Snowpark Container Services: Additional considerations for services

FEATURE — GENERALLY AVAILABLE

Available to accounts in AWS and Microsoft Azure commercial regions, with some exceptions. For more information, see Available regions.

Connecting to Snowflake from inside a container

When you start a service (including job services), Snowflake provides credentials to the running containers, enabling your container code to use drivers for connecting to Snowflake and executing SQL (similar to any other code on your computer connecting to Snowflake). The provided credentials authenticate as the owner role (the role that created the service). Snowflake provides some of the information as environment variables in your containers.

Every object in Snowflake has an owner role. The service's owner role determines what capabilities your service is allowed to perform when interacting with Snowflake. This includes executing SQL, accessing stages, and service-to-service networking.

Note
A service's owner role refers to the role that created the service. You can also define one or more service roles to manage access to the endpoints the service exposes. For more information, see Managing access to service endpoints.

When you create a service, Snowflake also creates a *service user* specific to that service. When the service runs a query, it runs the query as the service user, using the service's owner role. What the service user can do is determined by this owner role.

The service's owner role cannot be any of the privileged roles, such as ACCOUNTADMIN, SECURITYADMIN, and ORGADMIN. This is to limit what a misbehaving service can do, requiring customers to be more intentional to enable a service to perform administrative operations.

To view the queries issued by a specific service user, you can use the ACCOUNTADMIN role to view the query history. The user name of the service user appears in the following forms:

- For a service created before the 8.35 server release, the service user name is of the format `SF$SERVICE$<unique-id>`.
- For a service created after the 8.35 server release, the service user name is the same as the service name.

Connecting to Snowflake

Snowflake provides the following environment variables for you to configure a Snowflake client in your service code:

- **SNOWFLAKE_ACCOUNT:** Provides the account locator for the Snowflake account the service is currently running under.
- **SNOWFLAKE_HOST:** Provides the hostname used to connect to Snowflake.

Snowflake also provides an OAuth token in the container in a file named `/snowflake/session/token`. When creating a connection, you provide this token to the connector.

When creating a connection to Snowflake from a container, you must use SNOWFLAKE_HOST, SNOWFLAKE_ACCOUNT, and the OAuth token. You cannot use the OAuth token without also using SNOWFLAKE_HOST, and you cannot use the OAuth token outside Snowpark Container Services. For more information, see Using an OAuth token to execute SQL.

Note
Using external access integrations to access Snowflake can mean sending potentially sensitive information over the internet. Whenever possible, services should use the provided OAuth token to access the SNOWFLAKE_HOST hostname. This avoids the need to access Snowflake from the Internet.

For sample code using various Snowflake drivers, see Snowflake Connection Samples.

Examples

- In Tutorial 2 (see `main.py`), the code reads the environment variables as follows:

```
SNOWFLAKE_ACCOUNT = os.getenv('SNOWFLAKE_ACCOUNT')
SNOWFLAKE_HOST = os.getenv('SNOWFLAKE_HOST')
```

The code passes these variables to a connection creation code for the Snowflake client of choice. The container uses these credentials to create a new session, with the owner role as the primary role, to execute queries. The following example code is the minimum needed to create a Snowflake connection in Python:

```
def get_login_token():
  with open('/snowflake/session/token', 'r') as f:
    return f.read()

conn = snowflake.connector.connect(
  host = os.getenv('SNOWFLAKE_HOST'),
  account = os.getenv('SNOWFLAKE_ACCOUNT'),
  token = get_login_token(),
  authenticator = 'oauth'
)
```

- When you use the default host (that is, you don't include the `host` argument when creating a connection), connecting to Snowflake using other forms of authentication (such as username and password) is supported. For example, the following connection specifies the user name and password to authenticate:

```
conn = snowflake.connector.connect(
  account = os.getenv('SNOWFLAKE_ACCOUNT'),
  user = '<user-name>',
  password = <password>
)
```

Using a default host requires external access integration with a network rule allowing access from your service to the Snowflake hostname for your account. For example, if your account name is *MyAccount*, the hostname will be *myaccount.snowflakecomputing.com*. For more information, see Configuring network egress.

- Create a network rule that matches your account's Snowflake API hostname:

```
CREATE OR REPLACE NETWORK RULE snowflake_egress_access
  MODE = EGRESS
```

```
    TYPE = HOST_PORT
    VALUE_LIST = ('myaccount.snowflakecomputing.com');
```

- Create an integration using the preceding network rule:

```
CREATE EXTERNAL ACCESS INTEGRATION snowflake_egress_access_integrat
  ALLOWED_NETWORK_RULES = (snowflake_egress_access)
  ENABLED = true;
```

Configuring the database and schema context for executing SQL

This section explains two concepts:

- The logic Snowflake uses to determine the database and schema in which to create your service.
- The method through which Snowflake conveys this information to your containers, thus enabling the container code to execute SQL in the same database and schema context.

Snowflake uses the service name to determine the database and schema in which to create a service:

- Example 1: In the following CREATE SERVICE and EXECUTE JOB SERVICE commands, the service name does not explicitly specify a database and schema name. Snowflake creates the service and the job service in the current database and schema.

```
-- Create a service.
CREATE SERVICE test_service IN COMPUTE POOL ...

-- Execute a job service.
EXECUTE JOB SERVICE
  IN COMPUTE POOL tutorial_compute_pool
  NAME = example_job_service ...
```

- Example 2: In the following CREATE SERVICE and EXECUTE JOB SERVICE commands, the service name includes a database and schema name. Snowflake creates the service and job service in the specified database (`test_db`) and schema (`test_schema`), regardless of the current schema.

```
-- Create a service.
CREATE SERVICE test_db.test_schema.test_service IN COMPUTE POOL ...

-- Execute a job service.
EXECUTE JOB SERVICE
  IN COMPUTE POOL tutorial_compute_pool
  NAME = test_db.test_schema.example_job_service ...
```

When Snowflake starts a service, it provides the database and schema information to the running containers using the following environment variables:

- SNOWFLAKE_DATABASE
- SNOWFLAKE_SCHEMA

Your container code can use environment variables in the connection code to determine which database and schema to use, as shown in this example:

```
conn = snowflake.connector.connect(
  host = os.getenv('SNOWFLAKE_HOST'),
  account = os.getenv('SNOWFLAKE_ACCOUNT'),
  token = get_login_token(),
  authenticator = 'oauth',
  database = os.getenv('SNOWFLAKE_DATABASE'),
  schema = os.getenv('SNOWFLAKE_SCHEMA')
)
```

Example

In Tutorial 2, you create a Snowflake job service that connects with Snowflake and executes SQL statements. The following steps summarize how the tutorial code uses the environment variables:

1. In the common setup (see the Common Setup section), you create resources, including a database and a schema. You also set the current database and schema for the session:

   ```
   USE DATABASE tutorial_db;
   ...
   USE SCHEMA data_schema;
   ```

2. After you create a job service (by running EXECUTE JOB SERVICE), Snowflake starts the container and sets the following environment variables in the container to the current database and schema of the session:

   - SNOWFLAKE_DATABASE is set to "TUTORIAL_DB"
   - SNOWFLAKE_SCHEMA is set to "DATA_SCHEMA"

3. The job code (see `main.py` in Tutorial 2) reads these environment variables:

```
SNOWFLAKE_DATABASE = os.getenv('SNOWFLAKE_DATABASE')
SNOWFLAKE_SCHEMA = os.getenv('SNOWFLAKE_SCHEMA')
```

4. The job code sets the database and schema as the context in which to execute the SQL statements (`run_job()` function in `main.py`):

```
{
   "account": SNOWFLAKE_ACCOUNT,
   "host": SNOWFLAKE_HOST,
   "authenticator": "oauth",
   "token": get_login_token(),
   "warehouse": SNOWFLAKE_WAREHOUSE,
   "database": SNOWFLAKE_DATABASE,
   "schema": SNOWFLAKE_SCHEMA
}
...
```

   Note
   SNOWFLAKE_ACCOUNT, SNOWFLAKE_HOST, SNOWFLAKE_DATABASE, SNOWFLAKE_SCHEMA are environment variables that Snowflake generates for the application container, but SNOWFLAKE_WAREHOUSE is not (the Tutorial 2 application code created this variable because Snowflake does not pass a warehouse name to a container).

Specifying the warehouse for your container

If your service connects to Snowflake to execute a query in a Snowflake warehouse, you have the following options to specify a warehouse:

- **Specify a warehouse in your application code.** Specify a warehouse as part of the connection configuration when starting a Snowflake session to run queries in your code. For an example, see Tutorial 2.

- **Specify a default warehouse when creating a service.** Specify the optional QUERY_WAREHOUSE parameter in the CREATE SERVICE (or EXECUTE JOB SERVICE) command to provide a default warehouse. If your application code does not provide a warehouse as part of connection configuration, Snowflake uses the default warehouse. Use the ALTER SERVICE command to change the default warehouse.

If you specify a warehouse using both methods, the warehouse specified in the application code is used.

Using an OAuth token to execute SQL

All Snowflake-provided clients support OAuth as a way to authenticate. Service containers also use the OAuth mechanism to authenticate with Snowflake. For example, when a container wants to execute SQL, the container creates a connection to Snowflake, similar to any other Snowflake client:

```
def get_login_token():
  with open('/snowflake/session/token', 'r') as f:
    return f.read()

conn = snowflake.connector.connect(
  host = os.getenv('SNOWFLAKE_HOST'),
  account = os.getenv('SNOWFLAKE_ACCOUNT'),
  token = get_login_token(),
  authenticator = 'oauth'
)
```

When you create a service, Snowflake runs the containers and provides an Oauth token for containers to use at the following location within the container: `/snowflake/session/token`.

Note the following details about this OAuth token:

- You should read the contents of the `/snowflake/session/token` file immediately before use because the content expires in 10 minutes, and Snowflake refreshes this file every few minutes. After a container connects to Snowflake successfully, the expiration time does not apply to the connection (as with any sessions that users create directly).

- This OAuth token is valid only within the specific Snowflake service. You cannot copy the OAuth token and use it outside the service.
- Using the OAuth token, the containers connect with Snowflake as the service user and use the service's owner role.
- Using the OAuth token to connect will create a new session. The OAuth token is not associated with any existing SQL session.

Note
A significant difference between executing stored procedures and executing a service is that stored procedures run in the same session as the SQL that runs them. But every time a container establishes a new connection, you are creating a new session.

Connecting to Snowflake from inside a container using caller's rights

PREVIEW FEATURE — OPEN

Configuring caller's rights for your service is a preview feature.

Containers execute queries by connecting to Snowflake as the service user and using the service's owner role that grants privileges to the service. In certain application scenarios, you might need to execute queries using the context of the end user rather than the service user. The caller's right feature is used in this context.

For example, suppose you create a service that exposes a public endpoint to provide a web application that displays a dashboard using data stored in Snowflake. You grant other users in your Snowflake account access to the dashboard (by granting the service role to those users). When any of these end users log in to the dashboard, you want the dashboard to display only the data the user has access to.

However, because containers by default execute queries using service user and service's owner role, the dashboard shows the data the service's owner role has access to, regardless of which end user connected to the endpoint. As a result, the data in the user's dashboard is not limited to what the end user has access to, which allows the user to access data that they should not be allowed to access.

To limit the dashboard to show only data accessible to the logged in user, the application containers need to execute SQL using privileges granted to the end user. You can enable this by using caller's rights in the application.

Note

- The caller's rights feature is supported only when accessing a service using network ingress. The feature is not available when using a service function to access the service.
- The caller's rights feature is currently not supported in a Snowflake Native App (apps with containers).

Configuring caller's rights for your service

Configuring caller's rights for your application is a two-step process.

1. Set `executeAsCaller` to `true` in the service specification as shown in the following specification fragment:

   ```
   spec:
     containers:
     ...
   capabilities:
     securityContext:
       executeAsCaller: true
   ```

   This explicitly tells Snowflake that the application intends to use caller's rights and causes Snowflake to insert the `Sf-Context-Current-User-Token` header in every incoming request before sending the request to the application container. This user token facilitates query execution as the calling user. If not specified, `executeAsCaller` defaults to `false`.

   Specifying the `executeAsCaller` option does not affect the service's ability to execute queries as the service user and service's owner role. With `executeAsCaller` enabled, the service has the option to connect to Snowflake both as a calling user and as a service user.

2. Update your application code. To establish a Snowflake connection on behalf of the calling user, update your code to create a login token that includes both the OAuth token that Snowflake provided to the service and the user token from the `Sf-Context-Current-User-Token` header. The login token should follow this format: `<service-oauth-token>.<Sf-Context-Current-User-Token>`.

   This is demonstrated in the following Python code fragment:

```
# Environment variables below will be automatically populated by Snowf
SNOWFLAKE_ACCOUNT = os.getenv("SNOWFLAKE_ACCOUNT")
SNOWFLAKE_HOST = os.getenv("SNOWFLAKE_HOST")

def get_login_token():
    with open("/snowflake/session/token", "r") as f:
        return f.read()

def get_connection_params(ingress_user_token = None):
    # start a Snowflake session as ingress user
    # (if user token header provided)
    if ingress_user_token:
        logger.info("Creating a session on behalf of the current user.
        token = get_login_token() + "." + ingress_user_token
    else:
        logger.info("Creating a session as the service user.")
        token = get_login_token()

    return {
        "account": SNOWFLAKE_ACCOUNT,
        "host": SNOWFLAKE_HOST,
        "authenticator": "oauth",
        "token": token
    }

def run_query(request, query):
    ingress_user_token = request.headers.get('Sf-Context-Current-User-
    # ingress_user_token is None if header not present
    connection_params = get_connection_params(ingress_user_token)
    with Session.builder.configs(connection_params).create() as sessio
      # use the session to execute a query.
```

In the example above:

- The `get_login_token` function reads the file where Snowflake copied the OAuth token for the container to use.
- The `get_connection_params` function constructs a token by concatenating of the OAuth token and the user token from the `Sf-Context-Current-User-Token` header. The function includes this token in a dictionary of parameters that the application uses to connect to Snowflake.

For an example with step-by-step instructions, see Create a service with caller’s rights enabled.

Accessing a service with caller's rights configured

Configuring caller's rights is about your service establishing a Snowflake connection on behalf of the caller. How you log in to the service's ingress endpoints (programmatically or using a browser) remains the same. After login, the following apply:

- Accessing a public endpoint using a browser: After you log into an endpoint, the service establishes a connection to Snowflake on behalf of the calling user using the default role of the user. If there is no default role configured for the user, the PUBLIC role is used.
- Accessing a public endpoint programmatically: When logging into an endpoint programmatically using JWT token, you can optionally set the `scope` parameter to specify the role to activate

Currently, after a service establishes a caller's right connection to Snowflake on behalf of the caller, switching roles is not supported. If your application needs to use different roles to access different objects, set up the user to have all secondary roles active by default. To do this, use the ALTER USER command to set the DEFAULT_SECONDARY_ROLES property of the user to ('ALL'):

```
ALTER USER my_user SET DEFAULT_SECONDARY_ROLES = ( 'ALL' );
```

Managing caller grants to a service

When a service creates a caller's rights session, the session operates as the calling user (not as the service user). When an operation is performed using this session, Snowflake applies two permissions checks:

1. The first permissions check is as if the user created the session directly. These are the normal permission checks that Snowflake performs for the user.
2. The second permissions check verifies the service is allowed to perform the operation on behalf of a user. Snowflake verifies this by ensuring that the service owner role has been granted the necessary caller grants.

In a caller's rights session, both the normal permission check and the service owner role's caller grants check must allow the operation; this is referred to as restricted caller's rights. By default the service has no permission to do anything on behalf of a user. You must explicitly grant caller grants to the service so it can run with the caller's privileges.

For example, suppose a user `U1` uses a role `R1`, that has the SELECT privilege on the table `T1`. When `U1` logs into the public endpoint of your service (`example_service`), which is configured to use the caller's rights, the service then establishes a connection with Snowflake on behalf of `U1`.

To allow the service to query table `T1` on behalf of `U1`, you need to grant the service's owner role the following privileges:

1. Privileges to resolve the table's name, by granting a caller grant that allows the service to run with the USAGE privilege on the database and schema for that table.
2. Privileges to use a warehouse to execute queries by granting a caller grant that allows the service to run with the USAGE privilege on a warehouse.
3. Privileges to query the table by granting a caller grant that allows the service to run with the SELECT privilege on table `T1`.

```
-- Permissions to resolve the table's name.
GRANT CALLER USAGE ON DATABASE <db_name> TO ROLE <service_owner_role>;
GRANT CALLER USAGE ON SCHEMA <schema_name> TO ROLE <service_owner_role>;
-- Permissions to use a warehouse
GRANT CALLER USAGE ON WAREHOUSE <warehouse_name> TO ROLE <service_owner_ro
-- Permissions to query the table.
GRANT CALLER SELECT ON TABLE T1 TO ROLE <service_owner_role>;
```

Any role in your account that has the global MANAGE CALLER GRANT privilege can grant caller grants. For more information about caller grants, see GRANT CALLER and Restricted caller's rights.

Example

An example of a service that uses the caller's rights feature when executing SQL queries on behalf of the users is provided. For more information, see Create a service with caller's rights enabled.

Configuring network ingress

To allow anything to interact with your service from the internet, you declare the network ports on which your service is listening as endpoints in the service specification file. These endpoints control ingress.

By default, service endpoints are private. Only service functions and service-to-service communications can make requests to the private endpoints. You can declare an endpoint as public to allow requests to an endpoint from the internet. The service's owner role must have the BIND SERVICE ENDPOINT privilege on the account.

```
endpoints:
- name: <endpoint name>
  port: <port number>
  protocol : < TCP / HTTP / HTTPS >
  public: true
```

For an example, see Tutorial 1.

Note
Currently only services, not job services, support network ingress.

Ingress and web app security

You can create a Snowpark Container Services service for web hosting using the public endpoint support (network ingress). For added security, Snowflake employs a proxy service to monitor incoming requests from clients to your service and outgoing responses from your service to the clients. This section explains what the proxy does and how it impacts a service deployed to Snowpark Container Services.

Note
When you test a service locally, you are not using the Snowflake proxy and therefore there will be differences between your experience running a service locally as opposed to when deployed in Snowpark Container Services. Review this section and update your local setup for better testing.

For example:

- The proxy does not forward an incoming HTTP request if the request uses a banned HTTP method.
- The proxy sends a 403 response to the client if the Content-Type header in the response indicates that the response contains an executable.

Additionally, the proxy can also inject new headers and alter existing headers in the request and the response, with your container and data security in mind.

For example, upon receiving a request, your service might send HTML, JavaScript, CSS, and other content for a web page to the client browser in the response. The web page in the browser is part of your service, acting as the user interface. For security reasons, if your service has restrictions (such as a restriction on making network connections to other sites), you might also want the web page for your service to have the same restrictions.

By default, services have limited permissions to access the internet. The browser should also restrict the client app from accessing the internet and potentially sharing data in most cases. If you set up an External Access Integration (EAI) to allow your service to access `example.com` (see Configuring network egress), the web page for your service should also be able to access `example.com` through your browser.

The Snowflake proxy applies the same network restrictions on the service and the web page by adding a `Content-Security-Policy` (CSP) header in the response. By default, the proxy adds a baseline CSP in the response to protect against common security threats. Browser security is a best effort to balance functionality with security, it is a shared responsibility to ensure this baseline is appropriate for your use case. In addition, if your service is configured to use an EAI, the proxy applies the same network rules from the EAI to the CSP for the web page. This CSP enables the web page in the browser to access the same sites that the service can access.

The following sections explain how the Snowflake proxy handles incoming requests for your service and modifies the outgoing responses from your service to the clients.

Requests incoming to the service

When a request arrives, the proxy does the following before forwarding the request to the service:

- **Incoming requests with banned HTTP methods:** If an incoming HTTP request uses any of the following banned HTTP methods, the proxy does not forward the request to your service:
  - `TRACE`
  - `OPTIONS`
  - `CONNECT`

Responses outgoing to the clients

The Snowflake proxy applies these modifications to the response sent by your service before forwarding the response to the client.

- **Header Scrubbing:** Snowflake proxy removes these response headers, if present:
  - `X-XSS-Protection`
  - `Server`
  - `X-Powered-By`
  - `Public-Key-Pins`
- **Content-Type response header:** If your service response includes the Content-Type header with any of the following MIME type values (that indicate an executable), Snowflake proxy does not forward that response to the client. Instead, the proxy sends a `403 Forbidden` response.
  - `application/x-msdownload`: Microsoft executable.
  - `application/exe`: Generic executable.
  - `application/x-exe`: Another generic executable.
  - `application/dos-exe`: DOS executable.
  - `application/x-winexe`: Windows executable.
  - `application/msdos-windows`: MS-DOS Windows executable.
  - `application/x-msdos-program`: MS-DOS executable.
  - `application/x-sh`: Unix shell script.
  - `application/x-bsh`: Bourne shell script.
  - `application/x-csh`: C shell script.
  - `application/x-tcsh`: Tcsh shell script.
  - `application/batch`: Windows batch file.
- **X-Frame-Options response header:** To prevent clickjacking attacks, the Snowflake proxy sets this response header to `DENY`, preventing other web pages from using an iframe to the web page for your service.
- **Cross-Origin-Opener-Policy (COOP) response header:** Snowflake sets the COOP response header to `same-origin` to prevent referring cross-origin windows from accessing your service tab.
- **Cross-Origin-Resource-Policy (CORP) response header:** Snowflake sets the CORP header to `same-origin` to prevent external sites from loading resources exposed by

the ingress endpoint (for example, in an iframe).

- **X-Content-Type-Options response header:** Snowflake proxy sets this header to `nosniff` to ensure the clients do not change the MIME type stated in the response by your service.
- **Cross-Origin-Embedder-Policy (COEP) response header:** Snowflake proxy sets the COEP response header to `credentialless`, which means when loading a cross-origin object such as an image or a script, if the remote object does not support Cross-Origin Resource Sharing (CORS) protocol, Snowflake does not send the credentials when loading it.
- **Content-Security-Policy-Report-Only response header:** Snowflake proxy replaces this response header with a new value directing the client to send the CSP reports to Snowflake.
- **Content-Security-Policy (CSP) response header:** By default the Snowflake proxy adds the following baseline CSP to protect against common web attacks.

  ```
  default-src 'self' 'unsafe-inline' 'unsafe-eval' blob: data:; object-sı
  ```

  There are two content security policy considerations:

  - In addition to the baseline content security policy that proxy adds, the service itself can explicitly add a CSP in the response. A service might choose to enhance security by adding a stricter CSP. For example, a service might add the following CSP to allow scripts only from `self`.

    ```
    script-src 'self'
    ```

    In the resulting response sent to the client, there will be two CSP headers. Upon receiving the response, the client browsers then apply the strictest content security policy that includes the additional restrictions specified by each policy.

  - If you configure an External Access Integration (EAI) to allow your service to access an external site (Configuring network egress), the Snowflake proxy creates a CSP that allows your web page to access that site. For example, suppose a network rule associated with an EAI allows your service egress access to `example.com`. Then, Snowflake proxy adds this CSP response header:

```
default-src 'self' 'unsafe-inline' 'unsafe-eval' http://example.con
```

Browsers honor the content access policy received in the response. In this example, browsers allow the app access to `example.com` but not other sites.

Configuring network egress

Your application code might require access to the internet. By default, application containers don't have permission to access the internet. You need to enable internet access using External Access Integrations (EAIs).

Typically, you want an account administrator to create EAIs to manage external access allowed from services (including job services). Account administrators can then grant EAI usage to specific roles that developers use to run services.

The following example outlines the steps in creating an EAI that allows egress traffic to specific destinations specified using network rules. You then refer to the EAI when creating a service to allow requests to specific internet destinations.

Example

Suppose you want your application code to send requests to the following destinations:

- HTTPS requests to translation.googleapis.com
- HTTP and HTTPS requests to google.com

Follow these steps to enable your service to access these domains on the internet:

1. Create an External Access Integration (EAI). This requires appropriate permissions. For example, you can use ACCOUNTADMIN role to create an EAI. This is a two-step process:
   a. Use the CREATE NETWORK RULE command to create one or more egress network rules listing external destinations you want to allow access to. You can accomplish this example with one network rule, but for illustration, we create two network rules:
      i. Create a network rule named `translate_network_rule`:

```
CREATE OR REPLACE NETWORK RULE translate_network_rule
  MODE = EGRESS
```

```
    TYPE = HOST_PORT
    VALUE_LIST = ('translation.googleapis.com');
```

This rule allows TCP connections to the `translation.googleapis.com` destination. The domain in the VALUE_LIST property does not specify the optional port number, so the default port 443 (HTTPS) is assumed. This allows your application to connect to any URL that starts with `https://translation.googleapis.com/`.

ii. Create a network rule named `google_network_rule`:

```
CREATE OR REPLACE NETWORK RULE google_network_rule
  MODE = EGRESS
  TYPE = HOST_PORT
  VALUE_LIST = ('google.com:80', 'google.com:443');
```

This allows your application to connect to any URL that starts with `http://google.com/` or `https://google.com/`.

Note

For the `VALUE_LIST` parameter, you must provide a full host name. Wildcards (for example, `*.googleapis.com`) are not supported.

Snowpark Container Services supports only the network rules that allow ports 22, 80, 443, and 1024+. If a network rule referenced allows access to other ports, creation of the service will fail. Contact your account representative if you require use of additional ports.

Note

To allow your service to send HTTP or HTTPS requests to any destination on the internet, you specify “0.0.0.0” as the domain in the VALUE_LIST property. The following network rule allows sending both “HTTP” and “HTTPS” requests anywhere on the internet. Only ports 80 or 443 are supported with “0.0.0.0”.

```
CREATE NETWORK RULE allow_all_rule
  TYPE = 'HOST_PORT'
```

```
    MODE= 'EGRESS'
    VALUE_LIST = ('0.0.0.0:443','0.0.0.0:80');
```

b. Create an external access integration (EAI) that specifies that the preceding two egress network rules are allowed:

```
CREATE EXTERNAL ACCESS INTEGRATION google_apis_access_integration
  ALLOWED_NETWORK_RULES = (translate_network_rule, google_network_r
  ENABLED = true;
```

Now the account admin can grant usage of the integration to developers to allow them to run a service that can access specific destinations on the internet.

```
GRANT USAGE ON INTEGRATION google_apis_access_integration TO ROLE t
```

2. Create the service by providing the EAI as shown in the following examples. The owner role that is creating the service needs the USAGE privilege on the EAI and READ privilege on the secrets referenced. Note that you cannot use the ACCOUNTADMIN role to create a service.

- Create a service:

```
USE ROLE test_role;

CREATE SERVICE eai_service
  IN COMPUTE POOL MYPOOL
  EXTERNAL_ACCESS_INTEGRATIONS = (GOOGLE_APIS_ACCESS_INTEGRATION)
  FROM SPECIFICATION
  $$
  spec:
    containers:
      - name: main
        image: /db/data_schema/tutorial_repository/my_echo_service_
        env:
          TEST_FILE_STAGE: source_stage/test_file
        args:
          - read_secret.py
    endpoints:
      - name: read
```

```
        port: 8080
  $$;
```

This example CREATE SERVICE request uses an inline service specification and specifies the optional EXTERNAL_ACCESS_INTEGRATIONS property to include the EAI. The EAI specifies the network rules that allow egress traffic from the service to the specific destinations.

- Execute a job service:

```
EXECUTE JOB SERVICE
  IN COMPUTE POOL tt_cp
  NAME = example_job_service
  EXTERNAL_ACCESS_INTEGRATIONS = (GOOGLE_APIS_ACCESS_INTEGRATION)
  FROM SPECIFICATION $$
  spec:
    container:
    - name: curl
      image: /tutorial_db/data_schema/tutorial_repo/alpine-curl:lat
      command:
      - "curl"
      - "http://google.com/"
  $$;
```

This example EXECUTE JOB SERVICE command specifies inline specification and the optional EXTERNAL_ACCESS_INTEGRATIONS property to include the EAI. This allows egress traffic from the job to destinations specified in the network rules the EAI allows.

Network egress using private connectivity

Instead of routing network egress via the public internet, you might opt to direct your service's egress traffic through a private connectivity endpoint.

You first need to create the private connectivity endpoint in your Snowflake account. Then configure a network rule to permit outgoing traffic to use private connectivity. The process for setting up an External Access Integration (EAI) remains the same as described in the preceding section.

Note

Private communication requires that both Snowflake and the customer's cloud account use the same cloud provider and same region.

For example, if you want to enable your service's outbound internet access to an Amazon S3 bucket via private connectivity, you do the following:

1. Enable the private link connectivity for the self-maintained endpoint service (Amazon S3). For step-by-step instructions, see AWS Private Link for Amazon S3.
2. Call the SYSTEM$PROVISION_PRIVATELINK_ENDPOINT system function to provision a private connectivity endpoint in your Snowflake VNet. This enables Snowflake to connect to the external service (in this example, Amazon S3) using private connectivity.

   ```
   USE ROLE ACCOUNTADMIN;

   SELECT SYSTEM$PROVISION_PRIVATELINK_ENDPOINT(
     'com.amazonaws.us-west-2.s3',
     '*.s3.us-west-2.amazonaws.com'
   );
   ```

3. In the cloud provider account, approve the endpoint. In this example, for Amazon AWS, see Accept or reject connection requests in the AWS documentation. Also, to approve the endpoint in Azure, see the Azure documentation.
4. Use the CREATE NETWORK RULE command to create an egress network rule specifying the external destinations that you want to allow access to.

   ```
   CREATE OR REPLACE NETWORK RULE private_link_network_rule
     MODE = EGRESS
     TYPE = PRIVATE_HOST_PORT
     VALUE_LIST = ('<bucket-name>.s3.us-west-2.amazonaws.com');
   ```

   The TYPE parameter value is set to PRIVATE_HOST_PORT. It indicates that the network rule allows outgoing network traffic to use private connectivity.
5. The rest of the steps to create an EAI and use it to create a service are the same as explained in the preceding section (see Configuring network egress).

For more information about working with private connectivity endpoints, see the following:

- Manage private connectivity endpoints: AWS
- Manage private connectivity endpoints: Azure

Configuring network communications between containers

There are two considerations:

- **Communications between containers of a service instance:** If a service instance runs multiple containers, these containers can communicate with each other over localhost (there is no need to define endpoints in the service specification).
- **Communication between containers across multiple services or multiple service instances:** Containers belonging to different services (or different instances of the same service) can communicate using endpoints defined in specification files. For more information, see Service-to-service communications.

Passing credentials to a container using Snowflake secrets

There are many reasons why you might want to pass Snowflake managed credentials into your container. For example, your service might communicate with external endpoints (outside Snowflake), in which case you will need to provide credential information in your container for your application code to use.

To provide credentials, first store them in Snowflake secret objects. Then, in the service specification, use `containers.secrets` to specify which secret objects to use and where to place them inside the container. You can either pass these credentials to environment variables in the containers, or make them available in local files in the containers.

Specifying Snowflake secrets

Specify a Snowflake secret by name or reference (reference is applicable only in the Native Application scenario):

- **Pass Snowflake secret by name:** You can pass a secret name as the `snowflakeSecret` field value.

```
...
secrets:
- snowflakeSecret:
    objectName: '<secret-name>'
```

```
      <other info about where in the container to copy the secret>
      ...
```

Note that you can optionally specify `<secret-name>` directly as the `snowflakeSecret` value.

- **Pass Snowflake secret by reference:** When using Snowpark Container Services to create a Native App (an app with containers), the app producer and consumers use different Snowflake accounts. In some contexts an installed Snowflake Native App needs to access existing secret objects in the consumer account that exist outside the APPLICATION object. In this case, developers can use the "secrets by reference" specification syntax to handle credentials as shown:

  ```
  containers:
  - name: main
    image: <url>
    secrets:
    - snowflakeSecret:
        objectReference: '<reference-name>'
      <other info about where in the container to copy the secret>
  ```

  Note that the specification uses `objectReference` instead of the `objectName` to provide a secret reference name.

Specifying secrets placement inside the container

You can tell Snowflake to either place the secrets in the containers as environment variables or write them into local container files.

Pass secrets as environment variables

To pass Snowflake secrets to containers as environment variables, include `envVarName` in the `containers.secrets` field.

```
containers:
- name: main
  image: <url>
  secrets:
  - snowflakeSecret: <secret-name>
```

```
      secretKeyRef: username | password | secret_string |  'access_token'
      envVarName: '<env-variable-name>'
```

The `secretKeyRef` value depends on the type of Snowflake secret. Possible values are the following:

- `username` or `password` if the Snowflake secret is of the `password` type.
- `secret_string` if the Snowflake secret is of the `generic_string` type.

Note that Snowflake does not update secrets passed as environment variables after a service is created.

Example 1: Passing secrets of the *password* type as environment variables

In this example, you create the following Snowflake secret object of the `password` type:

```
CREATE SECRET testdb.testschema.my_secret_object
  TYPE = password
  USERNAME = 'snowman'
  PASSWORD = '1234abc';
```

To provide this Snowflake secret object to the environment variables (for example, `LOGIN_USER` and `LOGIN_PASSWORD`) in your container, add the following `containers.secrets` field in the specification file:

```
containers:
- name: main
  image: <url>
  secrets:
  - snowflakeSecret: testdb.testschema.my_secret_object
    secretKeyRef: username
    envVarName: LOGIN_USER
  - snowflakeSecret: testdb.testschema.my_secret_object
    secretKeyRef: password
    envVarName: LOGIN_PASSWORD
```

In this example, the `snowflakeSecret` value is a fully qualified object name because secrets can be stored in a different schema than the service that is being created.

The `containers.secrets` field in this example is a list of two `snowflakeSecret` objects:

- The first object maps `username` in the Snowflake secret object to the `LOGIN_USER` environment variable in your container.
- The second object maps the `password` in the Snowflake secret object to the `LOGIN_PASSWORD` environment variable in your container.

Example 2: Passing secrets of the *generic_string* type as environment variables

In this example, you create the following Snowflake secret object of the `generic_string` type:

```
CREATE SECRET testdb.testschema.my_secret
  TYPE=generic_string
  SECRET_STRING='
       some_magic: config
  ';
```

To provide this Snowflake secret object to environment variables (for example, GENERIC_SECRET) in your container, you add the following `containers.secrets` field in the specification file:

```
containers:
- name: main
  image: <url>
  secrets:
  - snowflakeSecret: testdb.testschema.my_secret
    secretKeyRef: secret_string
    envVarName: GENERIC_SECRET
```

Write secrets in local container files

To make Snowflake secrets available to your application container in local container files, include a `containers.secrets` field: To make Snowflake secrets available to your application container in local container files, include `directoryPath` in the `containers.secrets`:

```
containers:
- name: <name>
  image: <url>
  ...
```

```
    secrets:
    - snowflakeSecret: <snowflake-secret-name>
      directoryPath: '<local directory path in the container>'
```

Snowflake populates necessary files for the secret in this specified `directoryPath`; specifying the `secretKeyRef` is not necessary. Depending on the secret type, Snowflake creates the following files in the container under the directory path you provided:

- `username` and `password` if the Snowflake secret is of the `password` type.
- `secret_string` if the Snowflake secret is of the `generic_string` type.
- `access_token` if the Snowflake secret is of the `oauth2` type.

Note

After a service is created, if the Snowflake secret object is updated, Snowflake will update the corresponding secret files in the running containers.

Example 1: Passing secrets of the *password* type in local container files

In this example, you create the following Snowflake secret object of the `password` type:

```
CREATE SECRET testdb.testschema.my_secret_object
  TYPE = password
  USERNAME = 'snowman'
  PASSWORD = '1234abc';
```

To make these credentials available in local container files, add the following `containers.secrets` field in the specification file:

```
containers:
- name: main
  image: <url>
  secrets:
  - snowflakeSecret: testdb.testschema.my_secret_object
    directoryPath: '/usr/local/creds'
```

When you start your service, Snowflake creates two files inside the container: `/usr/local/creds/username` and `/usr/local/creds/password`. Your application code can then read these files.

Example 2: Passing secrets of the *generic_string* type in local container files

In this example, you create the following Snowflake secret object of the `generic_string` type:

```
CREATE SECRET testdb.testschema.my_secret
  TYPE=generic_string
  SECRET_STRING='
       some_magic: config
  ';
```

To provide this Snowflake secret object in local container files, you add the following `containers.secrets` field in the specification file:

```
containers:
- name: main
  image: <url>
  secrets:
  - snowflakeSecret: testdb.testschema.my_secret
    directoryPath: '/usr/local/creds'
```

When you start your service, Snowflake creates this file inside the containers: `/usr/local/creds/secret_string`.

Example 3: Passing secrets of the *oauth2* type in local container files

In this example, you create the following Snowflake secret object of the `oauth2` type:

```
CREATE SECRET testdb.testschema.oauth_secret
  TYPE = OAUTH2
  OAUTH_REFRESH_TOKEN = '34n;vods4nQsdg09wee4qnfvadH'
  OAUTH_REFRESH_TOKEN_EXPIRY_TIME = '2023-12-31 20:00:00'
  API_AUTHENTICATION = my_integration;
```

To make these credentials available in local container files, add the following `containers.secrets` field in the specification file:

```
containers:
- name: main
```

```
    image: <url>
    secrets:
    - snowflakeSecret: testdb.testschema.oauth_secret
      directoryPath: '/usr/local/creds'
```

Snowflake fetches the access token from the OAuth secret object and creates `/usr/local/creds/access_token` in the containers.

When a service uses secrets of the oauth2 type, the service is expected to use that secret to access an internet destination. An oauth secret must be allowed by External Access Integration (EAI); otherwise CREATE SERVICE or EXECUTE JOB SERVICE will fail. This extra EAI requirement only applies to secrets of the oauth2 type and not to other types of secrets.

In summary, the typical steps in creating such a service are:

1. Create a secret of the oauth2 type (shown earlier).
2. Create an EAI to allow use of the secret by a service. For example:

```
CREATE OR REPLACE EXTERNAL ACCESS INTEGRATION example_eai
  ALLOWED_NETWORK_RULES = (<name>)
  ALLOWED_AUTHENTICATION_SECRETS = (testdb.testschema.oauth_secret)
  ENABLED = true;
```

3. Create a service that includes a `containers.secrets` field in the specification. That also specifies the optional EXTERNAL_ACCESS_INTEGRATIONS property to include an EAI to allow use of the oauth2 secret.

   An example CREATE SERVICE (with inline specification) command:

```
CREATE SERVICE eai_service
  IN COMPUTE POOL MYPOOL
  EXTERNAL_ACCESS_INTEGRATIONS = (example_eai)
  FROM SPECIFICATION
  $$
  spec:
    containers:
      - name: main
        image: <url>
        secrets:
        - snowflakeSecret: testdb.testschema.oauth_secret
          directoryPath: '/usr/local/creds'
    endpoints:
      - name: api
```

```
          port: 8080
      $$;
```

For more information about egress, see Configuring network egress.

Guidelines and limitations

- **General limitations:** If you encounter any issues with these limitations, contact your account representative.
  - You can create up to 200 services in your Snowflake account.
  - Each service can expose up to 100 endpoints.
  - The following limitations apply when you enable internet access (see Configuring network egress) using external access integrations (EAIs).
    - Each service can support up to 10 EAIs (see CREATE SERVICE and ALTER SERVICE).
    - Each EAI can have up to 100 host names.
  - Snowpark Container Services supports outbound private connectivity on both AWS and Azure. Contact your account representative if you require inbound private connectivity.
- **Image plaform requirements:** Currently, Snowpark Container Services requires linux/amd64 platform images.
- **Service containers are not privileged:** Service containers do not run as privileged, and therefore cannot change the configuration of the hardware on the host and can change only limited OS configurations. Service containers can only perform operating system configurations that a normal user (that is, a user who doesn't require root) can do.
- **Renaming the database and schema:**
  - Do not rename databases and schemas where you already created a service. Renaming is effectively moving a service to another database and schema, which is not supported. For example:
    - The service DNS name will continue to reflect the old database and schema name.
    - Database and schema information that Snowflake provided to the running service containers will continue to refer to the old names.

    - New logs that services ingest in the event table will continue to refer to the old database and schema names.
    - The service function will continue to reference the service in the old database and schema, and when you invoke the service function, it will fail.
  - A service specification can reference objects such as Snowflake stages and image repositories. If you rename database or schema names where these objects reside, you need to manually update the database and schema names of the referenced objects in the service specification.
- **Dropping and un-dropping a database and schema:**
  - When you drop the parent database or schema, services are deleted asynchronously. This means that a service might continue to run for some time before internal processes remove it.
  - If you attempt to un-drop a previously deleted database or schema, there is no guarantee that services will be restored.
- **Ownership transfer:** Ownership transfer for services (including job services) is not supported.
- **Replication:** When dealing with replication in Snowflake, note the following:
  - Snowpark Container Services objects, such as services, compute pools, and repositories, cannot be replicated.
  - If you create a repository within a database, the entire database cannot be replicated. In cases where the database contains other resources, such as services or compute pools, the database replication process will succeed, but these individual objects within the database will not be replicated.
- **Job services timeout:** Snowpark Container Services job services runs synchronously. If a statement times out, the job service is canceled. The default statement timeout is two days. Customers can change the timeout by setting the parameter STATEMENT_TIMEOUT_IN_SECONDS using ALTER SESSION.

  ```
  ALTER SESSION SET statement_timeout_in_seconds=<time>
  ```

  Set it before running the EXECUTE JOB SERVICE command.