

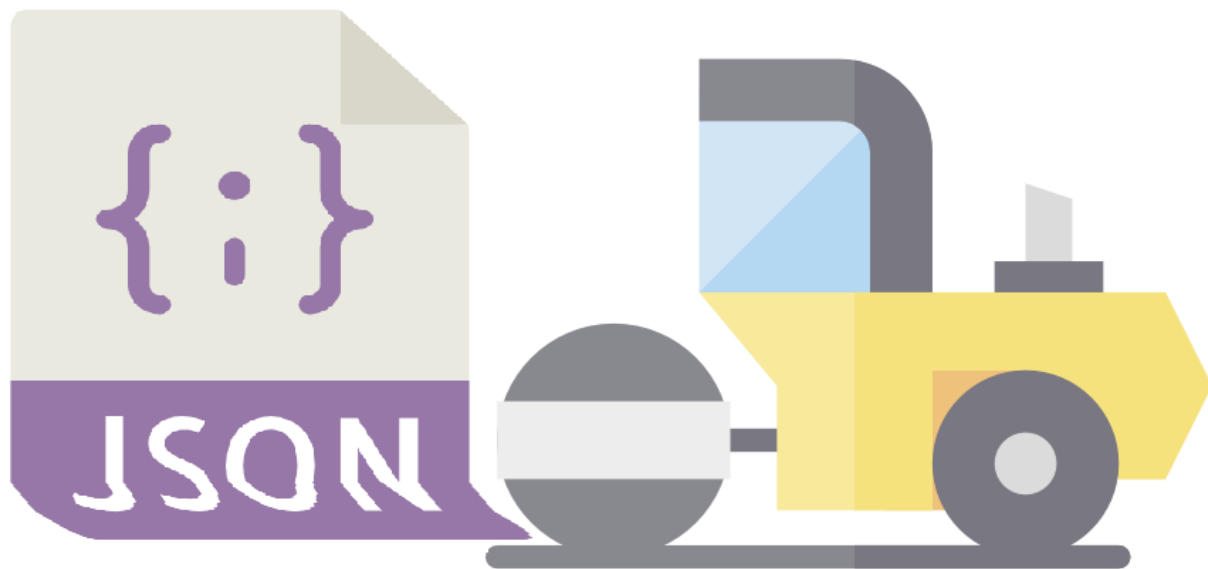
[Get started](#)[Open in app](#)

## Gary Strange

54 Followers · About

[Follow](#)

# Flattening JSON in Azure Data Factory

[Gary Strange](#) May 7, 2019 · 8 min read

JSON is a common data format for message exchange. Its popularity has seen it become the primary format for modern micro-service APIs. JSON allows data to be expressed as a graph/hierarchy of related information, including nested entities and object arrays. It benefits from its simple structure which allows for relatively simple direct serialization/deserialization to class-orientated languages.

Many enterprises maintain a BI/MI facility with some sort of Data warehouse at the beating heart of the analytics platform. Typically Data warehouse technologies apply schema on write and store data in tabular tables/dimensions. When ingesting data into the enterprise analytics platform, data engineers need to be able to source data from domain end-points emitting JSON messages. Messages that are formatted in a way that makes a lot of sense for message exchange (JSON) but gives ETL/ELT developers a problem to solve. How to transform a graph of data into a tabular representation.

There are many methods for performing JSON flattening but in this article, we will take a look at how one might use ADF to accomplish this. Microsoft currently supports two versions of ADF, v1 and v2. The content here refers explicitly to ADF v2 so please consider all references to ADF as references to ADF v2.

I'll be using Azure Data Lake Storage Gen 1 to store JSON source files and parquet as my output format. Although the storage technology could easily be Azure Data Lake Storage Gen 2 or blob or any other technology that ADF can connect to using its JSON parser. Again the output format doesn't have to be parquet. But I'm using parquet as it's a popular big data format consumable by spark and SQL polybase amongst others.

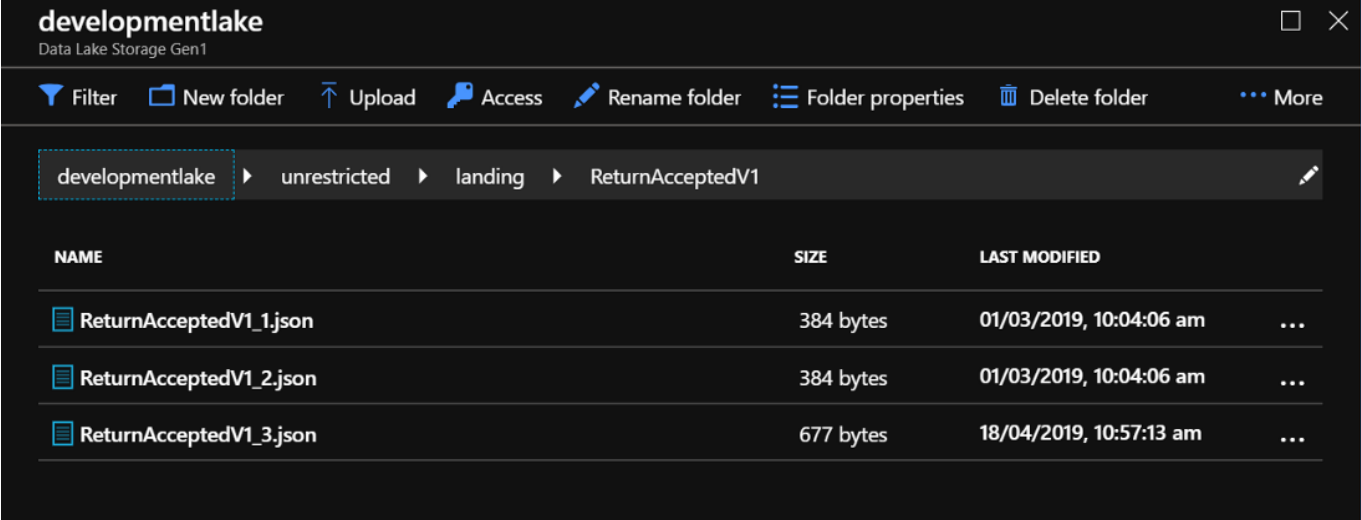
My test files for this exercise mock the output from an e-commerce returns micro-service.

Here is an example of the JSON:

```
{
  "returnReference": "4829U3N35MKR9E",
  "orderReference": "02KY5KL1Q21S",
  "customerId": 201582644,
  "items": [
    {
      "variantId": 613941,
      "quantity": 1,
      "returnReason": {
        "code": 1,
        "notes": "Too Large"
      }
    },
    {
      "variantId": 713941,
      "quantity": 5,
      "returnReason": {
```

```
    "code": 2,  
    "notes": "Doesn't suit me"  
  }  
},  
"timestamp": "2017-12-28T13:51:46.2671616Z"  
}
```

In the JSON structure, we can see a customer has returned two items. Those items are defined as an array within the JSON. This file along with a few other samples are stored in my development data-lake.



The screenshot shows the Azure Data Lake Storage Gen1 web interface. The breadcrumb path is 'developmentlake' > 'unrestricted' > 'landing' > 'ReturnAcceptedV1'. Below the path, there is a table with three columns: NAME, SIZE, and LAST MODIFIED. The table lists three files: 'ReturnAcceptedV1\_1.json' (384 bytes, 01/03/2019, 10:04:06 am), 'ReturnAcceptedV1\_2.json' (384 bytes, 01/03/2019, 10:04:06 am), and 'ReturnAcceptedV1\_3.json' (677 bytes, 18/04/2019, 10:57:13 am). Each file has a three-dot menu icon to its right.

NAME	SIZE	LAST MODIFIED
ReturnAcceptedV1_1.json	384 bytes	01/03/2019, 10:04:06 am
ReturnAcceptedV1_2.json	384 bytes	01/03/2019, 10:04:06 am
ReturnAcceptedV1_3.json	677 bytes	18/04/2019, 10:57:13 am

Azure Data Lake Gen 1

So we have some sample data, let's get on with flattening it. My ADF pipeline needs access to the files on the Lake, this is done by first granting my ADF permission to read from the lake. I used [Manage Identities](#) to allow ADF to have access to files on the lake.

For those readers that aren't familiar with setting up Azure Data Lake Storage Gen 1 I've included some guidance at the end of this article. I'm going to skip right ahead to creating the ADF pipeline and assume that most readers are either already familiar with Azure Datalake Storage setup or are not interested as they're typically sourcing JSON from another storage technology.

## ADF Pipeline

First off, I'll need an Azure DataLake Store Gen1 linked service. I set mine up using the Wizard in the ADF workspace which is fairly straight forward. Don't forget to test the

connection and make sure ADF and the source can talk to each other. If you hit some snags the Appendix at the end of the article may give you some pointers.

#### Data Lake Store selection method

☒ From Azure subscription

☐ Enter manually

##### Azure subscription

Visual Studio Enterprise (51dbb16f-c0c8-41e3-aea1-93f6e8573e73)

##### Data Lake Store account name \*

developmentlake

##### Tenant \*

4af8322c-80ee-4819-a9ce-863d5afbea1c

##### Authentication type \*

Managed Identity

Managed identity application ID: 4869de94-3c03-46d8-9534-0923d122a91c

Grant data factory managed identity access to your Azure Data Lake Storage Gen1. [Details](#)

#### Annotations

+ New

► Advanced ⓘ

Cancel

Test connection

Finish


## JSON Source Dataset


Now for the bit of the pipeline that will define how the JSON is flattened. Add an Azure Data Lake Storage Gen1 Dataset to the pipeline. Alter the name and select the Azure Data Lake linked-service in the connection tab.

Next, select the file path where the files you want to process live on the Lake.

Linked service \*

 AzureDataLakeStoreGen1

 Test connection

 Edit

+ New

File path

unrestricted/landing/ReturnAcceptedV1 / File

 Browse

Compression type

None

Filter by last modified

Start time (UTC)

End time (UTC)

To configure the JSON source select 'JSON format' from the file format drop down and 'Set of objects' from the file pattern drop down.

Hit the 'Parse JSON Path' button this will take a peek at the JSON files and infer it's structure. It's worth noting that as far as I know only the first JSON file is considered. So you need to ensure that all the attributes you want to process are present in the first file.

To explode the item array in the source structure type 'items' into the 'Cross-apply nested JSON array' field. This will add the attributes nested inside the items array as additional column to JSON Path Expression pairs.

JSON path settings

Cross-apply nested JSON array

☐ Edit

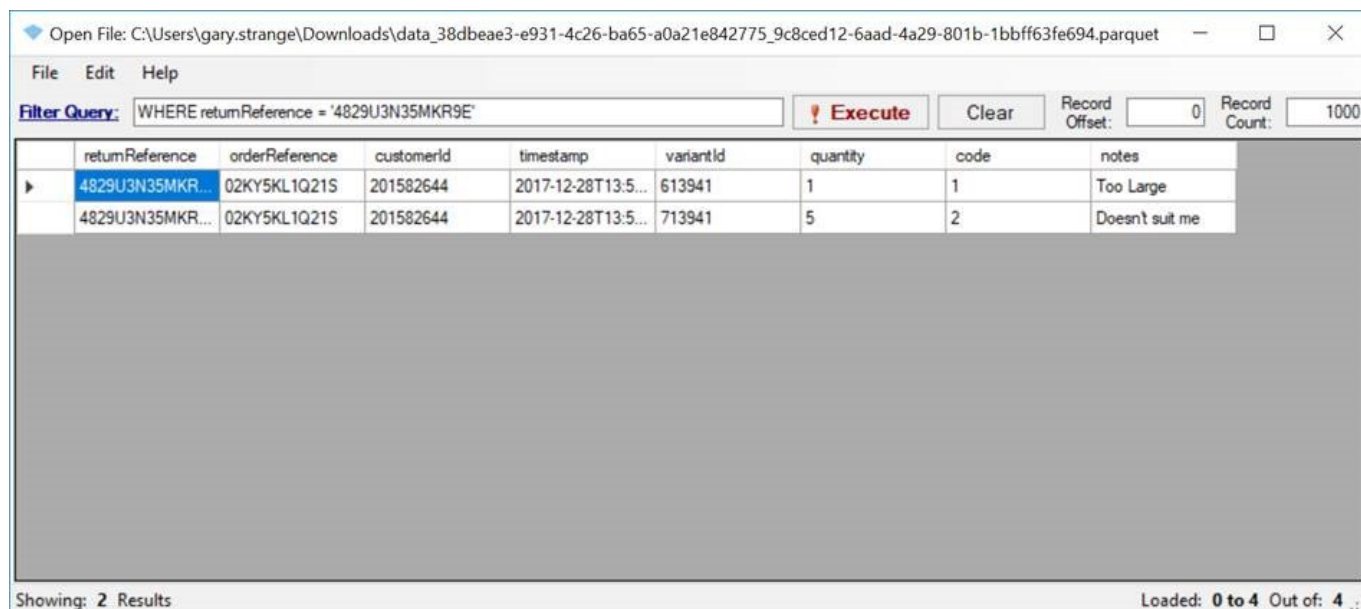
COLUMN NAME	JSONPATH EXPRESSION
returnReference	\$.['returnReference']
orderReference	\$.['orderReference']
customerId	\$.['customerId']
items	\$.['items']
timestamp	\$.['timestamp']

All that's left to do now is bin the original 'items' mapping. If left in, ADF will output the original 'items' structure as a string. Including escape characters for nested double quotes. Although the escaping characters are not visible when you inspect the data with the 'Preview data' button.

customerid	\$.['customerid']
items	\$.['items']
timestamp	\$.['timestamp']
variantId	['variantId']
quantity	['quantity']
code	['returnReason']['code']
notes	['returnReason']['notes']

This is the bulk of the work done. All that's left is to hook the dataset up to a 'copy activity' and sync the data out to a destination dataset.

I sent my output to a parquet file. The flattened output parquet looks like this...



	returnReference	orderReference	customerId	timestamp	variantId	quantity	code	notes
▶	4829U3N35MKR...	02KY5KL1Q21S	201582644	2017-12-28T13:5...	613941	1	1	Too Large
	4829U3N35MKR...	02KY5KL1Q21S	201582644	2017-12-28T13:5...	713941	5	2	Doesn't suit me

I'm using an open source [parquet viewer](#) I found to observe the output file. The input JSON document had two elements in the items array which have now been flattened out into two records.

I was able to create flattened parquet from JSON with very little engineer effort. That makes me a happy data engineer.

## Not So Good

I got super excited when I discovered that ADF could use JSON Path expressions to work with JSON data. Sure enough in just a few minutes, I had a working pipeline that was able to flatten simple JSON structures. However, as soon as I tried experimenting with more complex JSON structures I soon sobered up. It's certainly not possible to extract data from multiple arrays using cross-apply. Part of me can understand that running two or more cross-applies on a dataset might not be a grand idea. But I'd still like the option to do something a bit nutty with my data. I tried a possible workaround. What would happen if I used cross-apply on the first array, wrote all the data back out to JSON and

then read it back in again to make a second cross-apply? This isn't possible as the ADF copy activity doesn't actually support nested JSON as an output type. JSON structures are converted to string literals with escaping slashes on all the double quotes. So when I try to read the JSON back in, the nested elements are processed as string literals and JSON path expressions will fail.

Here is an example of the input JSON I used. You'll see that I've added a carrierCodes array to the elements in the items array.

```
{
  "returnReference": "4829U3N35MKR9E",
  "orderReference": "02KY5KL1Q21S",
  "customerId": 201582644,
  "items": [
    {
      "variantId": 613941,
      "quantity": 1,
      "returnReason": {
        "code": 1,
        "notes": "Too Large"
      },
      "carrierCodes": [ "20000K", "20005K", "40000K" ]
    }, {
      "variantId": 713941,
      "quantity": 5,
      "returnReason": {
        "code": 2,
        "notes": "Doesn't suit me"
      },
      "carrierCodes": [ "20000K", "20005K", "40000K" ]
    }
  ],
  "timestamp": "2017-12-28T13:51:46.2671616Z"
}
```

As mentioned if I make a cross-apply on the items array and write a new JSON file, the 'carrierCodes' array is handled as a string with escaped quotes.

```
{
  "returnReference": "4829U3N35MKR9E",
  "orderReference": "02KY5KL1Q21S",
  "customerId": 201582644,
  "timestamp": "2017-12-28T13:51:46.2671616Z",
  "variantId": 613941,
```

```

    "quantity": 1,
    "code": 1,
    "notes": "Too Large",
    "carrierCodes": "[\"20000K\", \"20005K\", \"40000K\"]"
  }, {
    "returnReference": "4829U3N35MKR9E",
    "orderReference": "02KY5KL1Q21S",
    "customerId": 201582644,
    "timestamp": "2017-12-28T13:51:46.2671616Z",
    "variantId": 713941,
    "quantity": 5,
    "code": 2,
    "notes": "Doesn't suit me",
    "carrierCodes": "[\"20000K\", \"20005K\", \"40000K\"]"
  }
}

```

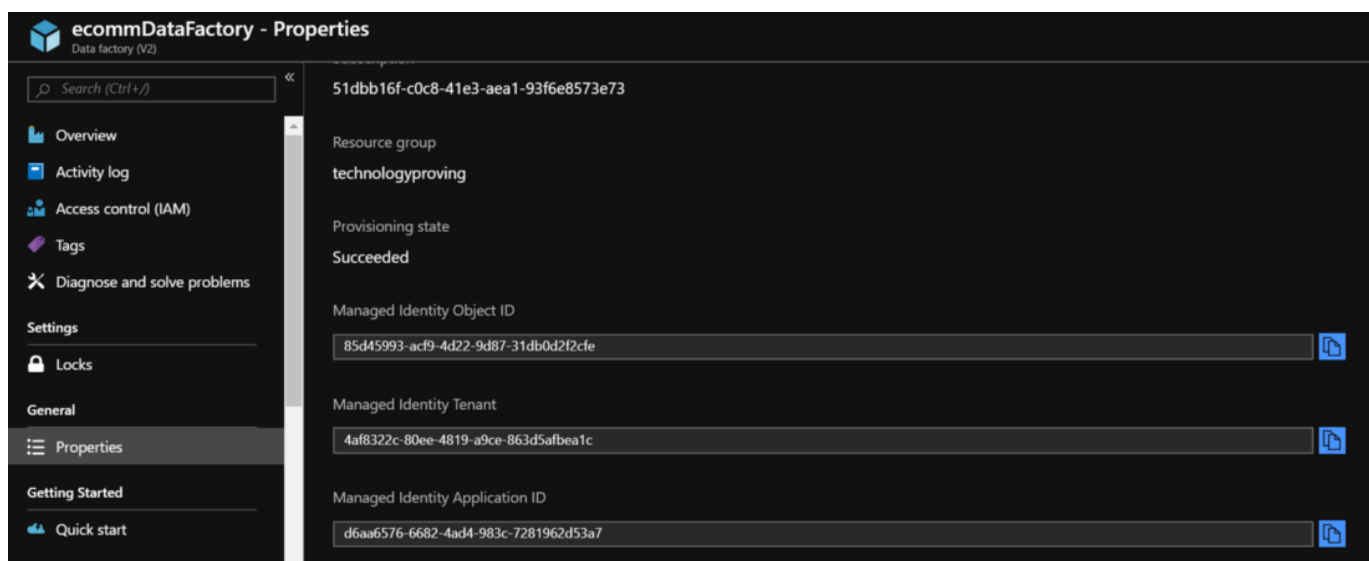
I hope you enjoyed reading and discovered something new about Azure Data Factory.

## Appendix: Azure Datalake Storage Setup

I've added some brief guidance on Azure Datalake Storage setup including links through to the official Microsoft documentation.

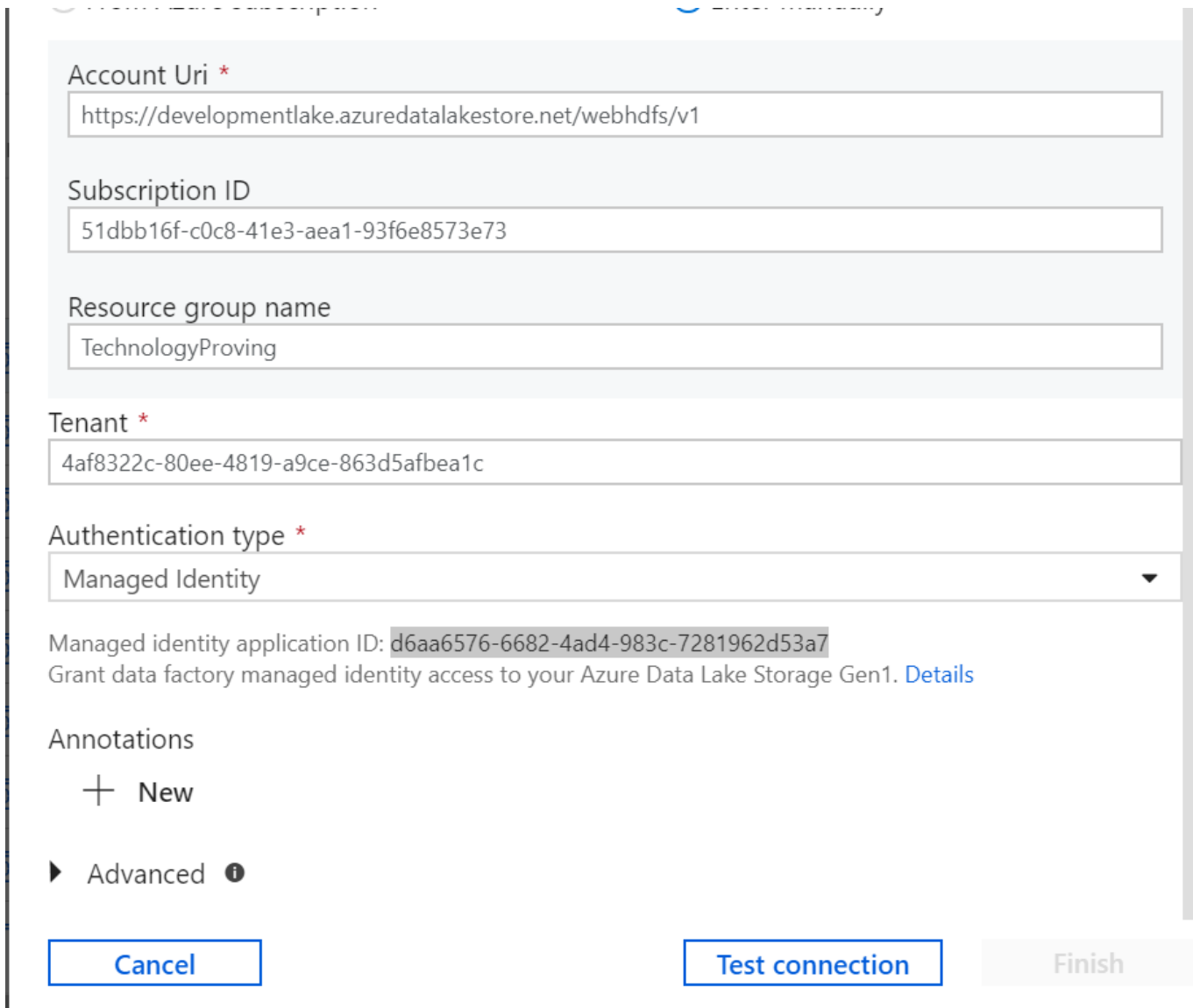
In the article, [Manage Identities](#) were used to allow ADF access to files on the data lake. There are a few ways to discover your ADF's Managed Identity Application Id.

1. You can find the Managed Identity Application ID via the portal by navigating to the ADF's General-Properties blade.





2. You can also find the Managed Identity Application ID when creating a new Azure DataLake Linked service in ADF.



Account Uri \*

https://developmentlake.azuredatastore.net/webhdfs/v1

Subscription ID

51dbb16f-c0c8-41e3-aea1-93f6e8573e73

Resource group name

TechnologyProving

Tenant \*

4af8322c-80ee-4819-a9ce-863d5afbea1c

Authentication type \*

Managed Identity

Managed identity application ID: d6aa6576-6682-4ad4-983c-7281962d53a7

Grant data factory managed identity access to your Azure Data Lake Storage Gen1. [Details](#)

Annotations

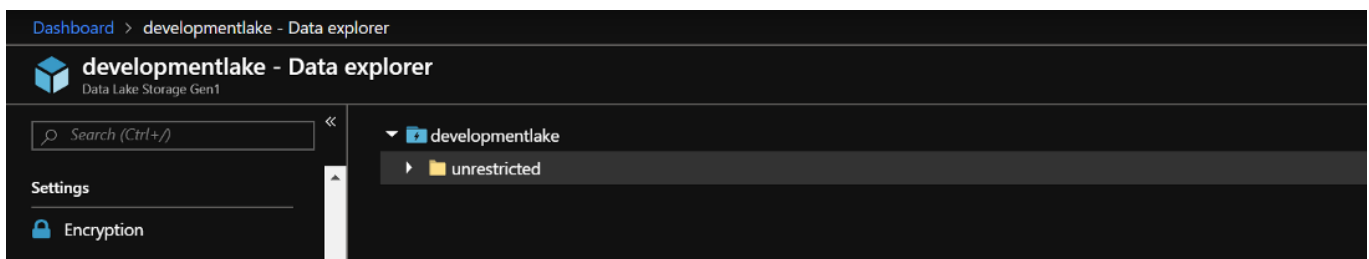
+ New

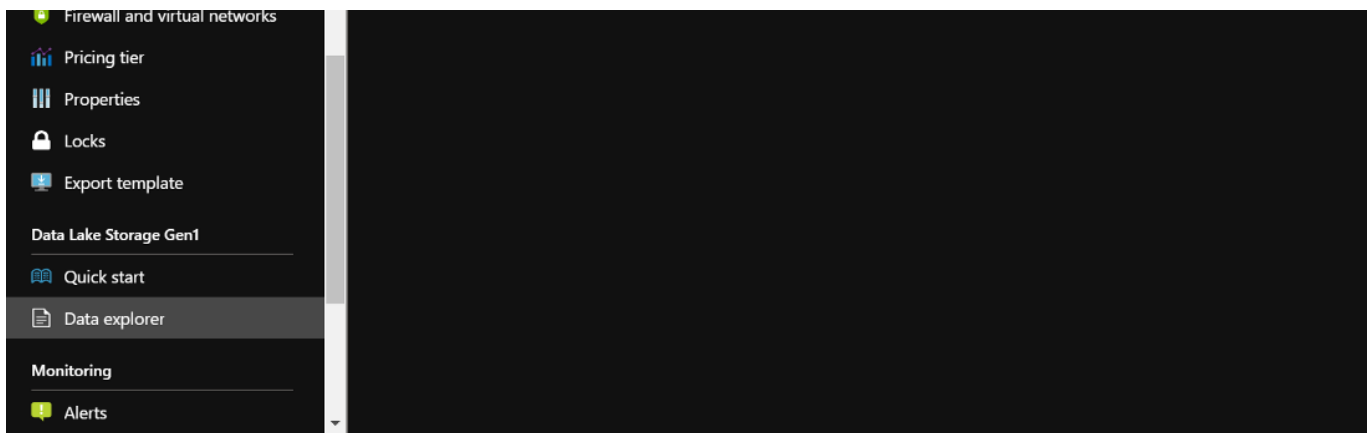
► Advanced ⓘ

Cancel Test connection Finish

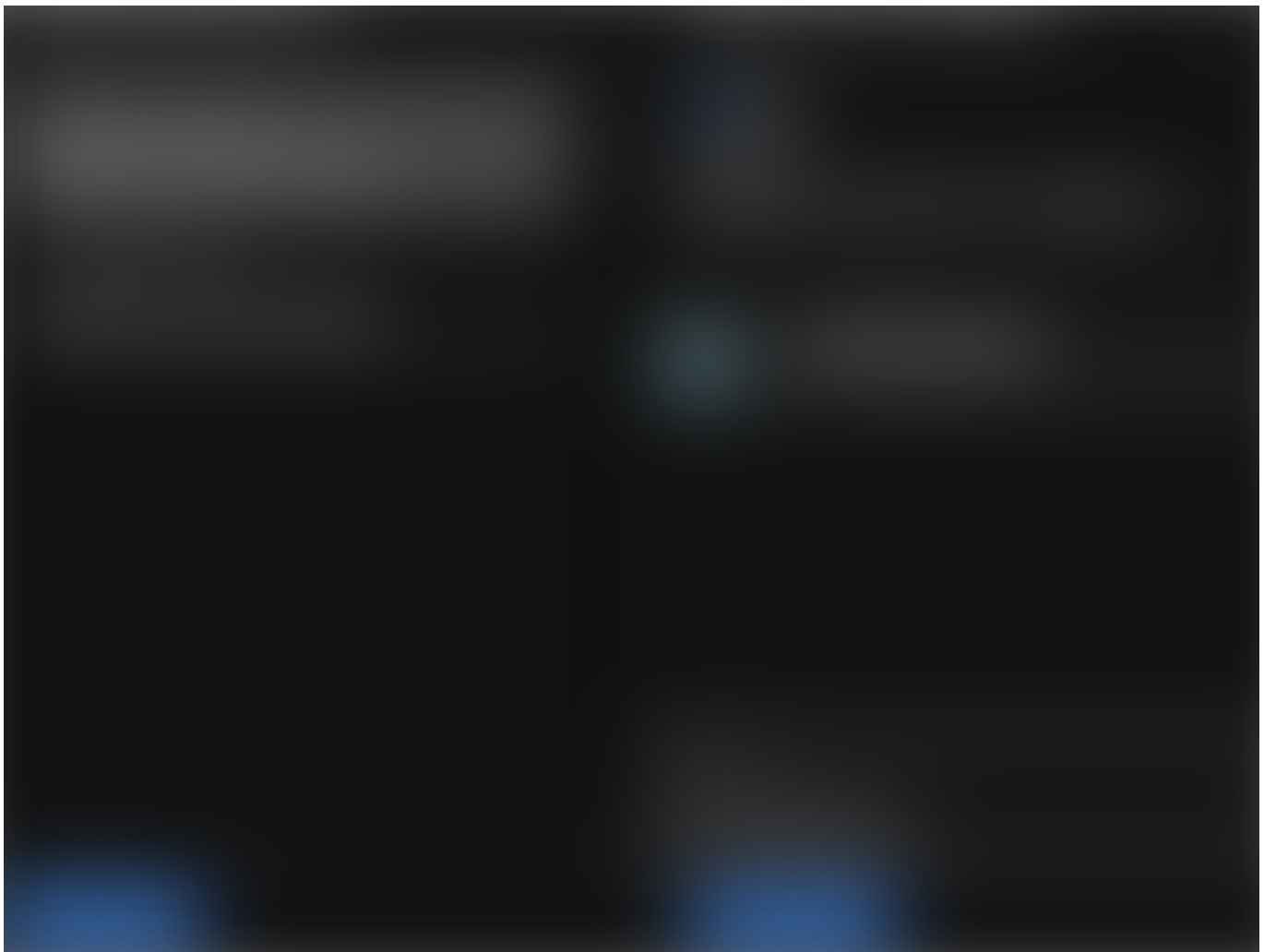
Once the Managed Identity Application ID has been discovered you need to configure Data Lake to allow requests from the Managed Identity. For the purpose of this article, I'll just allow my ADF access to the root folder on the Lake.

Via the Azure Portal, I use the DataLake Data explorer to navigate to the root folder.

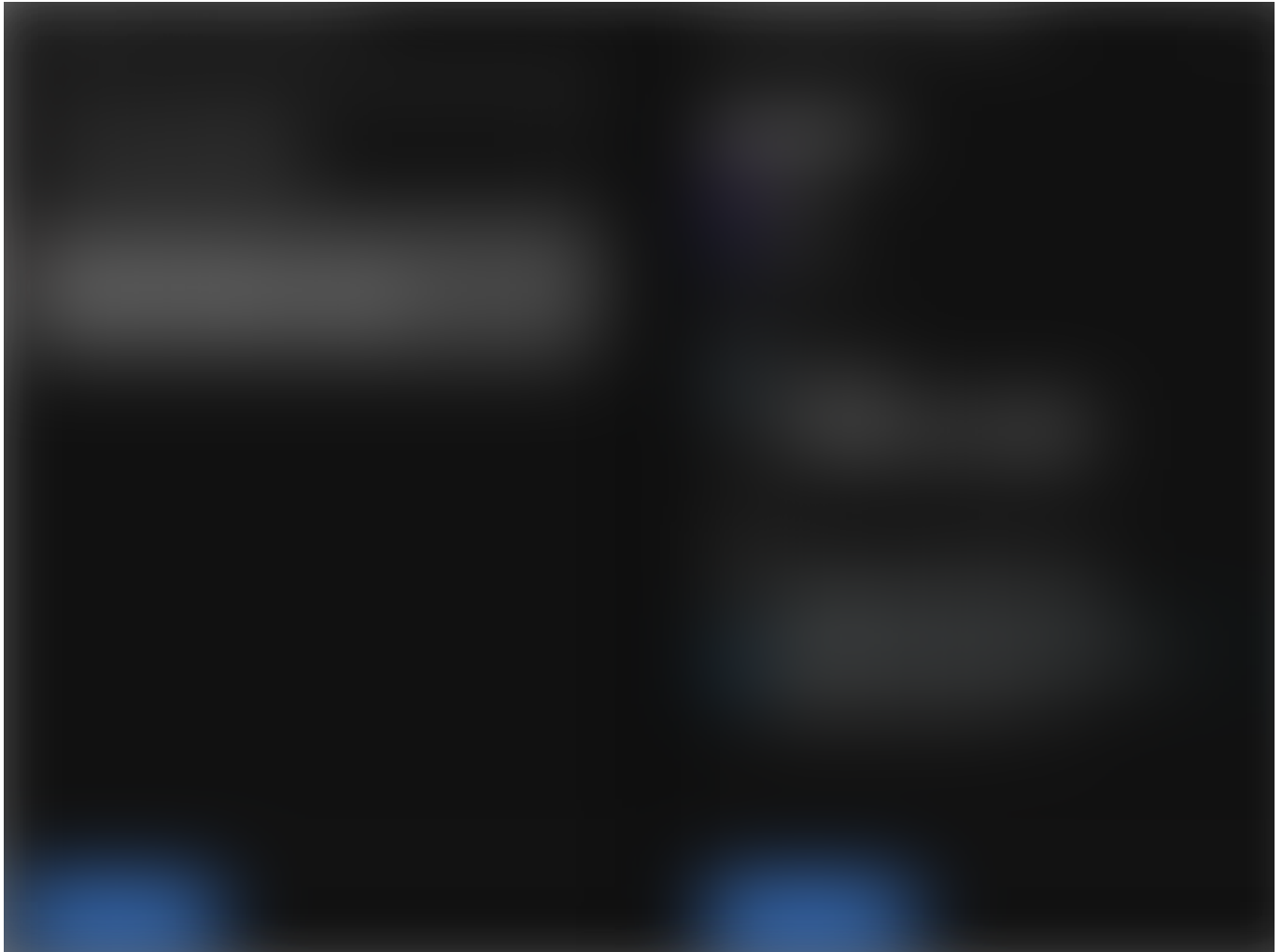




From there navigate to the Access blade. Then its 'add' button and here is where you'll want to type (paste) your Managed Identity Application ID. For a comprehensive guide on setting up Azure Datalake Security visit: <https://docs.microsoft.com/en-us/azure/data-lake-store/data-lake-store-secure-data>



Azure will find the user-friendly name for your Managed Identity Application ID, hit select and move onto permission config. For this example, I'm going to apply read, write and execute to all folders. I've also selected 'Add as: An access permission entry and a default permission entry'. For a more comprehensive guide on ACL configurations visit: <https://docs.microsoft.com/en-us/azure/data-lake-store/data-lake-store-access-control>



## Credits

Thanks to [Jason Horner](#) and his session at [SQLBits 2019](#)

Imagery from:

[Freepik](#)

[Smashicons](#)

[Big Data](#)

[Azure Data Factory](#)

[Json](#)

[Azure Data Lake](#)

[Parquet](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

