

Lab 12: Accessing ADLS Gen2 Using Service Principal (Production-Ready, Step-by-Step)

Author: Dr. Sandeep Kumar Sharma

Why this lab matters (short)

Yes — you're right. Using a **Service Principal (SP)** is the recommended and secure way to allow Databricks to access ADLS in production. Compared to account keys or SAS tokens, Service Principals provide fine-grained control, can be rotated, and avoid embedding long-lived credentials in notebooks.

This lab recreates the typical code flow you shared, but with **best-practice recommendations** (do not hard-code secrets; use Databricks Secret Scopes instead). I'll show both: the quick demo approach and the recommended secure approach.

Learning Objective

By the end of this lab you will be able to:

- Understand what a Service Principal is and why it's secure
- Configure Azure AD Service Principal permissions for ADLS Gen2
- Use a Service Principal in Databricks to authenticate to ADLS Gen2
- Use Databricks Secret Scopes to avoid hard-coding secrets
- Read and write files using `abfss://` paths in Spark

Learning Outcome

You will be able to:

- Create Spark configs for OAuth (Client Credentials flow)
- Mount or directly read ADLS Gen2 using `abfss://` URIs
- Verify permissions and troubleshoot common errors

Prerequisites

1. Azure subscription and permission to register an app / create a service principal in Azure AD.
2. The Service Principal must be assigned **Storage Blob Data Contributor** (or equivalent) on the target storage account or container.
3. Databricks workspace and permissions to create secret scopes (or permission to ask your platform team to create one).
4. Example values (replace with real values):

```
storage_account_name = "sandeepkaushikstorage321"  
storage_container_name = "input"
```

```
client_id = "c2609c9b-6e38-40db-980f-96ff0ff22ae7"
# DO NOT paste your client_secret here in production
client_secret = "<CLIENT_SECRET>"
tenant_id = "30cc3176-5a12-41ed-b15b-2e34c82f6e8c"
```

Part A — Quick Concept: What is a Service Principal?

A **Service Principal** is an identity created for use with applications, hosted services, and automated tools to access Azure resources. Think of it as a login (client id + secret/certificate) that you can grant precise permissions. Unlike account keys, SPs are auditable and can be rotated.

Part B — Create and Configure Service Principal (Azure Portal / Azure CLI)

(If you already have SP and client id/secret, skip to Part C.)

Azure Portal steps (brief): 1. Go to **Azure Active Directory → App registrations → New registration**. Give a name and register.
2. In **Certificates & secrets**, create a **Client secret** and copy it (store securely).
3. Go to your **Storage Account → Access control (IAM) → Add role assignment**. Assign **Storage Blob Data Contributor** to the Service Principal (search the app name).

Azure CLI (optional):

```
# create SP
az ad sp create-for-rbac --name "my-databricks-sp"
--role "Storage Blob Data Contributor"
--scopes /subscriptions/<sub-id>/resourceGroups/<rg>/providers/
Microsoft.Storage/storageAccounts/<storage-account>
```

This returns `appId` (client_id), `password` (client_secret), and `tenant`.

Part C — Recommended: Store SP Secret in Databricks Secret Scope

We must never hard-code secrets in notebooks. Use Databricks secret scopes. If you don't have a scope, create one (CLI or UI). Example using Databricks CLI or the UI:

Create secret scope (Databricks CLI)

```
databricks secrets create-scope --scope my-adls-scope
```

Store secret (Databricks CLI)

```
databricks secrets put --scope my-adls-scope --key sp-client-secret  
# then paste the client_secret when prompted
```

Now in notebook we will access `dbutils.secrets.get("my-adls-scope", "sp-client-secret")`.

Part D — Spark Configuration Using Service Principal (Secure way)

Use this pattern in your notebook. Replace placeholders with your actual values.

```
storage_account_name = "sandeepkaushikstorage321"
storage_container_name = "input"
client_id = "c2609c9b-6e38-40db-980f-96ff0ff22ae7"
tenant_id = "30cc3176-5a12-41ed-b15b-2e34c82f6e8c"

# Get the secret from Databricks secret scope (recommended)
client_secret = dbutils.secrets.get(scope="my-adls-scope", key="sp-client-secret")

# Set Spark configs for OAuth (client credentials)
spark.conf.set(f"fs.azure.account.auth.type.{storage_account_name}.dfs.core.windows.net", "OAuth")
spark.conf.set(f"fs.azure.account.oauth.provider.type.{storage_account_name}.dfs.core.windows.net",
              "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider")
spark.conf.set(f"fs.azure.account.oauth2.client.id.{storage_account_name}.dfs.core.windows.net", client_id)
```

```

spark.conf.set(f"fs.azure.account.oauth2.client.secret.
{storage_account_name}.dfs.core.windows.net", client_secret)
spark.conf.set(f"fs.azure.account.oauth2.client.endpoint.
{storage_account_name}.dfs.core.windows.net",
              f"https://login.microsoftonline.com/{tenant_id}/oauth2/token")

# Construct abfss path
full_adls_path = f"abfss://{{storage_container_name}}
@{storage_account_name}.dfs.core.windows.net/"

# Verify by listing files
display(dbutils.fs.ls(full_adls_path))

```

Notes: - We used `abfss://` because ADLS Gen2 (HNS enabled) prefers this scheme. - `dfs.core.windows.net` domain is used for ADLS Gen2. - `ClientCredsTokenProvider` performs client credentials OAuth flow under the hood.

Part E — Demo: Read a CSV and Write Back (Transformation Example)

Once the Spark config is in place, you can read/write normally.

```

input_csv = full_adls_path + "employee.csv"

# Read
df = spark.read.csv(input_csv, header=True, inferSchema=True)
display(df)

# Example transformation: add a column
from pyspark.sql.functions import current_timestamp

transformed = df.withColumn("ingested_ts", current_timestamp())
display(transformed)

# Write back to a different folder in the same container
output_path = full_adls_path + "processed/employee_transformed"
transformed.write.mode("overwrite").parquet(output_path)

# Verify
display(dbutils.fs.ls(full_adls_path + "processed"))

```

Part F — Alternative: Quick Demo (Directly in Notebook, not recommended)

If you must demo quickly (for a classroom only), you can set `client_secret` directly in the notebook like your original snippet. **But avoid this on shared or production workspaces.**

```
# QUICK DEMO (NOT RECOMMENDED FOR PRODUCTION)
client_secret = "<CLIENT_SECRET_HERE>" # temporary demo only
# set the same spark.conf values as above
```

Part G — Troubleshooting Common Errors

- `403 Forbidden` → check SP role assignment and scope (storage account vs container).
- `Invalid client secret` → ensure secret is correct and not expired.
- `AADSTS7000215` → possible wrong client id/tenant or app not configured for client credentials.
- `Mount errors` → do not mount with SP unless necessary; use direct `abfss` access.

Part H — Security Best Practices (Must read)

1. Use **Databricks Secret Scopes** or **Azure Key Vault-backed scopes** to store `client_secret`.
2. Assign the **least privilege** role (e.g., Container-level rather than full storage account if possible).
3. Rotate secrets regularly and monitor sign-ins for the SP in Azure AD.
4. Use Managed Identity (if available) for even stronger security (no secrets).

Practice Tasks

1. Create a Service Principal and assign it Storage Blob Data Contributor role to the storage account.
2. Create a Databricks secret scope and store the SP secret.
3. Run the secure notebook flow (Part D) and read a file using `abfss://`.
4. Transform and write the result to `processed/` folder in the container.

End of Lab 12

This lab shows how to use a Service Principal to access ADLS Gen2 in Databricks securely, with runnable code and best practices. If you want, I will now create **Lab 13 – Using Managed Identity (Azure AD) for**

Databricks or generate a ready-to-run notebook that includes helper functions (`connect()`, `read_csv()`, `write_parquet()`).