Source: admin.md

Admin & System

Administrative features for monitoring, auditing, and system management.

System Dashboard

Get system-wide statistics and catalog summaries.

```
from pypangolin import PangolinClient

client = PangolinClient(uri="http://localhost:8080")
client.login("admin", "password")

# Get system stats (Root only)
stats = client.system.get_stats()
print(f"Catalogs: {stats.catalogs_count}, Tenants: {stats.tenants_count}")

# Get catalog summary
summary = client.system.get_catalog_summary("my_catalog")
print(f"Tables: {summary.table_count}, Namespaces: {summary.namespace_count}")
```

Audit Logging

Track all operations for compliance and debugging.

```
# List recent audit events
events = client.audit.list_events(limit=10)
for event in events:
    print(f"{event.action} on {event.resource_type} at {event.timestamp}")

# Count events
total = client.audit.count()
print(f"Total audit events: {total}")

# Get specific event
event = client.audit.get(event_id)
print(f"Result: {event.result}, User: {event.user_id}")
```

Asset Search

Search across all assets with tag filtering.

```
# Search by query
results = client.search.query("sales")

# Search with tags
```

```
results = client.search.query("customer", tags=["pii", "verified"])

for asset in results:
    print(f"{asset.name} ({asset.kind}) - {asset.catalog}.{asset.namespace}")
    print(f"  Has Access: {asset.has_access}, Discoverable: {asset.discoverable}")
```

Token Management

Generate and manage API tokens for automation.

```
# Generate token
token_response = client.tokens.generate("ci-pipeline", expires_in_days=90)
token = token_response["token"]

# List my tokens
tokens = client.tokens.list_my_tokens()
for t in tokens:
    print(f"{t['name']}: expires {t.get('expires_at', 'never')}")

# Revoke token
client.tokens.revoke(token_id)
```

System Configuration

Manage system-wide settings (Root only).

```
# Get current settings
settings = client.system.get_settings()
print(f"Public signup: {settings['allow_public_signup']}")

# Update settings
new_settings = {
    "allow_public_signup": False,
    "default_retention_days": 90
}
client.system.update_settings(new_settings)
```

Source: auth_mode.md

Authenticated Mode Guide

This guide covers how to use `pypangolin` when the Pangolin API is running in Authenticated Mode (default).

Prerequisites

-

Pangolin API running with `PANGOLIN_AUTH_MODE=auth` (default).

A valid user account (or Root credentials).

1. Connecting as Root

The system has a built-in Root superuser defined by environment variables (`PANGOLIN_ROOT_USER`, `PANGOLIN_ROOT_PASSWORD`) or database seeding.

```
from pypangolin import PangolinClient

# Connect and Login as Root
client = PangolinClient(
    uri="http://localhost:8080",
    username="admin",
    password="password"
)

print(f"Logged in. Token: {client.token}")
```

2. Onboarding a Tenant

The Root user's primary job is to create Tenants and onboard Tenant Admins.

```
# 1. Create a Tenant
tenant = client.tenants.create(name="acme_corp")
print(f"Created Tenant: {tenant.id}")

# 2. Create a Tenant Admin
# Note: Root users cannot login to tenants directly with their root credentials in some configurations.
# Best practice is to create a specific admin for the tenant.
admin_user = client.users.create(
    username="acme_admin",
    email="admin@acme.com",
    role="tenant-admin",
    tenant_id=tenant.id,
    password="secure_password"
)
```

3. Connecting as a Tenant User

Regular users and Tenant Admins must be scoped to their tenant.

```
# Connect as Tenant Admin
client = PangolinClient(
    uri="http://localhost:8080",
    username="acme_admin",
    password="secure_password",
    tenant_id=tenant.id # Optional if username is unique, but recommended
```

```
)

# The client automatically handles the Tenant Context for subsequent requests
warehouses = client.warehouses.list()
```

4. Managing Resources

Once authenticated as a Tenant Admin, you can manage resources for your organization.

Warehouses

Warehouses define where your data is stored (S3, GCS, Azure, Local).

```
warehouse = client.warehouses.create_s3(
    name="analytics_wh",
    bucket="my-company-data",
    region="us-east-1",
    access_key="AWS_ACCESS_KEY", # Optional: for credential vending
    secret_key="AWS_SECRET_KEY",
    vending_strategy="AwsStatic"
)
```

Catalogs

Catalogs organize your data tables (Iceberg, Delta, etc.).

```
catalog = client.catalogs.create(
    name="gold_data",
    warehouse="analytics_wh",
    type="Local" # Native Pangolin Catalog
)
```

Namespaces & Tables

Manage the hierarchy within your catalog.

```
# Working with Namespaces
ns_client = client.catalogs.namespaces("gold_data")
ns_client.create(["sales", "2024"])

# Listing Namespaces
namespaces = ns_client.list()
```

Source: README.md

Database Connections Documentation

Secure credential management for database connections with encrypted storage.

Overview

PyPangolin provides secure credential management for database connections using Fernet encryption. Credentials are encrypted before storage and decrypted only when creating connections.

Main Guide

Database Connections Overview - Complete guide to database connection management

SQL Databases

PostgreSQL *Tested* - Open-source relational database

MySQL *Tested* - Popular relational database

Amazon Redshift ⚠ *Untested* - Cloud data warehouse (Postgres-compatible)

NoSQL Databases

MongoDB *Tested* - Document database

Cloud Data Warehouses

Snowflake ⚠ *Untested* - Cloud data platform

Azure Synapse ⚠ *Untested* - Microsoft analytics service

Google BigQuery ⚠ *Untested* - Serverless data warehouse

Analytics Platforms

Dremio *Tested* - Data lakehouse platform with Arrow Flight

Quick Start

```
# Install with database support
pip install "pypangolin[postgres]"
pip install "pypangolin[mysql]"
pip install "pypangolin[mongodb]"
pip install "pypangolin[dremio]"

# Or install all database support
pip install "pypangolin[all-connections]"
```

Security

All database connections use Fernet encryption for credential storage. See individual guides for security best practices and key management strategies.

Source: bigquery.md

Google BigQuery Connections

[!WARNING]
Untested Implementation

This connection type has been implemented based on Google Cloud BigQuery's Python client documentation but has not been tested against a live BigQuery project. The implementation should work as designed, but users should verify functionality in their environment.

Securely store and connect to Google BigQuery with encrypted credentials.

Installation

```
pip install "pypangolin[bigquery]"
```

Registering a Connection

Using Service Account JSON

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import BigQueryAsset
from cryptography.fernet import Fernet
import json

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")

# User-managed encryption (recommended)
encryption_key = Fernet.generate_key().decode('utf-8')

# Load service account JSON
with open('service-account.json', 'r') as f:
    credentials_json = f.read()

BigQueryAsset.register(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="prod_bigquery",
    connection_string="bigquery://my-gcp-project",
    credentials={
```

```
        "project_id": "my-gcp-project",
        "credentials_json": credentials_json  # Service account JSON as string
    },
    encryption_key=encryption_key,
    store_key=False,
    description="Production BigQuery project"
)
```

Using Application Default Credentials

```
BigQueryAsset.register(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="dev_bigquery",
    connection_string="bigquery://my-dev-project",
    credentials={
        "project_id": "my-dev-project"
        # No credentials_json - will use Application Default Credentials
    },
    encryption_key=encryption_key,
    store_key=False,
    description="Development BigQuery (uses ADC)"
)
```

Connecting to BigQuery

```
import os

encryption_key = os.getenv("BIGQUERY_ENCRYPTION_KEY")

bq_client = BigQueryAsset.connect(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="prod_bigquery",
    encryption_key=encryption_key
)

# Use the BigQuery client
query = """
    SELECT
        DATE(timestamp) as date,
        COUNT(*) as event_count
    FROM `project.dataset.events`
    WHERE DATE(timestamp) >= DATE_SUB(CURRENT_DATE(), INTERVAL 7 DAY)
    GROUP BY 1
    ORDER BY 1 DESC
"""
```

```
query_job = bq_client.query(query)
results = query_job.result()

for row in results:
    print(f"Date: {row.date}, Events: {row.event_count}")
```

Required Credentials

- **project_id** (required) - GCP project ID

Optional Credentials

- **credentials_json** - Service account JSON as string (if not using ADC)

Connection String Format

```
bigquery://project-id
```

Authentication Methods

1. Service Account JSON (Recommended for Production)

- Most secure for production
- Credentials encrypted and stored in Pangolin
- No dependency on environment configuration

2. Application Default Credentials (Development)

- Uses `gcloud auth application-default login`
- Good for development/testing
- Requires proper GCP SDK configuration

Example Usage

Query Data

```
bq_client = BigQueryAsset.connect(client, "data_sources", "warehouses", "prod_bigquery",
encryption_key=key)

# Standard SQL query
query = """
    SELECT
```

```
        user_id,
        COUNT(*) as action_count,
        MAX(timestamp) as last_action
    FROM `my-project.analytics.user_actions`
    WHERE DATE(timestamp) = CURRENT_DATE()
    GROUP BY user_id
    HAVING action_count > 10
"""

df = bq_client.query(query).to_dataframe()
print(df.head())
```

Create Table

```
from google.cloud import bigquery

bq_client = BigQueryAsset.connect(client, "data_sources", "warehouses", "prod_bigquery",
encryption_key=key)

schema = [
    bigquery.SchemaField("name", "STRING", mode="REQUIRED"),
    bigquery.SchemaField("age", "INTEGER", mode="REQUIRED"),
    bigquery.SchemaField("email", "STRING", mode="NULLABLE"),
]

table_ref = bq_client.dataset("my_dataset").table("my_table")
table = bigquery.Table(table_ref, schema=schema)
table = bq_client.create_table(table)
print(f"Created table {table.project}.{table.dataset_id}.{table.table_id}")
```

Security Best Practices

- Use service account JSON with minimal required permissions
- Rotate service account keys regularly
- Use user-managed encryption keys
- Enable BigQuery audit logging
- Use VPC Service Controls for additional security
- Never commit service account JSON to version control

Troubleshooting

Authentication Errors

- Verify service account JSON is valid

- Check service account has necessary IAM roles
- Ensure project_id matches the service account project

Permission Denied

- Verify service account has BigQuery Data Viewer/Editor roles
- Check dataset-level permissions
- Ensure billing is enabled on the project

Quota Exceeded

- Check BigQuery quotas in GCP Console
- Consider using batch queries for large operations
- Implement rate limiting in your application

Notes

- BigQuery is serverless - no connection pooling needed
- Queries are billed based on data processed
- Use partitioned tables and clustering for cost optimization
- Standard SQL is recommended over Legacy SQL
- Test thoroughly in your environment before production deployment

Source: dremio.md

Dremio Connections

Connect to Dremio using Arrow Flight with dremioframe.

Installation

```
pip install "pypangolin[dremio]"
```

Registering a Connection

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import DremioAsset

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")
```

```
# For Dremio Cloud
DremioAsset.register(
    client,
    catalog="data_sources",
    namespace="analytics",
    name="dremio_cloud",
    connection_string="grpc+tls://data.dremio.cloud:443",
    credentials={
        "token": "your-personal-access-token",
        "project_id": "your-project-id",  # Optional: specify non-default project
        "tls": "true",
        "disable_certificate_verification": "false"
    },
    store_key=False,  # Recommended for tokens
    description="Dremio Cloud connection"
)

# For Dremio Software
DremioAsset.register(
    client,
    catalog="data_sources",
    namespace="analytics",
    name="dremio_onprem",
    connection_string="grpc://dremio.company.com:32010",
    credentials={
        "username": "dremio_user",
        "password": "password123"
    },
    store_key=True,
    description="On-premise Dremio"
)
```

Connecting to Dremio

```
import os

# Connect to Dremio Cloud
encryption_key = os.getenv("DREMIO_ENCRYPTION_KEY")

dremio_conn = DremioAsset.connect(
    client,
    catalog="data_sources",
    namespace="analytics",
    name="dremio_cloud",
    encryption_key=encryption_key
)

# Query using dremioframe
df = dremio_conn.query("SELECT * FROM Samples.\"samples.dremio.com\".\"NYC-taxi-trips\"
LIMIT 10")
print(df.head())
```

Connection String Formats

```
grpc://host:port              # Dremio Software (unencrypted)
grpc+tls://host:port          # Dremio with TLS
grpc+tls://data.dremio.cloud:443  # Dremio Cloud
```

Credential Options

- **Token-based** (Dremio Cloud): Use `token` credential
- **Username/Password** (Dremio Software): Use `username` and `password`
- **Project ID** (Dremio Cloud): Use `project_id` to specify a non-default project
- **TLS**: Set `tls` to `"true"` for encrypted connections

Notes

- Dremio Cloud requires TLS and personal access tokens
- For on-premise Dremio, verify the Arrow Flight port (default: 32010)
- `dremioframe` provides pandas-like DataFrame operations

Source: mongodb.md

MongoDB Connections

Securely store and connect to MongoDB databases with encrypted credentials.

Installation

```
pip install "pypangolin[mongodb]"
```

Registering a Connection

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import MongoDBAsset

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")

MongoDBAsset.register(
    client,
    catalog="data_sources",
    namespace="databases",
    name="prod_mongo",
    connection_string="mongodb://mongo.example.com:27017/admin",
```

```
    credentials={
        "username": "dbuser",
        "password": "securepassword123"
    },
    store_key=True,  # Or False for user-managed
    description="Production MongoDB database"
)
```

Connecting to Database

```
mongo_client = MongoDBAsset.connect(
    client,
    catalog="data_sources",
    namespace="databases",
    name="prod_mongo"
)

# Use the MongoDB client
db = mongo_client["mydb"]
collection = db["mycollection"]
documents = collection.find().limit(10)

for doc in documents:
    print(doc)

mongo_client.close()
```

Connection String Formats

```
mongodb://host:port/database
mongodb://host:port  # No specific database
mongodb+srv://cluster.mongodb.net/database  # MongoDB Atlas
```

Important Notes

- Use `/admin` database for authentication with root credentials
- MongoDB credentials are inserted into the connection string automatically
- For MongoDB Atlas, include the full connection string with `+srv`

Source: mysql.md

MySQL Connections

Securely store and connect to MySQL databases with encrypted credentials.

Installation

```
pip install "pypangolin[mysql]"
```

Registering a Connection

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import MySQLAsset
from cryptography.fernet import Fernet

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")

# User-managed encryption (recommended)
encryption_key = Fernet.generate_key().decode('utf-8')

MySQLAsset.register(
    client,
    catalog="data_sources",
    namespace="databases",
    name="prod_mysql",
    connection_string="mysql://db.example.com:3306/production",
    credentials={
        "username": "dbuser",
        "password": "securepassword123"
    },
    encryption_key=encryption_key,
    store_key=False,
    description="Production MySQL database"
)
```

Connecting to Database

```
import os

encryption_key = os.getenv("DB_ENCRYPTION_KEY")

conn = MySQLAsset.connect(
    client,
    catalog="data_sources",
    namespace="databases",
    name="prod_mysql",
    encryption_key=encryption_key
)

cursor = conn.cursor()
cursor.execute("SELECT DATABASE()")
print(cursor.fetchone())
cursor.close()
conn.close()
```

Connection String Formats

```
mysql://host:port/database
mysql://host/database  # Default port 3306
```

Security Best Practices

Same as PostgreSQL - use user-managed keys for production, store keys in secrets managers, and leverage Pangolin RBAC.

Source: postgresql.md

PostgreSQL Connections

Securely store and connect to PostgreSQL databases with encrypted credentials.

Installation

```
pip install "pypangolin[postgres]"
```

Registering a Connection

Auto-managed Encryption (Recommended for Development)

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import PostgreSQLAsset

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")

# Register with auto-generated encryption key (stored in properties)
PostgreSQLAsset.register(
    client,
    catalog="data_sources",
    namespace="databases",
    name="prod_postgres",
    connection_string="postgresql://db.example.com:5432/production",
    credentials={
        "username": "dbuser",
        "password": "securepassword123"
    },
    store_key=True,  # Key stored in asset properties
    description="Production PostgreSQL database"
)
```

User-managed Encryption (Recommended for Production)

```
from cryptography.fernet import Fernet

# Generate and securely store your encryption key
encryption_key = Fernet.generate_key().decode('utf-8')
# Store this key in environment variables or secrets manager!

PostgreSQLAsset.register(
    client,
    catalog="data_sources",
    namespace="databases",
    name="secure_postgres",
    connection_string="postgresql://db.example.com:5432/production",
    credentials={
        "username": "dbuser",
        "password": "securepassword123"
    },
    encryption_key=encryption_key,
    store_key=False,  # Key NOT stored - you must provide it when connecting
    description="Secure PostgreSQL with user-managed key"
)
```

Connecting to Database

Auto-managed Key

```
# Connect (key retrieved from properties)
conn = PostgreSQLAsset.connect(
    client,
    catalog="data_sources",
    namespace="databases",
    name="prod_postgres"
)

# Use the connection
cursor = conn.cursor()
cursor.execute("SELECT * FROM users LIMIT 10")
results = cursor.fetchall()
cursor.close()
conn.close()
```

User-managed Key

```
import os

# Retrieve key from environment variable
encryption_key = os.getenv("DB_ENCRYPTION_KEY")
```

```
# Connect (must provide key)
conn = PostgreSQLAsset.connect(
    client,
    catalog="data_sources",
    namespace="databases",
    name="secure_postgres",
    encryption_key=encryption_key
)

# Use the connection
cursor = conn.cursor()
cursor.execute("SELECT version()")
print(cursor.fetchone())
cursor.close()
conn.close()
```

Connection String Formats

```
postgresql://host:port/database
postgresql://host/database  # Default port 5432
```

Security Best Practices

[!IMPORTANT]
Production Recommendations

Use user-managed keys for production databases

Store encryption keys in environment variables or secrets managers (AWS Secrets Manager, HashiCorp Vault)

Rotate credentials periodically and re-register assets

Use Pangolin RBAC to restrict who can access connection assets

Monitor access via audit logs

[!WARNING]
Auto-managed Keys

When `store_key=True`, anyone with Read access to the asset can decrypt credentials. Only use for development/testing.

Advanced Usage

Connection with SSL

```
PostgreSQLAsset.register(
    client,
    catalog="data_sources",
```

```
    namespace="databases",
    name="ssl_postgres",
    connection_string="postgresql://db.example.com:5432/production",
    credentials={
        "username": "dbuser",
        "password": "securepassword123",
        "sslmode": "require"  # Additional connection parameters
    },
    store_key=False
)
```

Troubleshooting

Connection Refused

- Verify PostgreSQL is running and accessible
- Check firewall rules
- Ensure correct host and port

Authentication Failed

- Verify username and password
- Check PostgreSQL `pg_hba.conf` for authentication method
- Ensure user has necessary permissions

Encryption Key Errors

- For user-managed keys, ensure you provide the same key used during registration
- Fernet keys must be 32 bytes base64-encoded

Source: redshift.md

Amazon Redshift Connections

[!WARNING]
Untested Implementation

This connection type has been implemented based on Redshift's PostgreSQL compatibility but has not been tested against a live Redshift cluster. The implementation should work as designed, but users should verify functionality in their environment.

Securely store and connect to Amazon Redshift data warehouses with encrypted credentials.

Installation

```
pip install "pypangolin[redshift]"
```

Redshift uses the PostgreSQL driver (`psycopg2`) since Redshift is PostgreSQL-compatible.

Registering a Connection

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import RedshiftAsset
from cryptography.fernet import Fernet

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")

# User-managed encryption (recommended)
encryption_key = Fernet.generate_key().decode('utf-8')

RedshiftAsset.register(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="prod_redshift",

connection_string="redshift://cluster-name.region.redshift.amazonaws.com:5439/analytics"
,
    credentials={
        "username": "redshift_user",
        "password": "securepassword123"
    },
    encryption_key=encryption_key,
    store_key=False,
    description="Production Redshift cluster"
)
```

Connecting to Redshift

```
import os

encryption_key = os.getenv("REDSHIFT_ENCRYPTION_KEY")

conn = RedshiftAsset.connect(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="prod_redshift",
    encryption_key=encryption_key
)
```

```
# Use the connection (same as PostgreSQL)
cursor = conn.cursor()
cursor.execute("SELECT version()")
print(cursor.fetchone())
cursor.close()
conn.close()
```

Connection String Format

```
redshift://cluster-name.region.redshift.amazonaws.com:5439/database
redshift://cluster-endpoint:5439/database
```

Security Best Practices

- Use IAM authentication when possible (requires additional setup)
- Enable SSL/TLS for connections
- Use VPC security groups to restrict access
- Rotate credentials regularly
- Use user-managed encryption keys

Redshift-Specific Features

Using SSL

```
RedshiftAsset.register(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="secure_redshift",
    connection_string="redshift://cluster.region.redshift.amazonaws.com:5439/analytics",
    credentials={
        "username": "redshift_user",
        "password": "securepassword123",
        "sslmode": "require"  # Require SSL connection
    },
    encryption_key=encryption_key,
    store_key=False
)
```

Example Usage

```
# Query Redshift
conn = RedshiftAsset.connect(client, "data_sources", "warehouses", "prod_redshift",
```

```
encryption_key=key)
cursor = conn.cursor()

# Redshift-optimized query
cursor.execute("""
    SELECT
        date_trunc('day', event_time) as day,
        event_type,
        count(*) as event_count
    FROM events
    WHERE event_time >= current_date - interval '7 days'
    GROUP BY 1, 2
    ORDER BY 1 DESC, 3 DESC
""")

for row in cursor:
    print(f"Day: {row[0]}, Event: {row[1]}, Count: {row[2]}")

cursor.close()
conn.close()
```

Troubleshooting

Connection Refused

- Verify cluster is available and not paused
- Check VPC security group rules
- Ensure cluster endpoint is correct
- Verify port 5439 is accessible

Authentication Failed

- Verify username and password
- Check if user exists in the cluster
- Ensure user has CONNECT privilege on database

Notes

- Redshift is PostgreSQL 8.0.2 compatible
- Uses `psycopg2` driver (same as PostgreSQL)
- Default port is 5439 (not 5432 like PostgreSQL)
- Supports most PostgreSQL syntax with some differences

Test thoroughly in your environment before production deployment

Source: snowflake.md

Snowflake Connections

[!WARNING]
Untested Implementation

This connection type has been implemented based on Snowflake's Python connector documentation but has not been tested against a live Snowflake instance. The implementation should work as designed, but users should verify functionality in their environment.

Securely store and connect to Snowflake data warehouses with encrypted credentials.

Installation

```
pip install "pypangolin[snowflake]"
```

Registering a Connection

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import SnowflakeAsset
from cryptography.fernet import Fernet

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")

# User-managed encryption (recommended)
encryption_key = Fernet.generate_key().decode('utf-8')

SnowflakeAsset.register(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="prod_snowflake",
    connection_string="snowflake://account.region.snowflakecomputing.com",
    credentials={
        "account": "myaccount",
        "username": "snowflake_user",
        "password": "securepassword123",
        "warehouse": "COMPUTE_WH",      # Optional
        "database": "ANALYTICS",        # Optional
        "schema": "PUBLIC",             # Optional
        "role": "ANALYST"               # Optional
    },
    encryption_key=encryption_key,
    store_key=False,
    description="Production Snowflake warehouse"
```

```
)
```

Connecting to Snowflake

```
import os

encryption_key = os.getenv("SNOWFLAKE_ENCRYPTION_KEY")

conn = SnowflakeAsset.connect(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="prod_snowflake",
    encryption_key=encryption_key
)

# Use the connection
cursor = conn.cursor()
cursor.execute("SELECT CURRENT_VERSION()")
print(cursor.fetchone())
cursor.close()
conn.close()
```

Required Credentials

account (required) - Snowflake account identifier

username (required) - Snowflake username

password (required) - Snowflake password

Optional Parameters

warehouse - Virtual warehouse to use

database - Default database

schema - Default schema

role - Role to use for the session

Connection String Format

```
snowflake://account.region.snowflakecomputing.com
```

The connection string is primarily for reference; actual connection parameters are extracted from credentials.

Security Best Practices

- Use user-managed encryption keys for production
- Store encryption keys in environment variables or secrets managers
- Rotate Snowflake credentials regularly
- Use Snowflake's role-based access control in conjunction with Pangolin RBAC

Troubleshooting

Connection Issues

- Verify account identifier is correct (format: `account.region.snowflakecomputing.com`)
- Check network connectivity to Snowflake
- Ensure user has appropriate permissions

Authentication Errors

- Verify username and password are correct
- Check if MFA is required (may need alternative authentication method)
- Ensure role has necessary privileges

Example Usage

```
# Query Snowflake
conn = SnowflakeAsset.connect(client, "data_sources", "warehouses", "prod_snowflake",
encryption_key=key)
cursor = conn.cursor()

cursor.execute("""
    SELECT
        date_trunc('day', order_date) as day,
        count(*) as order_count,
        sum(amount) as total_amount
    FROM orders
    WHERE order_date >= dateadd(day, -30, current_date())
    GROUP BY 1
    ORDER BY 1 DESC
""")

for row in cursor:
    print(f"Date: {row[0]}, Orders: {row[1]}, Amount: ${row[2]:,.2f}")

cursor.close()
conn.close()
```

Notes

- This implementation uses `snowflake-connector-python`
- Connection pooling is not currently implemented
- For production use, consider implementing connection retry logic
- Test thoroughly in your environment before production deployment

Source: synapse.md

Azure Synapse Analytics Connections

[!WARNING]
Untested Implementation

This connection type has been implemented based on Azure Synapse's ODBC driver documentation but has not been tested against a live Synapse workspace. The implementation should work as designed, but users should verify functionality in their environment.

Securely store and connect to Azure Synapse Analytics with encrypted credentials.

Installation

```
pip install "pypangolin[synapse]"
```

Additional Requirement: You must also install the Microsoft ODBC Driver for SQL Server on your system.

Installing ODBC Driver

Ubuntu/Debian:

```
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
curl  https://packages.microsoft.com/config/ubuntu/$(lsb_release  -rs)/prod.list  |  sudo
tee /etc/apt/sources.list.d/mssql-release.list
sudo apt-get update
sudo ACCEPT_EULA=Y apt-get install -y msodbcsql17
```

macOS:

```
brew tap microsoft/mssql-release https://github.com/Microsoft/homebrew-mssql-release
brew update
brew install msodbcsql17
```

Windows:

Download and install from Microsoft Download Center

Registering a Connection

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import SynapseAsset
from cryptography.fernet import Fernet

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")

# User-managed encryption (recommended)
encryption_key = Fernet.generate_key().decode('utf-8')

SynapseAsset.register(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="prod_synapse",
    connection_string="synapse://myworkspace.sql.azuresynapse.net",
    credentials={
        "server": "tcp:myworkspace.sql.azuresynapse.net,1433",
        "database": "mySynapseDW",
        "username": "synapse_user",
        "password": "securepassword123",
        "driver": "ODBC Driver 17 for SQL Server"  # Optional, defaults to this
    },
    encryption_key=encryption_key,
    store_key=False,
    description="Production Synapse Analytics"
)
```

Connecting to Synapse

```
import os

encryption_key = os.getenv("SYNAPSE_ENCRYPTION_KEY")

conn = SynapseAsset.connect(
    client,
    catalog="data_sources",
    namespace="warehouses",
    name="prod_synapse",
    encryption_key=encryption_key
)

# Use the connection
cursor = conn.cursor()
cursor.execute("SELECT @@VERSION")
print(cursor.fetchone())
cursor.close()
conn.close()
```

Required Credentials

server (required) - Synapse server endpoint (format: `tcp:server.sql.azuresynapse.net,1433`)

database (required) - Database name

username (required) - SQL authentication username

password (required) - SQL authentication password

Optional Credentials

driver - ODBC driver name (default: "ODBC Driver 17 for SQL Server")

Connection String Format

```
synapse://workspace.sql.azuresynapse.net
```

Authentication Methods

SQL Authentication (Implemented)

```
credentials={
    "server": "tcp:myworkspace.sql.azuresynapse.net,1433",
    "database": "mydb",
    "username": "sql_user",
    "password": "password123"
}
```

Azure AD Authentication (Future Enhancement)

Currently not implemented. For Azure AD auth, you would need to extend the implementation to support authentication tokens.

Example Usage

Query Data

```
conn = SynapseAsset.connect(client, "data_sources", "warehouses", "prod_synapse",
encryption_key=key)
cursor = conn.cursor()

# Query Synapse
cursor.execute("""
    SELECT
```

```
        CAST(order_date AS DATE) as date,
        COUNT(*) as order_count,
        SUM(total_amount) as revenue
    FROM dbo.orders
    WHERE order_date >= DATEADD(day, -30, GETDATE())
    GROUP BY CAST(order_date AS DATE)
    ORDER BY date DESC
""")

for row in cursor:
    print(f"Date: {row[0]}, Orders: {row[1]}, Revenue: ${row[2]:,.2f}")

cursor.close()
conn.close()
```

Using with Pandas

```
import pandas as pd

conn = SynapseAsset.connect(client, "data_sources", "warehouses", "prod_synapse",
encryption_key=key)

query = """
    SELECT TOP 1000
        customer_id,
        product_name,
        quantity,
        price
    FROM dbo.sales
    WHERE sale_date >= DATEADD(month, -1, GETDATE())
"""

df = pd.read_sql(query, conn)
print(df.head())

conn.close()
```

Security Best Practices

- Use Azure AD authentication when possible (requires custom implementation)
- Enable firewall rules to restrict access
- Use Azure Private Link for secure connectivity
- Rotate SQL credentials regularly
- Use user-managed encryption keys
- Enable Azure Synapse audit logging

Synapse-Specific Considerations

Performance Optimization

- Use CTAS (CREATE TABLE AS SELECT) for large data loads
- Leverage distribution and partitioning strategies
- Use PolyBase for external data access
- Consider result set caching

Resource Management

- Be aware of DWU (Data Warehouse Units) allocation
- Pause compute when not in use to save costs
- Use workload management for query prioritization

Troubleshooting

ODBC Driver Not Found

```
Error: [IM002] [Microsoft][ODBC Driver Manager] Data source name not found
```

Solution: Install Microsoft ODBC Driver 17 for SQL Server

Connection Timeout

- Verify firewall rules allow your IP address
- Check if Synapse workspace is paused
- Ensure correct server endpoint format

Authentication Failed

- Verify username and password are correct
- Check if user has access to the specified database
- Ensure SQL authentication is enabled (not just Azure AD)

Notes

- Synapse uses T-SQL (Transact-SQL) syntax
- Supports both dedicated SQL pools and serverless SQL pools

- This implementation targets dedicated SQL pools
- Connection string format differs from standard SQL Server
- Test thoroughly in your environment before production deployment

Source: connections.md

Database Connections

Securely store and manage database connection credentials with encrypted storage.

Overview

PyPangolin provides secure credential management for database connections using Fernet encryption. Credentials are encrypted before storage and decrypted only when creating connections.

Supported Databases

SQL Databases

- PostgreSQL - Open-source relational database *Tested*
- MySQL - Popular relational database *Tested*
- Amazon Redshift - Cloud data warehouse (Postgres-compatible) ⚠ *Untested*

NoSQL Databases

- MongoDB - Document database *Tested*

Cloud Data Warehouses

- Snowflake - Cloud data platform ⚠ *Untested*
- Azure Synapse - Microsoft analytics service ⚠ *Untested*
- Google BigQuery - Serverless data warehouse ⚠ *Untested*

Analytics Platforms

- Dremio - Data lakehouse platform with Arrow Flight *Tested*

Quick Start

1. Install Dependencies

```
# Install specific database support
pip install "pypangolin[postgres]"
pip install "pypangolin[mysql]"
pip install "pypangolin[mongodb]"
pip install "pypangolin[dremio]"

# Or install all connection support
pip install "pypangolin[all-connections]"
```

2. Register a Connection

```
from pypangolin import PangolinClient
from pypangolin.assets.connections import PostgreSQLAsset

client = PangolinClient(uri="http://localhost:8080")
client.login("username", "password")

PostgreSQLAsset.register(
    client,
    catalog="data_sources",
    namespace="databases",
    name="my_database",
    connection_string="postgresql://localhost:5432/mydb",
    credentials={
        "username": "dbuser",
        "password": "secret123"
    },
    store_key=True  # Auto-managed encryption
)
```

3. Connect to Database

```
conn = PostgreSQLAsset.connect(
    client,
    catalog="data_sources",
    namespace="databases",
    name="my_database"
)

# Use the connection
cursor = conn.cursor()
cursor.execute("SELECT * FROM users")
results = cursor.fetchall()
cursor.close()
conn.close()
```

Encryption Modes

Auto-managed Keys (Development)

```
Asset.register(..., store_key=True)
```

- Encryption key generated automatically
- Key stored in asset properties
- Convenient for development
- ⚠ Anyone with asset access can decrypt credentials

User-managed Keys (Production)

```
from cryptography.fernet import Fernet

key = Fernet.generate_key().decode('utf-8')
Asset.register(..., encryption_key=key, store_key=False)
```

- You control the encryption key
- Key NOT stored in Pangolin
- More secure for production
- Must provide key when connecting

Security Best Practices

[!IMPORTANT]
Production Security

Use user-managed keys for production databases

Store keys securely in environment variables or secrets managers

Rotate credentials regularly

Use Pangolin RBAC to control asset access

Monitor access via audit logs

Never commit keys to version control

Key Management

Environment Variables

```
import os

encryption_key = os.getenv("DB_ENCRYPTION_KEY")
```

```
conn = Asset.connect(..., encryption_key=encryption_key)
```

AWS Secrets Manager

```
import boto3
import json

secrets = boto3.client('secretsmanager')
secret = secrets.get_secret_value(SecretId='db-encryption-key')
encryption_key = json.loads(secret['SecretString'])['key']
```

HashiCorp Vault

```
import hvac

client = hvac.Client(url='https://vault.example.com')
secret = client.secrets.kv.v2.read_secret_version(path='db-keys')
encryption_key = secret['data']['data']['encryption_key']
```

Connection Lifecycle

Register: Store encrypted credentials in Pangolin

Connect: Retrieve and decrypt credentials, create connection

Use: Execute queries/operations

Close: Clean up connection resources

Rotate: Update credentials periodically

Troubleshooting

"No encryption key found"

For auto-managed keys: Ensure `store_key=True` was used during registration

For user-managed keys: Provide `encryption_key` parameter to `connect()`

"Fernet key must be 32 url-safe base64-encoded bytes"

Use `Fernet.generate_key()` to create proper keys

Don't use arbitrary strings as encryption keys

Connection Errors

- Verify database is accessible from your network
- Check connection string format
- Ensure credentials are correct
- Review database-specific documentation

See Also

- PostgreSQL Guide
- MySQL Guide
- MongoDB Guide
- Dremio Guide

Source: csv.md

CSV Assets

Manage CSV files as first-class citizens in your Pangolin Catalog.

Usage

Use `CsvAsset` to write and register CSV files.

```
import pandas as pd
from pypangolin.assets import CsvAsset

df = pd.DataFrame({"col1": ["val1"], "col2": ["val2"]})

CsvAsset.write(
    client=client,
    catalog="my_catalog",
    namespace="my_ns",
    name="my_csv_data",
    data=df,
    location="s3://bucket/data.csv",
    index=False, # Passed to to_csv
    storage_options={"key": "...", "secret": "..."},
    properties={"delimiter": ",", "encoding": "utf-8"}
)
```

Recommended Properties

Property	Description	Example
`file_size_bytes`	Size of the file	`2048`

`row_count`	Number of rows	`100`
`delimiter`	Field separator character	`,`, `\t`, `|`
`encoding`	Character encoding	`utf-8`, `latin1`
`has_header`	Whether the first row is a header	`true`, `false`

Source: delta.md

Delta Lake Assets

`pypangolin` allows you to create, write to, and register Delta Lake tables within the Pangolin Catalog.

Dependencies

To use Delta Asset support, install `pypangolin` with the `delta` extra or install `deltalake` manually:

```
pip install "pypangolin[delta]"
# or
pip install deltalake
```

Usage

You can use the `DeltaAsset` class to write a Pandas DataFrame as a Delta table and automatically register it.

```
import pandas as pd
from pypangolin import PangolinClient
from pypangolin.assets import DeltaAsset

# Initialize Client
client = PangolinClient("http://localhost:8080", "user", "password", tenant_id="...")

# Data
df = pd.DataFrame({"id": [1, 2, 3], "data": ["a", "b", "c"]})

# Write and Register
# This writes the dataframe to the specified S3 location as a Delta Table
# and registers it in the 'generic_demo' catalog under 'generic_ns' namespace.
DeltaAsset.write(
    client=client,
    catalog="generic_demo",
    namespace="generic_ns",
    name="my_delta_table",
    data=df,
    location="s3://my-bucket/path/to/delta_table",
    mode="overwrite", # or "append"
    storage_options={
        "AWS_ACCESS_KEY_ID": "...",
        "AWS_SECRET_ACCESS_KEY": "...",
```

```
        "AWS_REGION": "us-east-1"
    }
)
```

Manual Registration

If you have an existing Delta table, you can register it without writing data:

```
DeltaAsset.register(
    client=client,
    catalog="generic_demo",
    namespace="generic_ns",
    name="existing_delta_table",
    location="s3://existing-bucket/delta-table",
    properties={
        "delta.minReaderVersion": "1",
        "delta.minWriterVersion": "2",
        "description": "Sales data for Q3"
    }
)
```

Recommended Properties

When registering Delta tables, consider tracking these properties for better discoverability:

Property	Description	Example
`delta.minReaderVersion`	Minimum protocol version required to read	`1`
`delta.minWriterVersion`	Minimum protocol version required to write	`2`
`record_count`	Approximate number of rows	`1000000`
`size_in_bytes`	Total size of the table	`500MB`
`last_modified`	Timestamp of the last commit	`2025-12-24T12:00:00Z`
`partition_columns`	Comma-separated list of partition keys	`date,region`

Source: federated.md

Federated Catalogs & Views

Connect to remote Iceberg catalogs and create virtual views across your data.

Federated Catalogs

Federated catalogs allow you to connect to remote Iceberg REST catalogs and query them as if they were local.

Creating a Federated Catalog

```
from pypangolin import PangolinClient

client = PangolinClient(uri="http://localhost:8080")
client.login("admin", "password")

# Create federated catalog
fed_cat = client.federated_catalogs.create(
    name="prod_analytics",
    uri="https://prod-catalog.company.com",
    warehouse="prod_warehouse",
    credential="bearer_token_here",
    properties={
        "region": "us-east-1",
        "team": "analytics"
    }
)
```

Managing Federated Catalogs

```
# List all federated catalogs
catalogs = client.federated_catalogs.list()
for cat in catalogs:
    print(f"{cat.name}: {cat.properties.get('uri')}")

# Get specific catalog
catalog = client.federated_catalogs.get("prod_analytics")

# Test connection
result = client.federated_catalogs.test_connection("prod_analytics")
if result['status'] == 'connected':
    print(f" Connected to {result['base_url']}")

# Trigger sync
client.federated_catalogs.sync("prod_analytics")

# Get sync stats
stats = client.federated_catalogs.get_stats("prod_analytics")
print(f"Last sync: {stats.last_sync}, Tables: {stats.tables_synced}")

# Delete federated catalog
client.federated_catalogs.delete("prod_analytics")
```

Views

Create SQL views that reference tables across catalogs.

Creating Views

```
# Get view client for a catalog
```

```
view_client = client.catalogs.views("my_catalog")

# Create a view
view = view_client.create(
    namespace="analytics",
    name="high_value_customers",
    sql="""
        SELECT customer_id, SUM(amount) as total_spent
        FROM sales.transactions
        WHERE amount > 1000
        GROUP BY customer_id
    """,
    properties={
        "description": "Customers with high transaction values",
        "owner": "analytics-team"
    }
)
```

Querying Views

```
# Get view definition
view = view_client.get("analytics", "high_value_customers")
print(f"SQL: {view.sql}")
print(f"Properties: {view.properties}")
```

CRUD Operations

Full create, read, update, delete support for core resources.

Tenant Management

```
# Create tenant
tenant = client.tenants.create("new_tenant")

# Update tenant
updated = client.tenants.update(tenant.id, properties={
    "department": "engineering",
    "cost_center": "CC-1234"
})

# Delete tenant
client.tenants.delete(tenant.id)
```

Warehouse Management

```
# Create warehouse
wh = client.warehouses.create_s3(
```

```
    "data_warehouse",
    "s3://my-bucket",
    access_key="...",
    secret_key="..."
)

# Update warehouse
updated_wh = client.warehouses.update(
    "data_warehouse",
    storage_config={"s3.bucket": "s3://new-bucket"}
)

# Delete warehouse
client.warehouses.delete("data_warehouse")
```

Catalog Management

```
# Create catalog
cat = client.catalogs.create("analytics", "data_warehouse")

# Update catalog
updated_cat = client.catalogs.update("analytics", properties={
    "owner": "data-team",
    "sla": "gold"
})

# Delete catalog
client.catalogs.delete("analytics")
```

Best Practices

Federated Catalogs

Test connections before relying on federated catalogs in production

Monitor sync stats to ensure data freshness

Use credentials securely - store tokens in environment variables

Set appropriate properties for tracking ownership and metadata

Views

Document SQL logic in view properties

Version control view definitions outside Pangolin

Test views against sample data before deployment

Use qualified names to avoid ambiguity across catalogs

Source: git_operations.md

Git-like Operations

Starting with version 0.2.0, `pypangolin` supports Git-like semantics for managing table versions, branches, tags, and merges. These features allow for powerful data versioning workflows directly from Python.

Branching

Create and manage branches to isolate changes or experiment with data.

```
from pypangolin import PangolinClient

client = PangolinClient(...)

# Create a new branch 'dev' from 'main'
# Branch creation is scoped to a catalog
dev_branch          =          client.branches.create("dev",          from_branch="main",
catalog_name="my_catalog")
print(f"Created branch {dev_branch.name} at commit {dev_branch.head_commit_id}")

# List active branches
branches = client.branches.list(catalog_name="my_catalog")
for b in branches:
    print(f"- {b.name} ({b.branch_type})")

# Get branch details
branch = client.branches.get("dev")
```

Commits & History

Inspect the history of changes on a branch.

```
# List commits on the 'main' branch
commits = client.branches.list_commits("main", catalog_name="my_catalog")

for commit in commits:
    print(f"[{commit.id[:8]}] {commit.message} - {commit.timestamp}")
```

Merging

Merge changes from one branch into another. Pangolin handles conflict detection for schemas and data.

```
# Merge 'dev' into 'main'
```

```
try:
    operation = client.branches.merge(
        source_branch="dev",
        target_branch="main",
        catalog_name="my_catalog"
    )

    if operation.status == "merged":
        print("Merge successful!")

    elif operation.status == "conflicted":
        print(f"Merge conflict detected! Operation ID: {operation.id}")

        # List conflicts
        conflicts = client.merge_operations.list_conflicts(operation.id)
        for c in conflicts:
            print(f"Conflict in {c.asset_name}: {c.conflict_type}")

        # Resolve conflicts (Example: 'source' wins)
        for c in conflicts:
            client.merge_operations.resolve_conflict(c.id, resolution="source")

        # Complete the merge
        client.merge_operations.complete(operation.id)
        print("Merge completed after resolution.")

except Exception as e:
    print(f"Merge failed: {e}")
```

Tagging

Tag specific points in history for releases or snapshots.

```
# Create a tag on the current head of 'main'
main_branch = client.branches.get("main")
if main_branch.head_commit_id:
    tag = client.tags.create(
        "v1.0.0",
        commit_id=main_branch.head_commit_id,
        catalog_name="my_catalog"
    )
    print(f"Tagged release {tag.name}")

# List tags
tags = client.tags.list(catalog_name="my_catalog")
```

Source: governance.md

Governance & Security

Pangolin provides robust governance features including Role-Based Access Control (RBAC), Service Users for automation, and Business Metadata for asset enrichment.

Role-Based Access Control (RBAC)

Manage who can do what with Roles and Permissions.

Roles

```
from pypangolin import PangolinClient

client = PangolinClient(...)

# Create a custom role
data_steward = client.roles.create("DataSteward", description="Responsible for data quality")

# List roles
for role in client.roles.list():
    print(f"{role.name}: {role.id}")
```

Permissions

Grant granular permissions to users or roles.

```
# Grant 'MANAGE_METADATA' on a specific catalog to the DataSteward role
client.permissions.grant(
    role_id=data_steward.id,
    action="MANAGE_METADATA",
    scope_type="catalog",
    scope_id="catalog_uuid"
)

# Assign role to a user
client.permissions.assign_role(user_id="user_uuid", role_id=data_steward.id)
```

Service Users

Create restricted users for automated pipelines (ETL/CI).

```
# Create a service user (defaults to 'tenant-user' role)
bot = client.service_users.create("etl-bot")
print(f"API Key: {bot.api_key}") # Only shown once!

# Rotate key if compromised
new_bot = client.service_users.rotate_key(bot.id)
print(f"New Key: {new_bot.api_key}")
```

Business Metadata

Enrich assets with searchable metadata, tags, and properties.

```
# Upsert metadata
client.metadata.upsert(
    asset_id="asset_uuid_or_name",
    tags=["verified", "pii"],
    properties={
        "owner": "Data Team",
        "sla": "gold"
    },
    description="Primary sales transaction table",
    discoverable=True
)

# Search/Get
meta = client.metadata.get("asset_uuid")
print(meta.tags)
```

Access Requests

Users can request access to discoverable assets they don't have permission to view.

```
# Request access
request = client.metadata.request_access("asset_uuid", motivation="Q4 Analysis")

# Admins see requests
requests = client.metadata.list_requests()
```

Source: hudi.md

Hudi Assets

Apache Hudi tables can be registered in Pangolin.

Note: Currently, `pypangolin` only supports **Registration** of existing Hudi tables. Native Python writing is limited (mostly requires Spark), so you should write your Hudi tables using Spark/Flink/Deltastreamer and then register them here.

Usage

```
from pypangolin.assets import HudiAsset

HudiAsset.register(
    client=client,
    catalog="my_catalog",
    namespace="my_ns",
    name="my_hudi_table",
```

```
    location="s3://bucket/hudi/base/path",
    properties={
        "hoodie.table.type": "COPY_ON_WRITE",
        "hoodie.table.version": "5"
    }
)
```

Recommended Properties

Property	Description	Example
`hoodie.table.type`	Table Type	`COPY_ON_WRITE` or `MERGE_ON_READ`
`hoodie.table.version`	Protocol version	`5`
`last_commit_time`	Timestamp of last commit	`20240101120000`
`record_count`	Approx record count	`5000`

Source: iceberg.md

PyIceberg Integration Guide

`pypangolin` provides seamless integration with PyIceberg, simplifying authentication and configuration.

Installation

To use PyIceberg with Pangolin, install `pypangolin` with the appropriate storage extras to ensure you have the necessary file system drivers.

```
# For AWS S3 / MinIO
pip install "pypangolin[s3]"

# For Azure Blob Storage
pip install "pypangolin[azure]"

# For Google Cloud Storage
pip install "pypangolin[gcs]"

# For all backends
pip install "pypangolin[all]"
```

Getting an Iceberg Catalog

Instead of manually configuring the PyIceberg `RestCatalog`, use the `get_iceberg_catalog` helper. This function:

- Constructs the correct REST URI.
- Injects your authentication token.

- Sets the `X-Pangolin-Tenant` header (critical for multi-tenancy).
- Enables **Credential Vending** (`X-Iceberg-Access-Delegation`) by default.

```
from pypangolin import PangolinClient, get_iceberg_catalog

# 1. Login to Pangolin
client = PangolinClient(uri="http://localhost:8080", username="admin", password="password")

# 2. Get the Iceberg Catalog Client
# This catalog is fully authenticated and scoped to your user/tenant.
catalog = get_iceberg_catalog("my_catalog", uri=client.uri, token=client.token)

# 3. Use standard PyIceberg API
namespaces = catalog.list_namespaces()
table = catalog.load_table("sales.transactions")
```

Using with MinIO (or On-Prem S3)

Pangolin supports on-premise S3-compatible storage like MinIO.

1. Warehouse Configuration

When creating an S3 Warehouse for MinIO, you **must** provide the `endpoint` URL so the Pangolin server knows where to write metadata files.

```
client.warehouses.create_s3(
    name="minio_wh",
    bucket="my-data",
    endpoint="http://minio:9000", # URL reachable by the Pangolin SERVER
    access_key="minio",
    secret_key="minio123",
    vending_strategy="AwsStatic"
)
```

2. Client Configuration

When initializing the Iceberg catalog, you may also need to provide the endpoint for the **Client** (PyIceberg) to read/write data, as it might differ from the server's internal URL (e.g., `localhost` vs `minio` container name).

```
catalog = get_iceberg_catalog(
    "minio_catalog",
    uri=client.uri,
    token=client.token,
    # Pass extra properties to PyIceberg
    **{"s3.endpoint": "http://localhost:9000"}
)
```

3. Client-Side Credentials (No Vending)

If you are connecting to a catalog that does not have a warehouse configured, or if you prefer to manage credentials manually:

Disable credential vending: `credential_vending=False`.

Pass your storage credentials directly in the `**properties` argument.**

Note: For write operations (creating tables), the Pangolin Server must also have write access to the metadata location (e.g., via IAM roles or environment variables).

```
catalog = get_iceberg_catalog(
    "manual_catalog",
    uri=client.uri,
    token=client.token,
    credential_vending=False,
    **{
        "s3.endpoint": "http://localhost:9000",
        "s3.access-key-id": "YOUR_ACCESS_KEY",
        "s3.secret-access-key": "YOUR_SECRET_KEY",
        "s3.region": "us-east-1"
    }
)
```

Source: json.md

JSON Assets

Manage JSON data files with Pangolin.

Usage

Use `JsonAsset` to write and register JSON files.

```
import pandas as pd
from pypangolin.assets import JsonAsset

df = pd.DataFrame({"key": ["value"]})

JsonAsset.write(
    client=client,
    catalog="my_catalog",
    namespace="my_ns",
    name="my_json_data",
    data=df,
    location="s3://bucket/data.json",
    orient="records", # Passed to to_json
    lines=True,
    storage_options={"key": "...", "secret": "..."},
    properties={"orient": "records", "compression": "gzip"}
```

```
)
```

Recommended Properties

Property	Description	Example
`file_size_bytes`	Size of the file	`4096`
`row_count`	Number of records	`250`
`json_type`	Structure of the JSON	`records` (lines), `table`, `split`
`compression`	Compression codec if applicable	`gzip`, `none`
`schema_json`	Inferred schema structure	`{"fields": [...]}`

Source: lance.md

Lance Assets

Pangolin supports Lance (and LanceDB) datasets.

Dependencies

Install helper library:

```
pip install "pypangolin[lance]"
# or
pip install lancedb
```

Usage

Use `LanceAsset` to write data via `lancedb` and register the dataset.

```
import pandas as pd
from pypangolin.assets import LanceAsset

df = pd.DataFrame({"vector": [[1.1, 2.2], [3.3, 4.4]], "label": ["a", "b"]})

# Writes a new Lance dataset and registers it
LanceAsset.write(
    client=client,
    catalog="my_catalog",
    namespace="my_ns",
    name="my_vectors",
    data=df,
    location="/tmp/my_vectors.lance", # Local or S3 URI
    properties={"vector_dim": "1536", "model": "openai-ada-002"}
)
```

Recommended Properties

Property	Description	Example
`dataset_version`	Lance format version	`2`
`row_count`	Number of vectors/rows	`10000`
`vector_dim`	Dimension of vectors	`1536`
`index_type`	Indexing algorithm used	`IVF_PQ`
`model_name`	Model used for embeddings	`text-embedding-ada-002`

Source: no_auth_mode.md

No Auth Mode Guide

This guide covers how to use `pypangolin` when the Pangolin API is running in No Auth Mode. This mode is ideal for local development and testing.

Prerequisites

Pangolin API running with `PANGOLIN_NO_AUTH=true`.

1. Connecting without Credentials

In No Auth mode, requests without an authentication token are automatically treated as requests from the default **Tenant Admin**.

```
from pypangolin import PangolinClient

# Connect without credentials
client = PangolinClient(uri="http://localhost:8080")

# You are now acting as the default Tenant Admin
# Tenant ID: 00000000-0000-0000-0000-000000000000
```

2. Default Admin Actions

You can immediately create resources without any setup.

```
# Create a Warehouse
# Note: You still need to provide credentials for the STORAGE backend (e.g. MinIO/S3),
# but they are stored in the catalog, not used for API access.
wh = client.warehouses.create_s3(
    name="local_dev_wh",
    bucket="dev-bucket",
    access_key="minio",
    secret_key="minio123"
)

# Create a Catalog
```

```
catalog = client.catalogs.create("sandbox", warehouse="local_dev_wh")

# Create a Namespace
ns = client.catalogs.namespaces("sandbox").create(["experiments"])
print(f"Created: {ns.name}")
```

3. Creating Users

Even in No Auth mode, you can create users to test permissions or simulate a multi-user environment.

```
user = client.users.create(
    username="analyst_bob",
    email="bob@example.com",
    role="tenant-user",
    tenant_id="00000000-0000-0000-0000-000000000000",
    password="any_password"
)
```

4. Masquerading (Simulating Users)

To test as a specific user (e.g., to verify they CANNOT delete a catalog), you must authenticate.

Note: In the current version, the `/login` endpoint still validates passwords if called. To bypass this for testing without passwords, ensure you create users with known passwords or use the default admin context.

```
# Authenticate as the new user
bob_client = PangolinClient(
    uri="http://localhost:8080",
    username="analyst_bob",
    password="any_password", # Must match what was created
    tenant_id="00000000-0000-0000-0000-000000000000"
)

# Bob is restricted by TenantUser role
try:
    bob_client.catalogs.create("cannot_do_this", warehouse="local_dev_wh")
except Exception as e:
    print("Permission Denied as expected")
```

Source: other.md

Other Assets

Pangolin supports a wide variety of generic assets beyond standard table formats.

Available Types

`NimbleAsset`

`MlModelAsset`

`DirectoryAsset`

`VideoAsset`

`ImageAsset`

`DbConnectionString`

`OtherAsset` (Catch-all)

Usage

All these assets follow the same registration pattern.

```
from pypangolin.assets import MlModelAsset, VideoAsset, DbConnectionString

# Register an ML Model
MlModelAsset.register(
    client=client,
    catalog="ml_catalog",
    namespace="models",
    name="churn_predictor",
    location="s3://bucket/models/churn/v1",
    properties={
        "framework": "pytorch",
        "version": "1.0.0",
        "accuracy": "0.95"
    }
)

# Register a Video
VideoAsset.register(
    client=client,
    catalog="media_catalog",
    namespace="raw",
    name="demo_recording",
    location="s3://bucket/videos/demo.mp4",
    properties={
        "duration_seconds": "120",
        "resolution": "1080p",
        "codec": "h264"
    }
)

# Register a Database Connection
DbConnectionString.register(
    client=client,
    catalog="infra_catalog",
    namespace="connections",
```

```
    name="prod_postgres",
    location="postgresql://user:pass@host:5432/db", # Be careful with secrets!
    properties={
        "environment": "production",
        "db_type": "postgres"
    }
)
```

Recommended Properties

ML Models

Property	Description	Example
`framework`	ML Framework	`pytorch`, `tensorflow`, `sklearn`
`version`	Model version	`1.0.0`
`accuracy`	Performance metric	`0.98`
`input_shape`	Input tensor shape	`(1, 3, 224, 224)`

Video Files

Property	Description	Example
`duration_seconds`	Length of video	`3600`
`resolution`	Video resolution	`1920x1080`
`codec`	Encoding codec	`h264`, `av1`
`frame_rate`	Frames per second	`30`, `60`

Images

Property	Description	Example
`width`	Image width (px)	`1024`
`height`	Image height (px)	`768`
`format`	File format	`png`, `jpg`, `webp`

Database Connections

Property	Description	Example
`db_type`	Database engine	`postgres`, `mysql`, `snowflake`
`environment`	Deployment env	`prod`, `staging`
`host`	Hostname (if not in URI)	`db.example.com`

Source: paimon.md

Apache Paimon Assets

Manage Apache Paimon tables in Pangolin.

Dependencies

Install `pypaimon`:

```
pip install "pypangolin[paimon]"
```

Usage

You can register existing Paimon tables. Writing support is experimental via `pypaimon`.

Registration

```
from pypangolin.assets import PaimonAsset

PaimonAsset.register(
    client=client,
    catalog="my_catalog",
    namespace="my_ns",
    name="my_paimon_table",
    location="s3://bucket/paimon/table",
    properties={
        "snapshot.id": "1",
        "schema.id": "0"
    }
)
```

Recommended Properties

Property	Description	Example
`snapshot.id`	Current snapshot ID	`123`
`schema.id`	Current schema ID	`5`
`primary_keys`	Primary Keys	`id, user_id`
`partition_keys`	Partition Keys	`dt, hr`
`file_format`	Underlying file format	`orc`, `parquet`

Source: parquet.md

Parquet Assets

`pypangolin` simplifies the management of Parquet files as managed assets in your catalog.

Dependencies

Requires `pyarrow` (included in `pypangolin[all]` or base install).

Usage

Use `ParquetAsset` to write a DataFrame (Pandas/Polars/PyArrow) to a Parquet file and register it.

```
import pandas as pd
from pypangolin.assets import ParquetAsset

df = pd.DataFrame({"a": [1, 2], "b": [3, 4]})

ParquetAsset.write(
    client=client,
    catalog="my_catalog",
    namespace="my_ns",
    name="my_parquet_data",
    data=df,
    location="s3://bucket/data.parquet",
    storage_options={"key": "...", "secret": "..."},
    properties={"compression": "snappy", "rows": "100"}
)
```

Recommended Properties

Property	Description	Example
`file_size_bytes`	Size of the file	`1024`
`row_count`	Number of rows	`500`
`compression`	Compression codec used	`snappy`, `gzip`, `zstd`
`columns`	JSON list of column names	`["col1", "col2"]`
`schema_json`	Full schema definition	`{"fields": [...]}`

Source: vortex.md

Vortex Assets

Support for Vortex file format assets.

Dependencies

Install `vortex-data` (or similar) manually. Currently `pypangolin` supports registration only.

Usage

```
from pypangolin.assets import VortexAsset

VortexAsset.register(
    client=client,
    catalog="my_catalog",
    namespace="my_ns",
    name="my_vortex_file",
    location="s3://bucket/data.vortex",
    properties={
        "vortex.version": "1.0",
        "row_count": "1000"
    }
)
```

Recommended Properties

Property	Description	Example
`vortex.version`	Format version	`1.0`
`row_count`	Number of rows	`1000`
`compression`	Compression used	`zstd`
`schema_description`	Text description of schema	`User events log`