Published in Microsoft Azure

Nicholas Hurt Follow
Jan 20, 2020 · 13 min read · Listen

Securing access to Azure Data Lake gen2 from Azure Databricks

There are a number of ways to configure access to Azure Data Lake Storage gen2 (ADLS) from Azure Databricks (ADB). This blog attempts to cover the common patterns, advantages and disadvantages of each, and the scenarios in which they would be most appropriate. Here is a quick summary of options and some of the important considerations:

Scenario	Note	Language Support	Workspace Tier	Cluster Mode	JDBC /ODB
1. Access via SP	Access to all users via mount or direct	All	Standard	Std	Yes
2. Workspace isolation	1 permission group per workspace	All	Standard	Std	Yes
3. AAD Passthrough	Single user only on std cluster	All	Premium	HC for Python, Std for Scala & R	No
4. Cluster scoped SP	Requires cluster access control	All	Premium	Std	Yes
5. Session scoped SP	Requires secret access control	All	Premium	Std	No
6. Table Access Control	Requires cluster access control + table access control	Python/SQL	Premium	HC	Yes

Introduction

In a previous blog I covered the benefits of the lake and ADLS gen2 to those building a data lake on Azure. In another blog I cover the fundamental concepts and structure of the data lake, including some design considerations pre-requisite knowledge of ADLS gen2 which will be helpful when working through these patterns.

From a Databricks perspective, there are two common authentication mechanisms used to access ADLS gen2, either via service principal (SP) or Azure Active Directory

> *When a service principal with read-write access is used to create a mount point,* ***all users in the workspace*** *will have read and write access to the files under that mount point. Whereas if AAD passthrough is used, the users credentials are evaluated against the ACLs of the files and folders.*

Pattern 1. Access via Service Principal

If you wish to provide a group of users, for example data engineers, read and write access to a particular folder and it's contents, the easiest mechanism is to create a mount point using a single service principal at the required folder depth. The mount point (/mnt/<mount_name>) is created once-off per workspace and then may be used by **any user on any cluster in that workspace.**

> *Note access keys are not an option on ADLS whereas they can be used for normal blob containers without HNS enabled.*

Below is sample code to authenticate via a SP using OAuth2 and create a mount point in Scala.

```
configs = {"fs.azure.account.auth.type": "OAuth",
           "fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.Client
           "fs.azure.account.oauth2.client.id": "enter-your-service-principal-application-id-he
           "fs.azure.account.oauth2.client.secret": dbutils.secrets.get(scope = "enter-your-key
           "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/enter-

dbutils.fs.mount(
  source = "abfss://file-system-name@storage-account-name.dfs.core.windows.net/folder-path-here
  mount_point = "/mnt/mount-name",
  extra_configs = configs)
```

mount adls gen2 hosted with ❤ by GitHub view raw

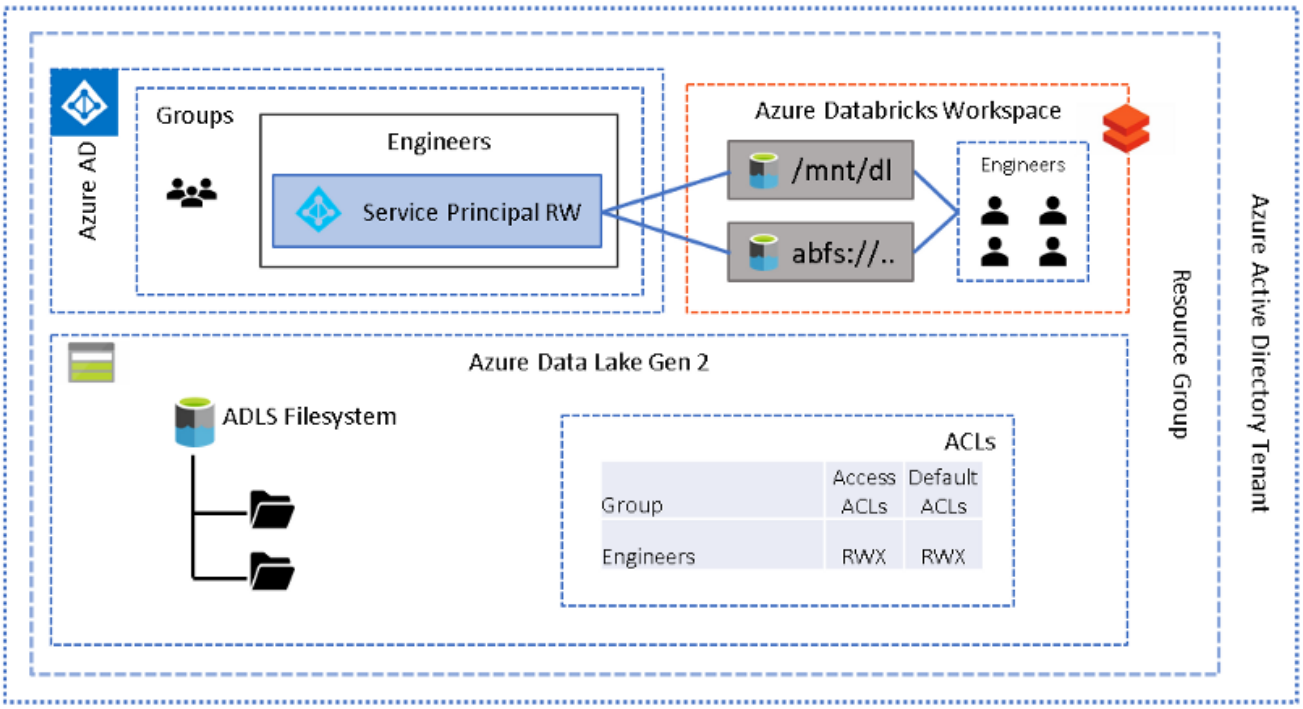
If one had chosen datalake as the mount name, one could verify this had been created using the CLI

```
>databricks configure — token
```

```
datalake
```

Secret scopes were used in the above example instead of using sensitive information in the clear, and most often key vault-backed secret scopes are recommended, however, take note that at the time of writing these can't be scripted via the API or CLI - only Databricks backed scopes can.

From an architecture perspective these are the basic components (excluding secret scopes for simplicity) where "dl" was used as the mount name. Note the mount and ACLs could be at the filesystem (root) level or at the folder level to grant access at the required filesystem depth.



Note the use of default ACLs otherwise any new folders created will be inaccessible

In addition to mount points, access can also be via direct path — Azure Blob Filesystem (ABFS - included in runtime 5.2 and above) as shown in the code snippet below. Tables accessible through SQL can be created using a mount point or direct path references. If you are using the original Windows Azure Storage Blob (WASB) driver it is recommended to use ABFS with ADLS due to greater efficiency with directory level operations.

```
# authenticate using a service principal and OAuth 2.0
spark.conf.set("fs.azure.account.auth.type", "OAuth")
spark.conf.set("fs.azure.account.oauth.provider.type", "org.apache.hadoop.fs.azurebfs.oauth2.Cli
spark.conf.set("fs.azure.account.oauth2.client.id", "enter-your-service-principal-application-ic
spark.conf.set("fs.azure.account.oauth2.client.secret", dbutils.secrets.get(scope = "secret-scop
spark.conf.set("fs.azure.account.oauth2.client.endpoint", "https://login.microsoftonline.com//er

# read data in delta format
readdf=spark.read.format("delta").load(abfs://file-system-name@storage-account-name.dfs.core.wir
```

direct access using service principal hosted with ❤ by GitHub view raw

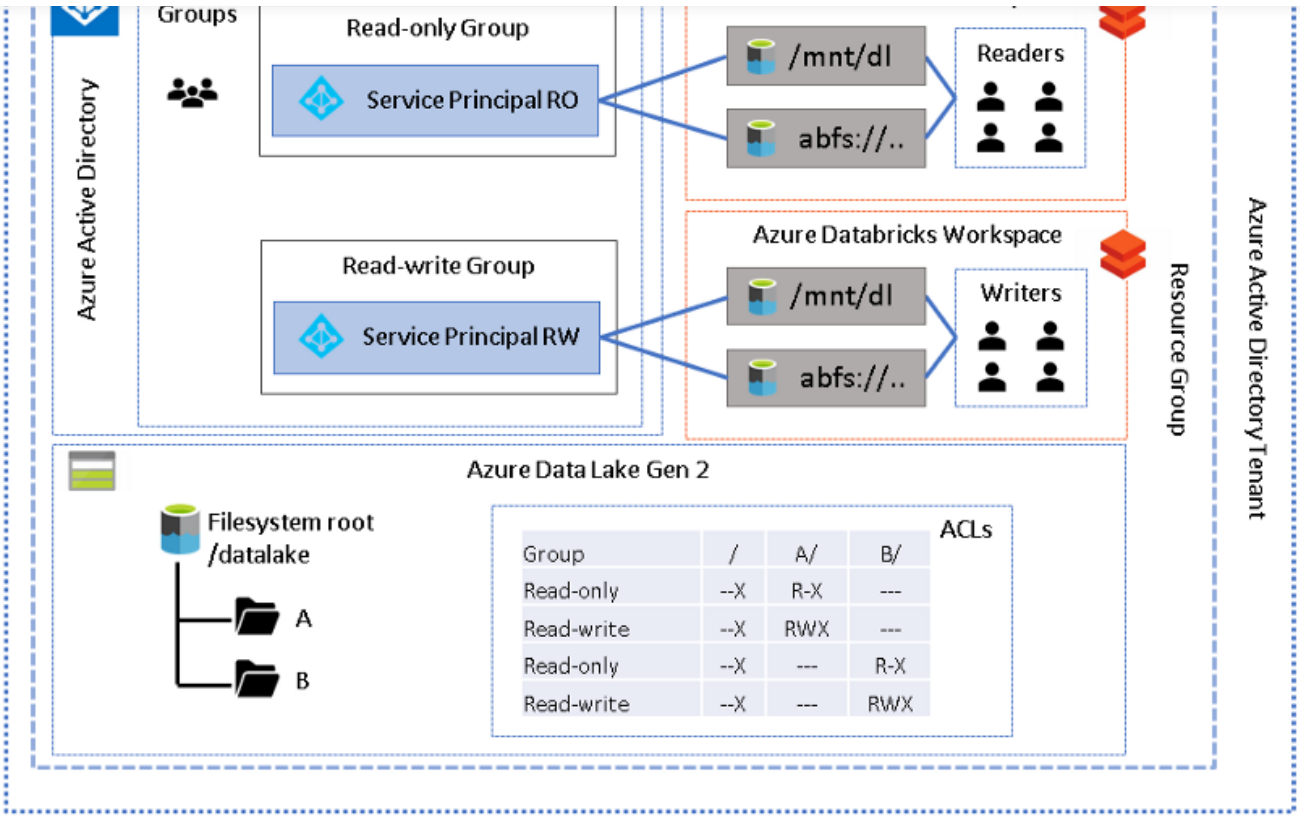
To create a table and start writing SQL queries against the data, reference the mount location (or direct path) and execute the SQL create table syntax either in a SQL notebook or using the SQL magic %sql. Tables can be created over a number of different file types but if you have a particularly complex csv files (e.g. one which includes social media data) you may need to consider special characters and the multiline option as show below:

```
%sql

-- simple create table statement assumes you have data in Delta Lake format at the specified lo
CREATE TABLE events
  USING DELTA
  LOCATION '/mnt/mount-name/path-to-folder-containing-Delta-Lake-data'

-- create table from csv with multi line columns
CREATE TABLE IF NOT EXISTS table_name_goes_here
USING CSV
OPTIONS (
header "true",
delimiter ",",
quote 'quote-characters-go-here',
escape 'escape-characters-go-here',
mode 'FAILFAST',
multiLine 'TRUE',
inferSchema "true",
path "/mnt/mout-name/path-to-csv.csv")
```

This access pattern is **unlikely to satisfy most security requirements**, especially if you have invested time and effort and time into building an enterprise data lake. Commonly there is a need to secure parts of the lake or even individual data assets, particularly if you have groups of readers and writers interacting with the lake, enriching and curating data as well as consuming data respectively. You will need to mitigate the risk of unauthorised access, data assets being deleted, inadvertently overwritten or corrupted. To this end, you may be tempted to create two mount points, granting read-only access to one service principal and write permissions to another. Remembering that mount points are accessible to all users in the workspace, this implementation will not work, which leads to our second approach...

Pattern 2. Multiple workspaces — permission by workspace

This is an extension of the first pattern whereby multiple workspaces are provisioned, and different groups of users are assigned to different workspaces. Each group/workspace will use a different service principal to govern the level of access required either via a configured mount point or direct path. Essentially, you are mapping a service principal to each group of users and each service principal will have a defined set of permissions on the lake. In order to assign users to a workspace simply ensure they are registered in your Azure Active Directory (AAD) and an admin (those with contributor or owner role on the workspace) will need to add users to the appropriate workspace. The architecture below depicts two different folders and two groups of users (readers and writers) on each.



* "dl" represents a mount name to filesystem "datalake"

The downside to this approach is the **proliferation of workspaces** — n groups = n workspaces and this may become rather costly if you cannot balance or justify the number of users/workload per workspace. The workspace itself does not incur cost, it is the fact that there needs to be at least one running cluster in the workspace to provide users access to the lake, and this cluster cannot be shared with other groups. Multiple workspaces also incur an administrative cost. The next pattern may overcome this challenge but may present others depending on your requirements...

Pattern 3 — AAD Credential passthrough

AAD passthrough will allow different groups of users to all work in the same workspace and access data either via mount point or direct path. Using this security mechanism, authenticated Databricks user's credentials are passed through to ADLS gen2 and the user's permissions are evaluated against the files and folder ACLs. The user needs to have the same identity in both in the AAD tenant and in the Databricks workspace. The feature is enabled at the cluster level under the advanced options.

```
1  val configs = Map("fs.azure.account.auth.type" -> "CustomAccessToken",  "fs.azure.account.custom
2  // Optionally, you can add <directory-name> to the source URI of your mount point.
3  dbutils.fs.mount(
4    source = "abfss://file-system-name@storage-account-name.dfs.core.windows.net/folder-path-here"
5    mountPoint = "/mnt/mount-name",
6    extraConfigs = configs)
```

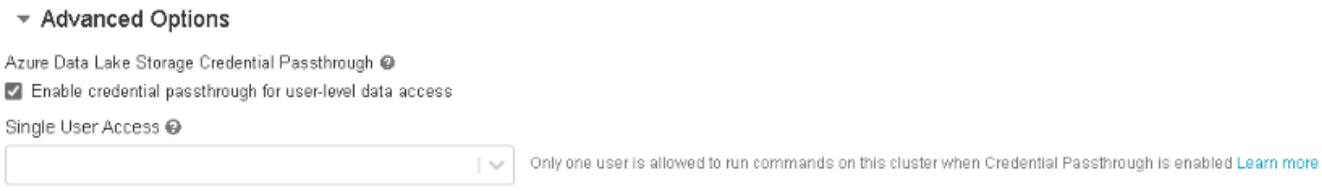
Mount with AAD passthough enabled hosted with ❤ by GitHub view raw

Any user reading or writing via the mount point will have their credentials evaluated. To access data **directly** without a mount point simply use the abfs path on a cluster with **AAD Passthrough enabled**, for example:

```
# read data in delta format using direct path
readdf = spark.read
 .format("<file format>")
 .load("abfss://<filesys>@<storageacc>.dfs.core.windows.net/<path>")
```

. . .

Originally this functionality was only available using high concurrency clusters and supported only Python and SQL notebooks, but recently standard clusters support for AAD passthrough using R and Scala notebooks were announced. One major consideration however for standard clusters, is that only a single user can be enabled per cluster.



Standard Clusters — enable AAD Passthrough

Note: for high concurrency clusters this setting still needs to be enabled under the advanced section of the cluster config:

Enable table access control and only allow Python and SQL commands

Azure Data Lake Storage Credential Passthrough

Enable credential passthrough for user-level data access and allow only Python and SQL commands

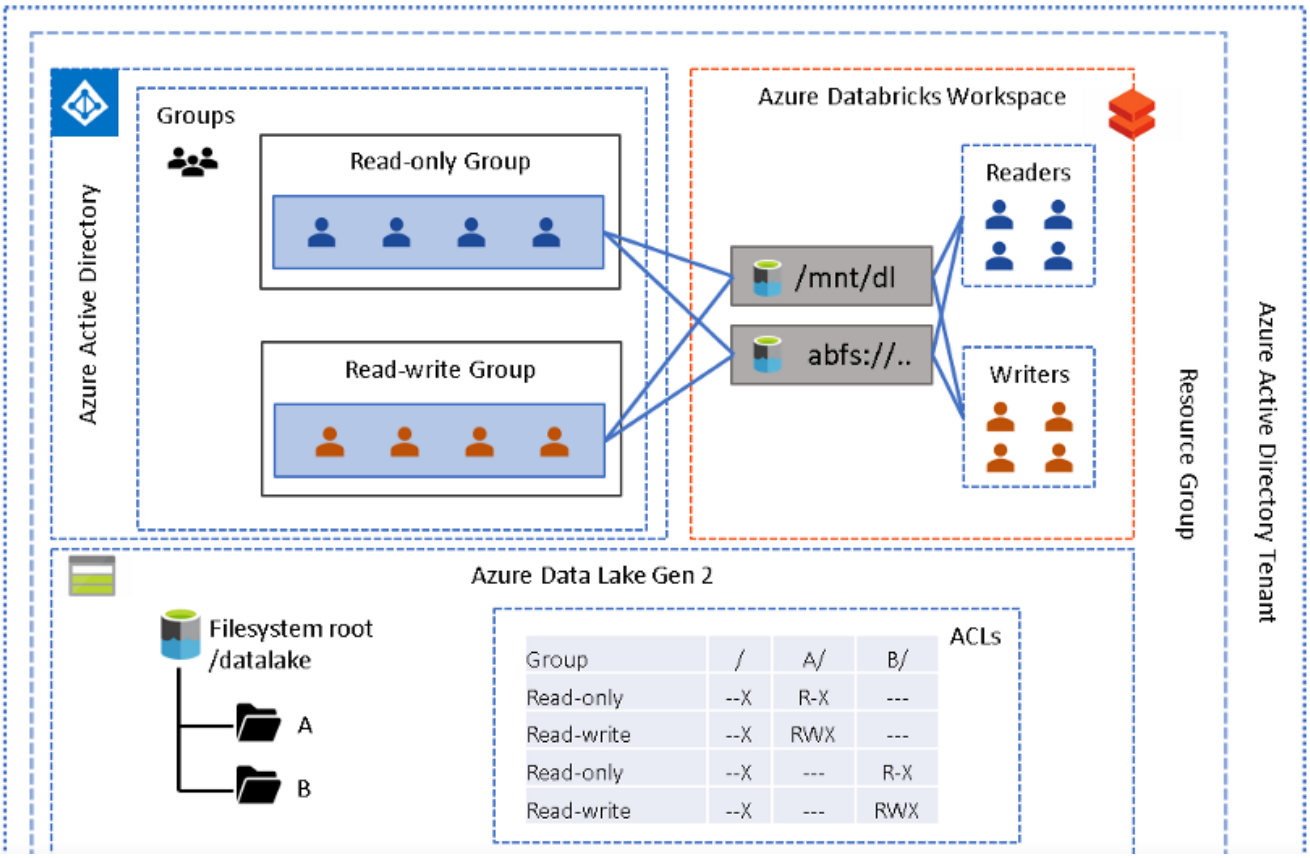
Spark Tags Logging Init Scripts Permissions

Spark Config

```
spark.databricks.cluster.profile serverless
spark.databricks.delta.preview.enabled true
spark.databricks.repl.allowedLanguages python,sql
spark.databricks.passthrough.enabled true
spark.databricks.pyspark.enableProcessIsolation true
```

High Concurrency Clusters — enable AAD Passthrough

A subtle but important difference in this pattern is that service principals are not required to delegate access, as it is the user's credentials that are used.



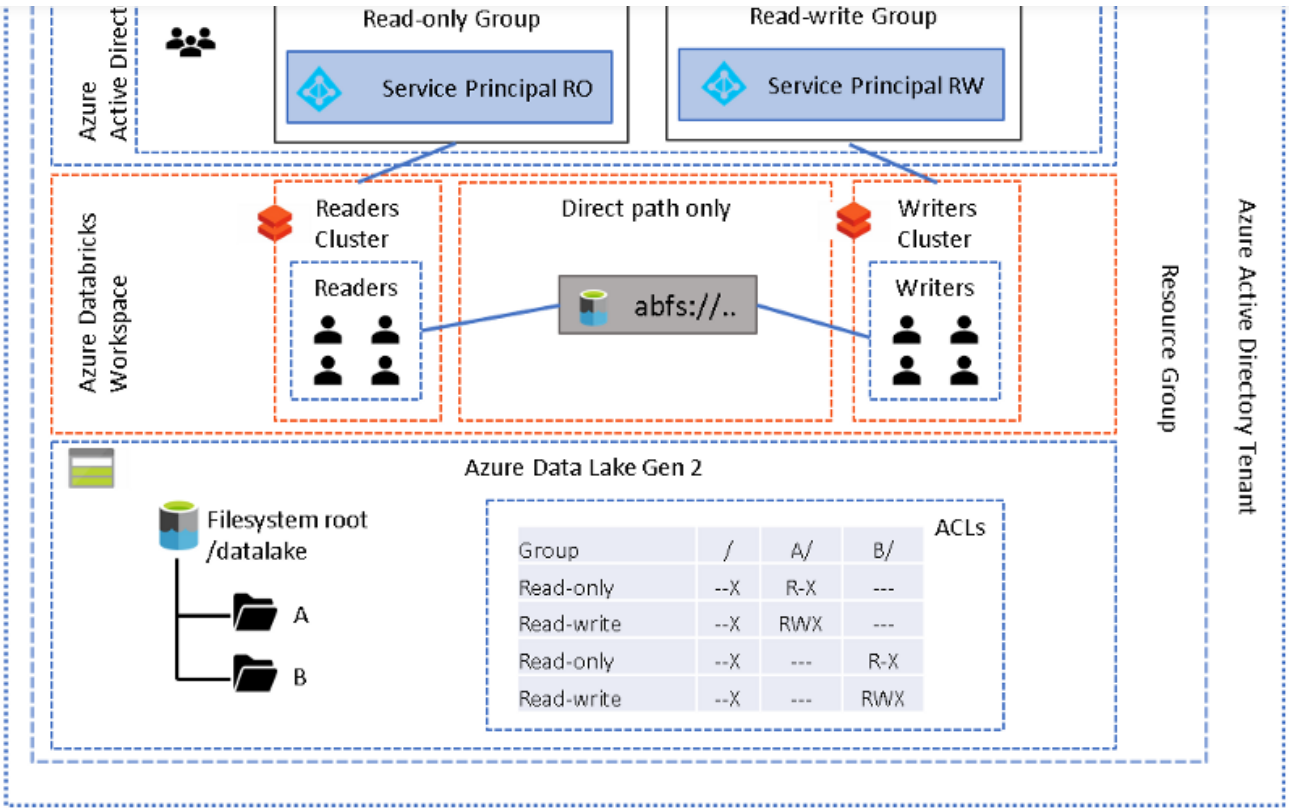
There are some further considerations to note at the time of writing:

- The minimum runtime versions as well as which PySpark ML APIs which are not supported, and associated supported features
- **Databricks Connect is not supported**
- **Jobs are not supported**
- jdbc/odbc (BI tools) is not yet supported

If any of these present a challenge or you would like to enable more than one Scala or R developer to work on a cluster at the same time, then you may need to consider one of the other patterns below.

Pattern 4. Cluster scoped Service Principal

In this pattern, each cluster is "mapped" to a unique service principal. By restricting users or groups to a particular cluster, using the "can attach to" permission, will ensure that access to the data lake is restricted by the ACLs of the mapped service principal.



This pattern will allow you to use multiple clusters in the same workspace, each with it's own permissions according to the service principal set in the cluster config:

```
fs.azure.account.auth.type OAuth
fs.azure.account.oauth.provider.type org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvic
fs.azure.account.oauth2.client.id <service-principal-application-id>
fs.azure.account.oauth2.client.secret {{secrets/<your scope name>/<secret name>}}
fs.azure.account.oauth2.client.endpoint https://login.microsoftonline.com/<tenant id>/oauth2/tok
```

Service Principal in Cluster Config hosted with ❤ by GitHub view raw

Note the way in secrets are referenced in the config section which is different from the usual dbutils syntax

The benefit of this approach is that the scope and secret names are not exposed to end-users and they do not require read access to the secret scope however the creator of the cluster will.

. . .

The pattern relies on users using the direct access method, via ABFS, and mount

```
# read data in delta format using direct path
readdf = spark.read
 .format("delta")
 .load("abfss://<filesys>@<storageacc>.dfs.core.windows.net/<path>")
```

Until there is an in-built way to prevent mount points being created, you may wish to write an alert utility which runs frequently checking for any mount points using the CLI (as shown in the first pattern) and sends a notification if any are unauthorised mount points are detected.

. . .

This pattern could be useful when both engineers and analysts require different sets of permissions and assigned to the same workspace. The engineers may need read access to one or more source data sets and then write access to a target location, with read write access to a staging or working location. This requires a single service principal to have access to all the data sets in order for the code to execute — more on this in the next pattern. The analysts may need read access to the target folder and nothing else. Analysts and engineers will be separated by cluster but allow them to work in same workspace.

The disadvantage of this approach is dedicated clusters for each permission group, i.e. no sharing of clusters across permission groups. In other words, each service principal, and therefore each cluster, should have sufficient permissions in the lake to run the desired workload on that cluster. The reason for this is that *a cluster can only be configured with a single service principal at a time*. In a production scenario the config should be specified through scripting the provisioning of clusters using the CLI or API.

Depending on the number of permission groups required, this pattern could result in a **proliferation of clusters**. The next pattern may overcome this challenge but will require each user to execute authentication code at run time.

attempting to access ADLS will need to use the direct access method and execute OAuth code prior to accessing the required folder. Consequently this approach will not work when using odbc/jdbc connections. Also note that **only one service principal can be set in session at a time** and this will have a significant influence the design as described later.

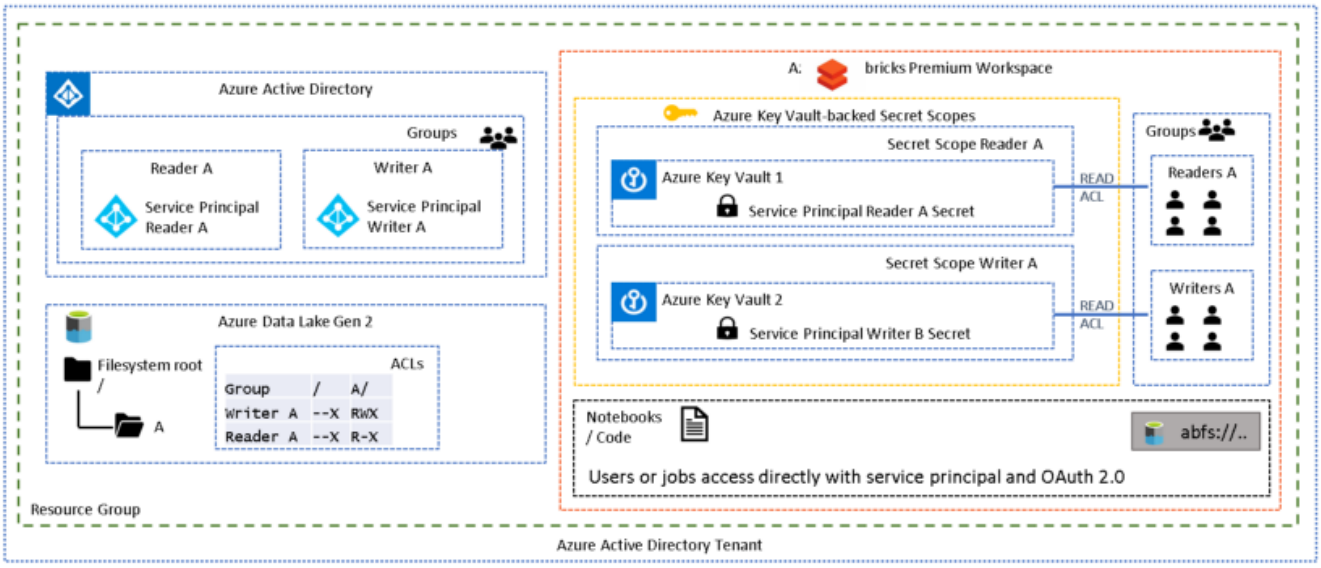
```
# authenticate using a service principal and OAuth 2.0
spark.conf.set("fs.azure.account.auth.type", "OAuth")
spark.conf.set("fs.azure.account.oauth.provider.type", "org.apache.hadoop.fs.azurebfs.oauth2.Cli
spark.conf.set("fs.azure.account.oauth2.client.id", "enter-your-service-principal-application-ic
spark.conf.set("fs.azure.account.oauth2.client.secret", dbutils.secrets.get(scope = "secret-scop
spark.conf.set("fs.azure.account.oauth2.client.endpoint", "https://login.microsoftonline.com//er

# read data in delta format
readdf=spark.read.format("delta").load(abfs://file-system-name@storage-account-name.dfs.core.wir
```

direct access using service principal hosted with ❤ by GitHub view raw

This pattern works well where different permission groups (such as analysts and engineers) are required but you don't want to have take on the administrative burden of isolating them by cluster. *As in the previous approach, mounting folders using the provided service principal/secret scope details should be forbidden.*

The mechanism which ensures that each group has the appropriate level of access is through their ability to "use" a service principal which has been added to the AAD group with the desired level of access. The way to effectively "map" the user group's level of access to a particular service principal is by granting the Databricks user group access to the secret scope (see below) which stores the credentials for that service principal. Armed with the secret scope name and the associated key name(s), users can then run the authorisation code shown above. The client.secret (service principal's secret) is stored as a secret in the secret scope but so to can any other sensitive details such as the service principals application ID and tenant ID.

> *The disadvantage of this approach is the proliferation of secret scopes of which there is a limit of 100 per workspace. Additionally the premium plan is required in order to assign granular permissions to the secret scope.*

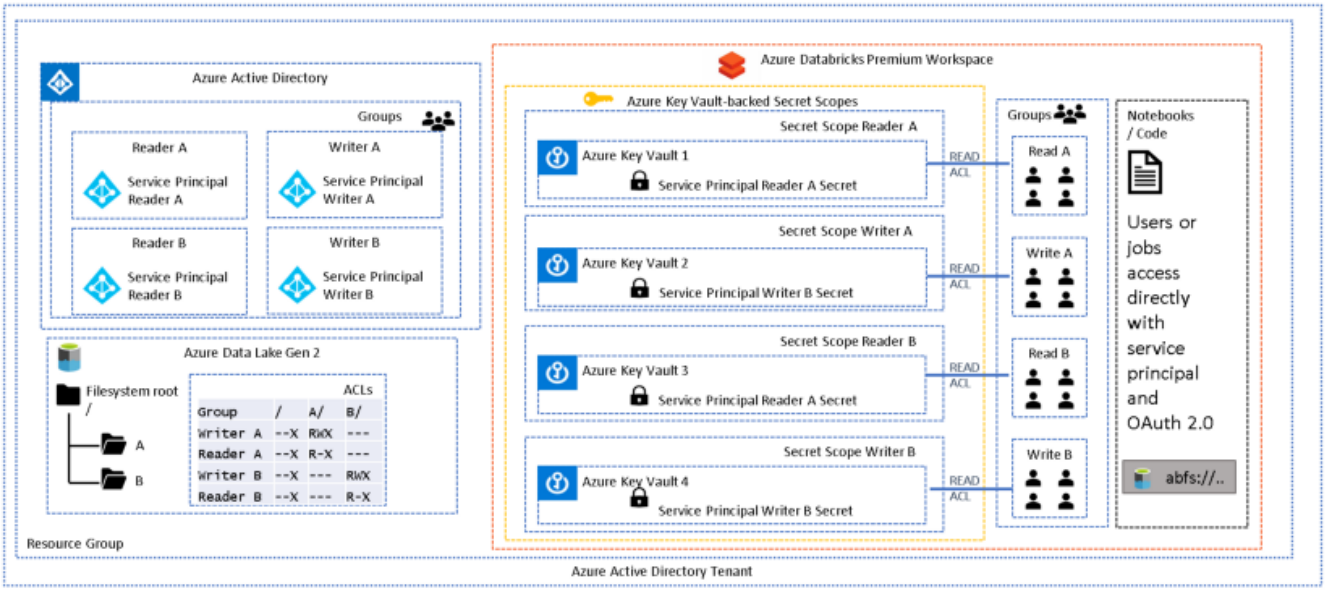


The above diagram depicts a single folder (A) with two sets of permissions, readers and writers. AAD groups reflect these roles and have been assigned appropriate folder ACLs. Each AAD group contains an associated service principal and the credentials for each service principal is stored in a unique secret scope. Each group in the Databricks workspace contains the appropriate users, and the group is assigned READ ACLs on the associated secret scope, which allows them to "use" the service principal mapped to their level of permission. Here is an example CLI command to grant read permissions to the GrWritersA group on SsWritersA secret scope. Note that ACLs are at secret scope level, not at secret level which means that one secret scope will be required per service principal.

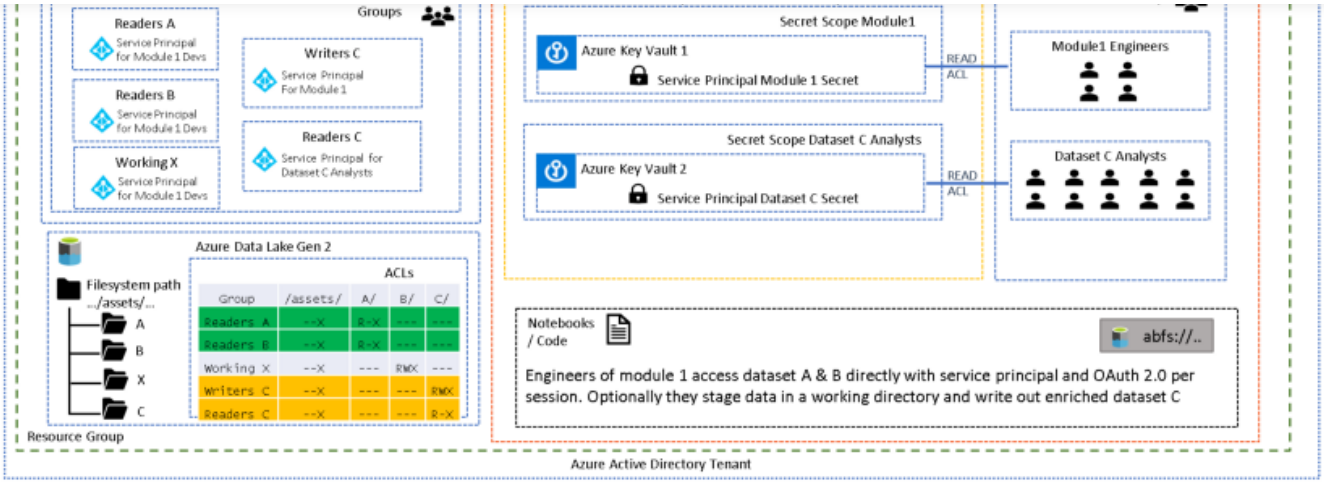
```
databricks secrets put-acl --scope SsWritersA --principal GrWritersA
--permission READ

databricks secrets get-acl --scope SsWritersA --principal GrWritersA

Principal Permission

 – – – – – – – – – – – –

GrWritersA READ
```

How this is may be implemented for your data lake scenario requires careful thought and planning. In very general terms this pattern being applied in one of

1. Analysts (read-only) and engineers (read-write) are working within a single folder structure, and they do not require access to additional datasets outside of their current directory. The diagram below depicts two folders A and B, perhaps representing two departments, and each department has their own analysts and engineers working on their data, and should not be allowed access to the other department's data.



2. Engineers and analysts are working on different projects and should have clear separation of concerns. Engineers working on "Module 1" require read access to multiple source data assets (A & B). Transformations and joins are run to produce another data asset (C). Engineers may also require a working or staging directory to persisting output during various stages (X). For the entire pipeline to execute, Service Principal for Module 1 Developers is added to the various groups which provide access to all necessary folders through assigned ACLs. Analysts need to produce analytics using the new data asset (C) but should not have access to the source data therefore they use the Service Principal for Dataset C which is added to the Readers C group only.



It may seem more logical to have one service principal per data asset but when multiple permissions are required for a single pipeline to execute in Spark then one needs to consider how lazy evaluation works. When attempting to use multiple service principals in the same notebook/session one needs to remember that the read and write will be executed only once the write is triggered. One cannot therefore set the authentication to one service principal for one folder and then to another prior to the final write operation, as the read operation will be executed only when the write is triggered.

> *This means a single service principal will need to encapsulate the permissions of a single pipeline execution rather than a single service principal per data asset.*

Pattern 6. Databricks Table Access Control

One final pattern, which not technically an access pattern to ADLS, implements security at the table (or view) level rather than the data lake level. This method is native to Databricks and involves granting, denying, revoking access to tables or views which may have been created from files residing in ADLS. Access is granted programmatically (from Python or SQL) to tables or views based on user/group. This approach requires both cluster and table access control to be enabled and requires a premium tier workspace. File access is disabled through a cluster level configuration which ensures the only method of data access for users is via the pre-configured tables or views. This works well for analytical (BI) tools accessing tables/views via odbc but limits users in their ability to access files directly and does not support R and Scala.

it will be a combination of these patterns which will suit your production scenario. Below is a table summarising the above access patterns and some important considerations of each.

Scenario	Note	Language Support	Workspace Tier	Cluster Mode	JDBC /ODB
1. Access via SP	Access to all users via mount or direct	All	Standard	Std	Yes
2. Workspace isolation	1 permission group per workspace	All	Standard	Std	Yes
3. AAD Passthrough	Single user only on std cluster	All	Premium	HC for Python, Std for Scala & R	No
4. Cluster scoped SP	Requires cluster access control	All	Premium	Std	Yes
5. Session scoped SP	Requires secret access control	All	Premium	Std	No
6. Table Access Control	Requires cluster access control + table access control	Python/SQL	Premium	HC	Yes