Use ADBC Connection Profiles in Python

Ian Cook
Apr 15, 2026

ADBC connection profiles let you define reusable, named connection configurations in TOML files. They serve the same purpose as ODBC Data Source Names (DSNs): instead of embedding driver names, URIs, credentials, and other options directly in your application code, you store them in a profile that can be referenced by name. This decouples connection configuration from application logic, making it easy to switch between environments (dev, staging, production) without changing code.

Connection profiles are an ADBC standard, not a Python-specific feature. They work across all languages that support ADBC, including Python, R, C/C++, Go, Java, JavaScript, and Ruby.

Profiles are especially valuable with ADBC drivers that expose many options. Rather than repeating these options in every script, you configure them once in a profile.

In this notebook, you will:

1. Create two connection profiles that point to separate "dev" and "prod" databases.
2. Use an environment variable to keep the password out of the profile files.
3. Connect to each environment by profile name using identical client code.

Requirements:

- Python 3
- Docker

Setup

Install the required dependencies:

```
%pip install -q dbc 'adbc-driver-manager>=1.11.0' pyarrow
```

```
Note: you may need to restart the kernel to use updated packages.
```

Install the Flight SQL ADBC driver:

```
!dbc install -q flightsql
```

```
⌧]11;?⌧\
```

Import the required modules:

```python
import os
import platform
import shutil
from pathlib import Path

from adbc_driver_manager import dbapi
```

Start Database Servers

Start two GizmoSQL instances to simulate separate dev and prod environments. GizmoSQL is a lightweight SQL server powered by DuckDB that supports Arrow Flight SQL, which means you can connect to it using the ADBC driver for Flight SQL. Each instance is initialized with TPC-H sample data:

```
!docker run -d --rm -it --init -p 31337:31337 --name gizmosql_dev \
    -e DATABASE_FILENAME=dev.db \
    -e TLS_ENABLED=0 \
    -e GIZMOSQL_PASSWORD=gizmosql_password \
    -e PRINT_QUERIES=1 \
    -e INIT_SQL_COMMANDS='CALL dbgen(sf=0.01);' \
    --pull always gizmodata/gizmosql:latest-slim
```

```
latest-slim: Pulling from gizmodata/gizmosql
Digest: sha256:97113b5ba52adee8cbb82c563bc418e6c682d597e3c6ea30eb075f99c8864686
Status: Image is up to date for gizmodata/gizmosql:latest-slim

45a3214ecb763433f5c7bfce583a8d767cae360e97a50cd19f881e651e2fb5c6
```

```
!docker run -d --rm -it --init -p 31338:31337 --name gizmosql_prod \
    -e DATABASE_FILENAME=prod.db \
    -e TLS_ENABLED=0 \
    -e GIZMOSQL_PASSWORD=gizmosql_password \
    -e PRINT_QUERIES=1 \
    -e INIT_SQL_COMMANDS='CALL dbgen(sf=0.1);' \
    --pull always gizmodata/gizmosql:latest-slim
```

```
latest-slim: Pulling from gizmodata/gizmosql

Digest: sha256:97113b5ba52adee8cbb82c563bc418e6c682d597e3c6ea30eb075f99c8864686
Status: Image is up to date for gizmodata/gizmosql:latest-slim

574147bb27cc5dea86aef794afbf14979fcecfadfa862456fb8dc0b5f5da5f37
```

Wait for both servers to be ready:

```python
import time

time.sleep(10)
```

Install Connection Profiles

ADBC connection profiles are TOML files stored in a platform-specific user configuration directory:

- **macOS**: `~/Library/Application Support/ADBC/Profiles/`
- **Linux**: `~/.config/adbc/profiles/`
- **Windows**: `%LOCALAPPDATA%\ADBC\Profiles\`

When you connect using `profile://gizmosql_dev`, the ADBC driver manager looks for a file named `gizmosql_dev.toml` in this directory and applies all the options it contains.

This recipe includes two pre-created profile files in the `data/connection-profiles/` directory. Copy them to the user configuration directory:

```python
system = platform.system()
if system == "Darwin":
    profile_dir = Path.home() / "Library" / "Application Support" / "ADBC" / "Profiles"
elif system == "Windows":
    profile_dir = Path(os.environ["LOCALAPPDATA"]) / "ADBC" / "Profiles"
else:
    xdg = os.environ.get("XDG_CONFIG_HOME", Path.home() / ".config")
    profile_dir = Path(xdg) / "adbc" / "profiles"

profile_dir.mkdir(parents=True, exist_ok=True)

data_dir = Path("../data/connection-profiles")
home = str(Path.home())
for profile_file in data_dir.glob("*.toml"):
    shutil.copy(profile_file, profile_dir / profile_file.name)
    dest = str(profile_dir / profile_file.name).replace(home, "~")
    print(f"Installed {profile_file.name} -> {dest}")
```

```
Installed gizmosql_prod.toml -> ~/Library/Application Support/ADBC/Profiles/gizmosql_prod.toml
Installed gizmosql_dev.toml -> ~/Library/Application Support/ADBC/Profiles/gizmosql_dev.toml
```

Here are the first few lines of `gizmosql_dev.toml`, which contain the most important settings:

```
profile_version = 1
driver = "flightsql"

[Options]
# Connection
uri = "grpc+tcp://localhost:31337"
username = "gizmosql_user"
password = "{{ env_var(GIZMOSQL_PASSWORD) }}"
```

Notice that the password uses `{{ env_var(GIZMOSQL_PASSWORD) }}` instead of a hardcoded value. At connection time, ADBC substitutes the value of the `GIZMOSQL_PASSWORD` environment variable. The full profile files also include Flight SQL driver options for TLS, timeouts, and other settings, all set to their defaults.

The two profiles are nearly identical. The only difference is the URI, which points each profile at a different GizmoSQL instance. In `gizmosql_dev.toml`:

```
uri = "grpc+tcp://localhost:31337"
```

In `gizmosql_prod.toml`:

```
uri = "grpc+tcp://localhost:31338"
```

Set the Password Environment Variable

The profiles reference the password via `{{ env_var(GIZMOSQL_PASSWORD) }}`. Set this environment variable so ADBC can substitute it at connection time. In a real deployment, this would be set in your shell environment or a secrets manager, not in code:

```python
os.environ["GIZMOSQL_PASSWORD"] = "gizmosql_password"
```

Connect Using Profiles

Connect to the dev environment by passing the profile URI. All driver options, credentials, and connection details are resolved from the `gizmosql_dev.toml` profile:

```python
with (
    dbapi.connect("profile://gizmosql_dev") as dev_conn,
    dev_conn.cursor() as dev_cursor,
):
    dev_cursor.execute("SELECT count(*) AS row_count FROM lineitem")
    result = dev_cursor.fetch_arrow_table()
    print(f"Dev lineitem rows: {result.column('row_count')[0].as_py()}")
```

```
Dev lineitem rows: 60175
```

Connect to the prod environment using the same code structure. Only the profile name changes:

```python
with (
    dbapi.connect("profile://gizmosql_prod") as prod_conn,
    prod_conn.cursor() as prod_cursor,
):
    prod_cursor.execute("SELECT count(*) AS row_count FROM lineitem")
    result = prod_cursor.fetch_arrow_table()
    print(f"Prod lineitem rows: {result.column('row_count')[0].as_py()}")
```

```
Prod lineitem rows: 600572
```

The dev database was generated at scale factor 0.01 and the prod database at 0.1, so prod has roughly 10x more rows. The application code is identical in both cases -- only the profile name differs.

Write Generic Client Code

Connection profiles make it possible to write fully generic client code. The profile name can be read from an environment variable, a configuration file, or a command-line argument. The code below works with any ADBC-compatible database without modification:

```python
def get_row_count(profile_name: str, table_name: str) -> int:
    with (
        dbapi.connect(f"profile://{profile_name}") as conn,
        conn.cursor() as cursor,
    ):
        cursor.execute(f"SELECT count(*) AS n FROM {table_name}")
        return cursor.fetch_arrow_table().column("n")[0].as_py()


for profile in ["gizmosql_dev", "gizmosql_prod"]:
    print(f"{profile}: {get_row_count(profile, 'orders')} orders")
```

```
gizmosql_dev: 15000 orders
gizmosql_prod: 150000 orders
```

Cleanup

Stop and remove the Docker containers:

```
!docker stop gizmosql_dev gizmosql_prod
```

```
gizmosql_dev

gizmosql_prod
```

Optionally, remove the installed profile files:

```python
(profile_dir / "gizmosql_dev.toml").unlink(missing_ok=True)
(profile_dir / "gizmosql_prod.toml").unlink(missing_ok=True)
```

Try It Yourself

This recipe uses two instances of the same database engine, but connection profiles are even more powerful when used across completely different databases. Because ADBC provides a common API across drivers, the same client code can target Snowflake, DuckDB, PostgreSQL, Databricks, or any other ADBC-supported database — only the profile changes. Some things to try:

- **Point profiles at different engines**: Create a profile for a local DuckDB instance and another for a cloud data warehouse like Snowflake. The same `dbapi.connect(f"profile://{name}")` call works for both.
- **Use a SQL transpiler**: Different databases have different SQL dialects. Pair connection profiles with a query transpiler like SQLGlot to automatically translate your SQL to the dialect of whichever database the profile connects to, making your client code fully engine-agnostic.
- **Parameterize the profile name**: Read the profile name from an environment variable (`os.environ["ADBC_PROFILE"]`) or a CLI argument so the same script can target dev, staging, or production without any code changes.
- **Explore more profile features**: Connection profiles support additional search paths, absolute file paths (`profile:///path/to/profile.toml`), and the `ADBC_PROFILE_PATH` environment variable for custom profile directories.