the script is to use a `MD5` hashing algorithm for comparing the data from the API endpoint and the local file, instead of using the `Compare-Object` cmdlet. This would allow you to quickly determine whether the data has changed without having to compare *each property* of the data objects.

```
# Calculate the hash values for the data from the API endpoint and the local
file
$apiDataArray = Get-DataHash -DataSource $apiEndpoint -Algorithm $algorithm
```

```
$fileDataArray = Get-DataHash -DataSource .\$localFileName.json -Algorithm
$algorithm
# Compare the hash values
if ($apiDataArray[0] -ne $fileDataArray[0]){

    Write-Host "Hash values not equal"

    # If the hash values are not equal, update the local file with the data
from the API endpoint
    $apiDataArray[2] | Out-File .\$localFileName.json
}
```

> Using a hashing algorithm to compare the data would be faster than using the `Compare-Object` cmdlet, especially if the data sets are large. However, it should be noted that using a hashing algorithm to compare the data will not provide the same level of detail as the `Compare-Object` cmdlet, as it will not show the specific differences between the data sets. It will only indicate whether the data has changed or not. This means that the script will not be performing incremental data extraction anymore. To perform incremental data extraction, you need a way to identify and process *only the changed data*.

7. **Reduce file I/O calls**: Using the `Where-Object` cmdlet to filter the results of the `Compare-Object` cmdlet can improve performance by reducing the amount of data that needs to be processed. This is preferrable to the ForEach-Object loop because it cuts the number of I/O calls.

```
# If the hash values are not equal, compare the ref data with the diff data
to determine which data has changed
$changedData = Compare-Object $apiDataArray[1] $fileDataArray[1] -Property
code, symbol, rate, description, rate_float -Passthru `
    | Where-Object {$_.SideIndicator -eq "<="}

# Display the changed data
$changedData | Format-Table

Write-Host "Updating CSV and JSON files..."

# Update CSV document with changed data from API endpoint
$changedData | Export-Csv -Path "C:\Users\ENGR. RICH\Downloads\bpiUSD.csv" -
NoTypeInformation -Append
```

References

- Get-FileHash
- Compare-Object
- Invoke-RestMethod
- Passing an array of bytes to system.IO.MemoryStream
- OpenAI

Supplementary

Justification for review of incremental data extraction in PIMS Microservice

The current idea of incremental data extraction implemented in the batch program passes a 24 hr interval date filter to the body of the HTTP API request. This is problematic for the following reasons:

1. **Duplicates**
   Data created at 6pm would be extracted twice based on the 24hr *inclusive* filter. For example, data extracted between 6pm monday and 6pm Tuesday, and data extracted between 6pm Tuesday and 6pm Wednesday; if data is created 6pm on Tuesday, it would be extracted twice.
2. **Possible data loss**
   If the program fails to execute at 6pm *exactly*, data created within the runtime delay period is lost. For example, if there's a 5 min delay at trigger time of 6pm, data within 6pm and 6:05pm is lost.

To solve this problem, we have to compare the data previously extracted with the incoming data. We only process data that changed in the source. That's the idea of incremental data extraction or incremental data pipelines.

Plan for PIMS Microservice Update

- Develop separate scripts for incremental data extraction and full batch data extraction
- Modify the batch script to call these scripts based run condition, that is, first run or subsequent runs.

End of documentation