Polaris Migrator/Synchronization Tool

Author: Mansehaj Singh
Date: Apr 10, 2025

Background

Migrating between two Polaris instances is a tedious manual process. It requires creating the entities, copying over the assignments between them, managing access control to be able to read and create the entities, and configuring authentication to support long running migrations.

There are several demonstrable use cases where a dedicated and maintained tool that performs migrations between two Polaris instances would be favourable:

- **Migration:** A user may have an active Polaris deployment that they want to migrate to a managed cloud offering like Snowflake Open Catalog.
- **Preventing Vendor Lock-In:** A user may currently have a managed Polaris offering and want the freedom to switch providers or to host Polaris themselves.
- **Backup:** Modern data solutions often require employing redundancy. This tool when run on a periodic cron can keep a backup that lags the source Polaris instance by some period.

The existing Nessie Iceberg Catalog Migrator tool which is being adopted into polaris-tools does support performing Iceberg REST Catalog migrations that are compatible with Polaris, but these have to be performed on a per-catalog basis, and does not support migrating Polaris management API entities.

This proposal introduces a new tool to enable full Polaris to Polaris migration/mirroring including all Polaris management entities as well as Iceberg metadata. The tool is designed with emphasis on idempotency and incremental migration.

One further goal of this tool is to introduce the logic in a way that is as decoupled from the CLI implementation as possible to enable usage of the tool's apis outside of the CLI implementation.

Design

The tool supports the following Management API entities:

- Optionally, Principals see [Design] Polaris Migrator/Synchronization Tool

- Optionally, assignment of Principals to Principal Roles see [Design] Polaris Migrator/Synchronization Tool
- Principal Roles
- Catalogs
    - Catalog Roles
        - Assignment of catalog roles to principal roles
        - Grants

The tool supports the following Iceberg Catalog entities:

- Namespaces
- Tables

~~Limitations:~~

- ~~The tool does not yet migrate Principals nor their assignment to Principal Roles. This is because credential migration has not yet been explored, so it currently needs to be handled with alternative solutions, like keeping an external IDP. See Synergy with an External IDP~~

Open API Code Generation

All API clients and models are generated from the Polaris management API rest specification. This ensures easier support and maintenance of the tool long-term, as we can automatically propagate the introduction of new properties, as well as easily write logic to support new entities introduced to Polaris. For example, once Policies are fully introduced.

Migration/Synchronization Workflow

(1) Idempotent Omnipotent Principal Creation

To do a full management and catalog internal migration, we need the tool to have access to catalog internals. In Apache Polaris, the design of the authorization engine aims to keep a distinction between the idea of a `service_admin` - someone with a service level access to perform actions like creating principals roles and principals, and a `catalog_admin` - someone who has access to catalog internal entities, like catalog roles, grants, tables, namespaces.

The tool is designed to be run with service_admin level permissions, but service_admin level access really only guarantees the ability to see that a catalog exists, but not necessarily modify anything internal to the catalog.

As such, we need to create a sort of super-root principal that has access to manage the catalog internals of every single catalog.

This principal should:

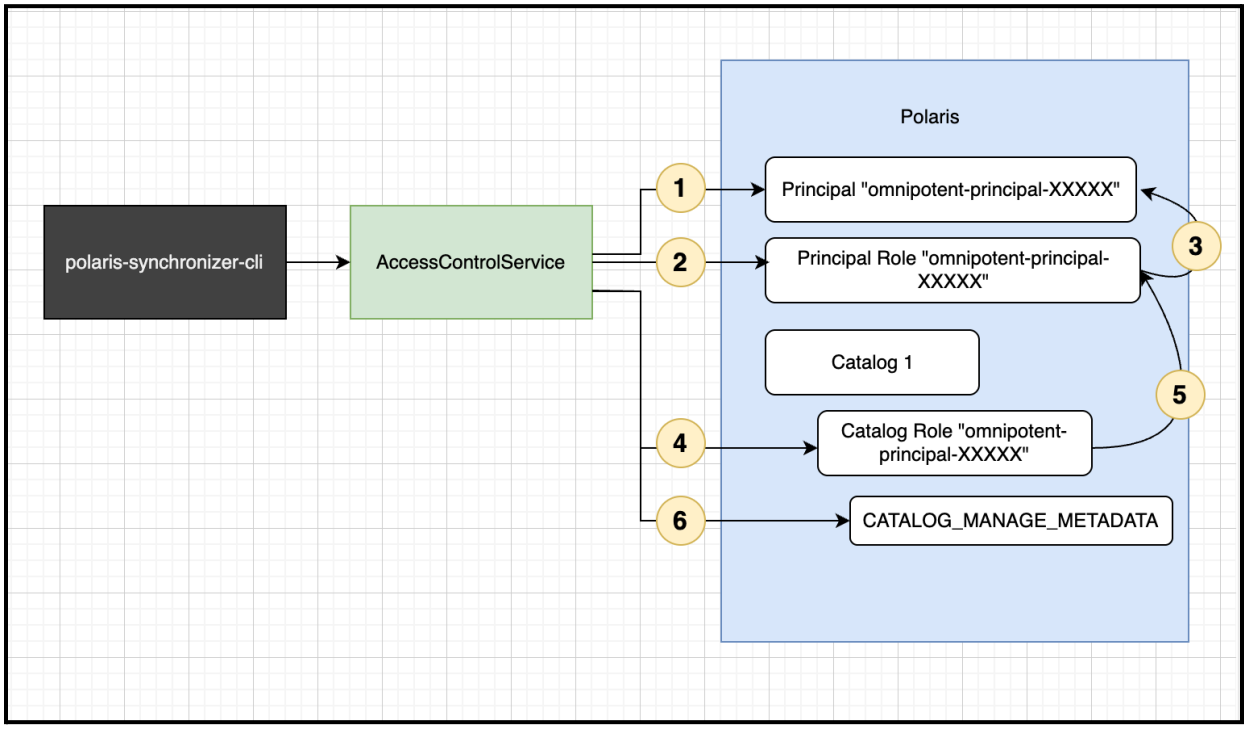
- Have read access to catalogs on the source Polaris instance
- Have write access to catalogs on the target Polaris instance
- Not conflict with any existing principal on either Polaris instance

This tool exposes a command, `create-omnipotent-principal`.
To decouple the creation of the omnipotent principal from the CLI, the tool introduces a manager class, `AccessControlService.` This command will, on a provided Polaris Instance:

1. Create/overwrite a new principal, omnipotent-principal-XXXXXX
2. Create/overwrite a new principal role, omnipotent-principal-XXXXXX,
3. Assign the principal role to the omnipotent principal

For each catalog:

4. Create/overwrite a catalog role, omnipotent-principal-XXXXXX
5. Assign the catalog role to the omnipotent principal role
6. Assign the following grants to that catalog role:
    - If created with write access: CATALOG_MANAGE_METADATA
    - If created with readonly access: CATALOG_READ_PROPERTIES, NAMESPACE_READ_PROPERTIES, NAMESPACE_LIST, TABLE_READ_PROPERTIES, TABLE_LIST, VIEW_READ_PROPERTIES, VIEW_LIST



To ensure that overwrite does not interfere with any user created entities and that the entities are not migrated over along with actual entities created on the source, we will use a property, IS_OMNIPOTENT_PRINCIPAL, to identify any omnipotent principal related entities for cleanup when this command is run/re-run. This ensures we never leave orphaned deployments of the tool.

This action will be a one-time setup, and will display the client credentials to stdout, like so:

```
=====================================================
Omnipotent Principal Credentials:
      name = omnipotent-principal-XXXXXX
      clientId = <client-id>
      clientSecret = <client-secret>
=====================================================
```

This will need to be performed both on the source and the target only once.

After this, during regular sync we may introduce more catalogs that do not have the omnipotent catalog role setup complete. To accommodate this, we will perform the following setup prior to migrating any entities under catalogs in the hierarchy:

- For each catalog:
    - Create a catalog role, omnipotent-principal-XXXXXX

    - Assign the following grants to that catalog role:
        - If target instance: CATALOG_MANAGE_METADATA
        - If source instance: CATALOG_READ_PROPERTIES, NAMESPACE_READ_PROPERTIES, NAMESPACE_LIST, TABLE_READ_PROPERTIES, TABLE_LIST, VIEW_READ_PROPERTIES, VIEW_LIST

Limitations:

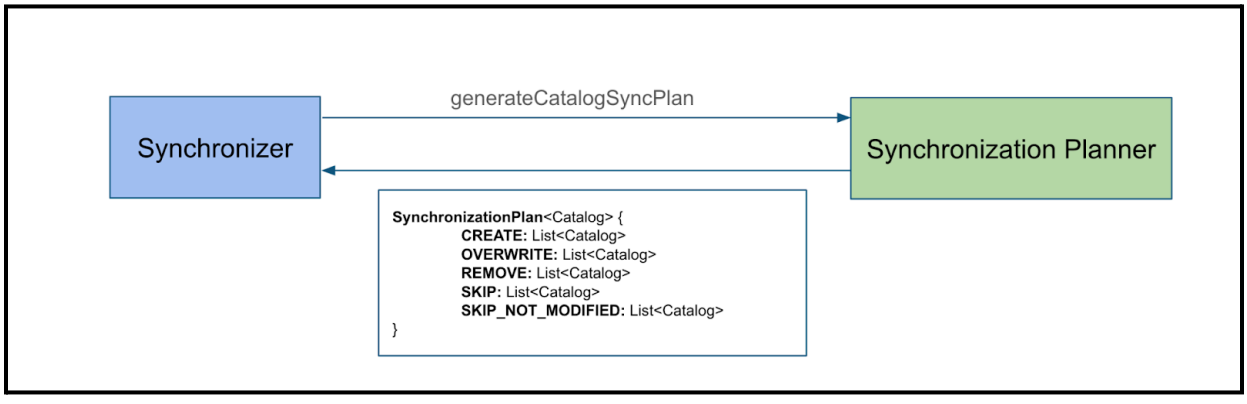
- While service_admin can list all catalogs, the service_admin still does not necessarily have the permissions to be able to create anything within a catalog. While the Polaris authorization engine does not provide a workaround to this, the tool is built to notify a user for which catalogs the creation of an omnipotent principal failed. For example:

```
None
INFO  - Encountered issues creating catalog roles for the following catalogs:
[catalog-1, catalog-2]
```

(2) Synchronization Planning

We want to support a configurable set of strategies for the synchronization tool because there may be a wide array of use cases, including supporting custom configuration. For example, a user may want to keep parity with the source, may just want to keep an append-only mode, migrate the entire instance instead of just the diff, etc. This means that we need to generate diff resolution strategies configurable in a way that is opaque to the actual synchronization that just takes these generated strategies and performs the actions associated with them.

To facilitate this, an interface, `SynchronizationPlanner`, has been introduced. This interface consumes all of the entities on the source of a particular type, all of the entities on the target of a particular type, and generates a `SynchronizationPlan<T>` that categorizes the entities into groupings of actions for the sync tool to perform.



The general structure of a synchronization plan is as follows:

```
SynchronizationPlan<EntityType> {
        [ACTION]: List<EntityType>,
        …
}
```

There are a few different actions we can take:

- **CREATE:** Should be used to signify the creation of entities that do not exist on the target, but should be created.
- **OVERWRITE:** Should be used to signify entities that exist both on the source and the target. These entities will be required to be dropped on the target first, then created on the target with the new entity from the source.
- **REMOVE:** Should be used to signify entities that exist on the target but not on the source. These entities should be cleaned up from the target as they may have been dropped on the source.
- **SKIP**
- **SKIP_AND_SKIP_CHILDREN:** Entities that we should skip the entire entity hierarchy of. For example, we do not want to modify the permissions for a catalog_admin or a service_admin.
- **SKIP_NOT_MODIFIED:** If we know an entity has not been modified on the source, and exists in the same state on the target, then we do not need to migrate the change.

There are a few different SynchronizationPlanner implementations built to exemplify why the pattern is useful. These can encapsulate each other with delegates and be built independent of the sync strategy being used. For example:

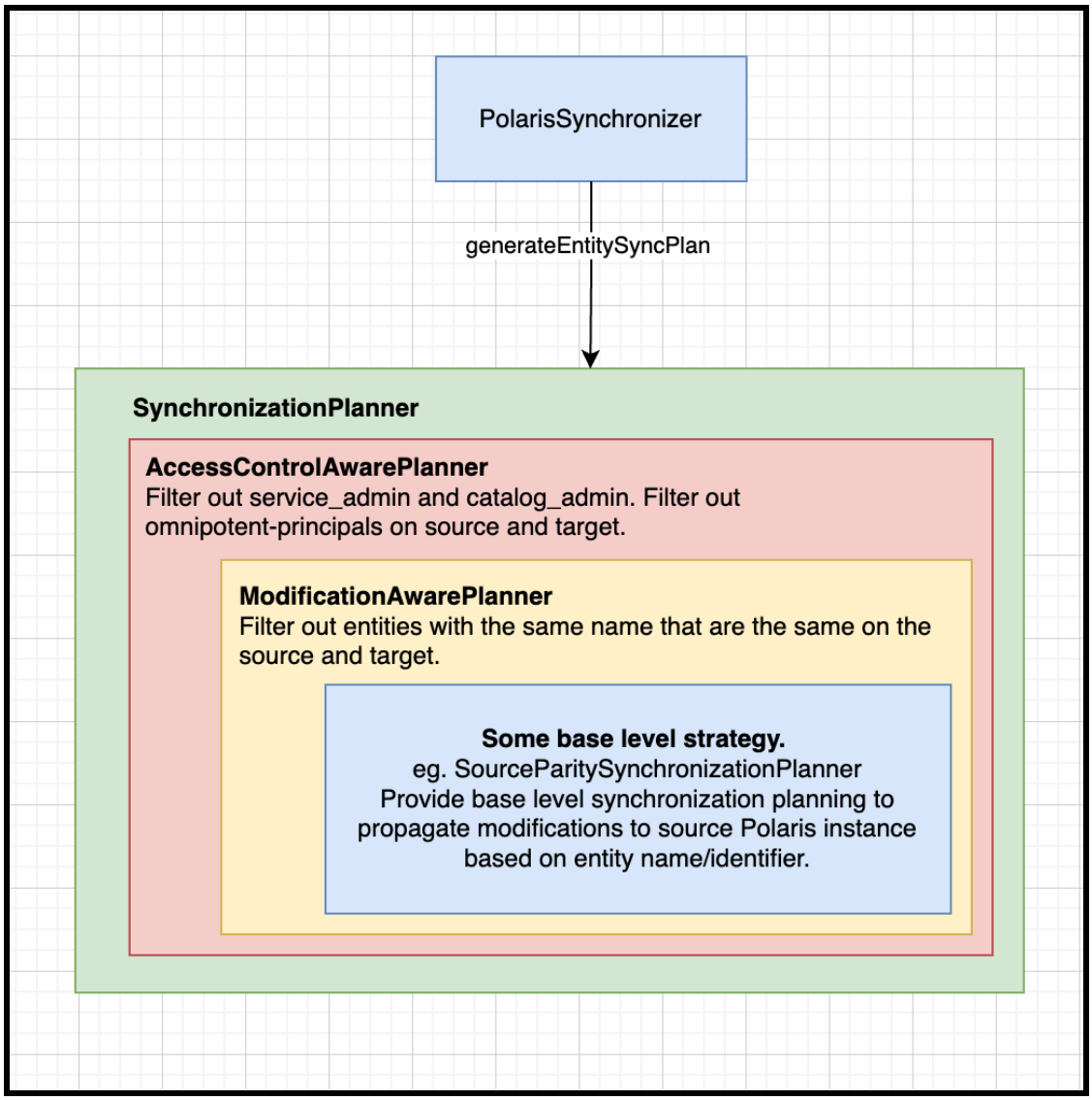
- **SourceParitySynchronizationPlanner:** This is the default synchronization planner that organizes out the entities to create, overwrite, and remove. It will take entities that exist only on the source, and put them in the CREATE list. It will take entities on both the source and target (no diffing, only based on the same identifier/name) and put them in the OVERWRITE list. It will take entities only on the target and put them in the REMOVE list.
- **AccessControlAwarePlanner:** Will filter out access control related mechanisms from the synchronization. For example, we do not want to copy the omnipotent principal from the source to the target. Nor do we want to remove the omnipotent principal on the target when it does not exist on the source. Furthermore, since the tool assumes service_admin, it will filter out service_admin related modifications. It will also filter out catalog_admin from overwrites, since the catalog_admin role cannot be dropped. This will put those entities into the SKIP list.
- **ModificationAwarePlanner:** Filters out entities by comparing their fields. This will put entities whose fields have not changed between the source and target into the SKIP_NOT_MODIFIED list. This is how we achieve incremental synchronization. This is

done by converting both source and target entities to a key-value map of properties. Then, a set of fields are excluded and the remaining fields are compared. This is required in case changes to fields are made in Polaris, then the tool need not be updated to reflect property comparisons, these changes will be picked up automatically. The exclusion list includes:

- For all entities
  - `createTimestamp`
  - `lastUpdateTimestamp`
  - `entityVersion`
- For catalogs, we also need to omit some information in the StorageConfigInfo because this is set by Polaris based on environment configurations, and thus will always be different from source and target
  - S3
    - `storageConfigInfo.userArn`
  - Azure
    - `storageConfigInfo.multiTenantAppName`
    - `storageConfigInfo.consentUrl`
  - GCP
    - `storageConfigInfo.gcsServiceAccount`

There are other planners and cli options with demonstrable use cases that we can introduce seamlessly to make this configurable in the future. The eventual roadmap is to be able to pass in any implementqation from the CLI. For example,

- Add a planner that does not modify the target at all except to create entities, essentially performing a No-Op on create and remove. This could better fit bulk migration use cases, or partial synchronization, perhaps some entities are being actively written to on the target and should not be modified.
- Add a planner that filters by entity name. For example, a given user of the tool may want to migrate only certain catalogs and provide configuration to filter which catalogs or table identifiers to migrate.



(3) Performing the Migration

Once the synchronization plan for a particular entity is generated, the PolarisSynchronizer will carry out the actions related to that sync plan. The only modification actions from the SynchronizationPlan that the tool will perform are:

- CREATE
- OVERWRITE
- REMOVE

In the case of an overwrite, if the entity cannot be dropped without removing child entities, then a cascading drop will be performed. For example, for catalogs, all catalog-roles (except for the catalog_admin role) and namespaces have to be dropped first.

This is the following order that migration is performed in:

- Principal Roles
- Catalogs
- For each Catalog:
    - Catalog Roles
    - Namespaces
    - For each namespace:
        - Nested namespaces
            - …
        - Tables
    - For each catalog role:
        - Assignments to principal roles (these depend on principal roles being created)
        - Grants (these depend on tables and namespaces being created first)

For example, we may have the following result of the table discovery stage:

Source	Target
ns	ns
└─table1	└─table1
└─table2	
~~table3~~	table3
table4	table4
newTableName	oldTableName

Here, we can see that there are some differences between the catalogs caused by certain changes made to the source catalog:

- ns.table2 was created on the source catalog
- table3 was removed on the source catalog
- oldTableName was renamed on the source catalog to newTableName

After running the tool with the SourceParitySynchronizationPlanner, we will perform the following actions:

- ns.table2 will be created on the target catalog

- `newTableName` will be created on the target catalog
- `table3` will be removed on the target catalog
- `oldTableName` will be removed on the target catalog

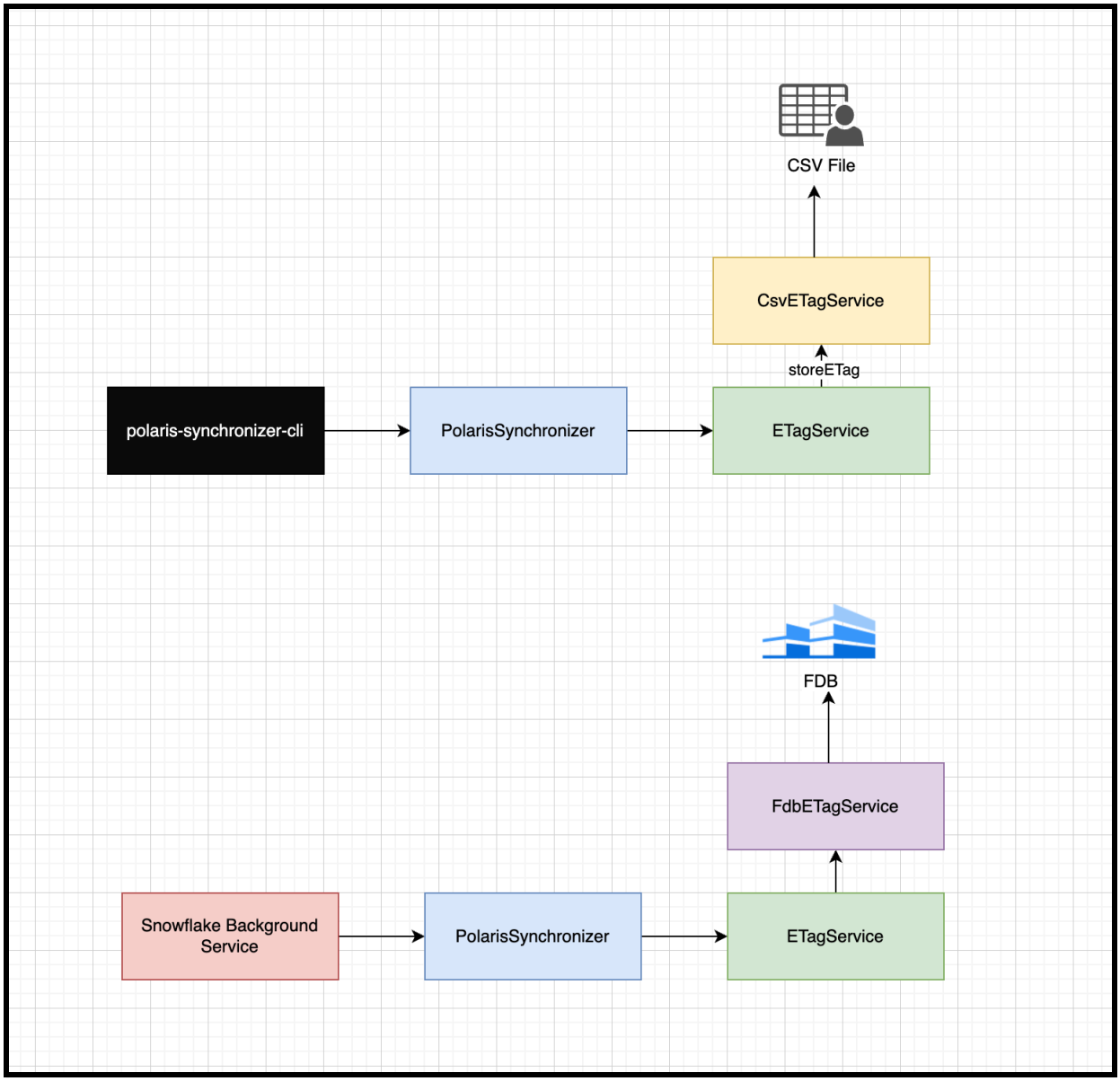
Incremental Table Migration

Since table migration will be the most taxing migration to perform as it requires issuing an Iceberg loadTable for each table in a catalog, the tool has built in support to accommodate ETag aware table loading.

There was a change introduced in Iceberg release 1.8.0 that introduced freshness aware table loading. The idea behind this is that on an initial loadTable, the catalog can provide the HTTP ETag header to specify a specific version of the metadata. Then, a subsequent loadTable can specify this ETag value in the HTTP If-None-Match header, and the server can respond with a 304 NOT MODIFIED if the table metadata has not changed.

The support for this logic has been introduced to Polaris in this pull request. However, enabling its use is dependent on upgrading the version of the Iceberg REST spec in the Polaris repository.

While PolarisSynchronizer itself is designed to be stateless, the tool does need a strategy to store ETags. For the CLI tool, this can be within a CSV file. However, for example if this were to run in-process in a background service in Snowflake, we may need to persist these to a store like FDB. To decouple storage and retrieval of ETags from the persistence store, this tool introduces an interface, `ETagManager`, that an implementing class can provide different persistence functionality for.



Migrating Principals & Synergy with an External IDP (Exploratory)

Migrating principals results in the client credentials for that principal being reset on the target Polaris instance. However, having this functionality available is important as it is nontrivial to manually migrate principals and their assignment to principal roles. For this reason, migration of principals and their assignments to principal roles is gated behind a flag, --sync-principals.

In order to expose the new credentials so that someone running the tool can update client credentials for their workloads, the tool logs the client credentials from the target to stdout.

There is a warning printed to stdout and in the tool's help text ensuring that the security concerns of this approach are clear for anyone choosing to enable principal migration.

Example:

```
None
WARN  - Principal migration will reset credentials on the target Polaris
instance. Principal migration will log the new target Principal credentials to
stdout.

...

INFO  - Overwrote principal principal-2 on target. Target credentials:
<client-id>:<client-secret> - 1/1
```

The following section is in the exploratory phase. Some ambiguity needs to be resolved and testing needs to be performed to verify that this flow will work with Polaris.

Limitation: It may be that `federated=True` is an internal property, in which case we cannot observe it through the API and it would not be copied from source to target by the tool.

Principal migration not being supported is still a major pain point of the tool and can lead to a significant increase in the amount of manual work needed to move between Polaris deployments. The proposed workaround is storing all principals in an External IDP like Okta.

For example, all identities can be managed in Polaris with Okta where Users are represented as Principals, and Groups are represented as Principal Roles. Users can belong to many Groups, which carries over as Principal to Principal Role assignment in Polaris.

With an external IDP, these Okta identity entities have entities lazily created at the first time of authentication in Polaris with a property `federated=True.`

The existing tool will carry over entities exactly as they are with all properties. As such, when the tool is run, it will carry over this property for principal roles to the target. Then, the tool can assign catalog roles to the principal roles, and the external IDP can handle principal creation and assignment of the principal roles that correspond to groups in the external IDP.

The flow looks as follows:

1. The tool will create the principal role with `federated=True` on the target Polaris instance.

2. The tool will assign the relevant catalog roles to the federated principal role on the target Polaris instance.

   Upon first time of authentication on the target Polaris instance, Polaris will

3. Create a principal with `federated=True`
4. Attempt to create a principal role with `federated=True`, but see that it already exists
5. Assign the federated principal role to the federated principal