Build a Python App with CockroachDB and psycopg2

psycopg3 psycopg2 SQLAlchemy Django asyncpg

Tip:

Check out our developer courses at Cockroach University.
This tutorial shows you how build a simple Python application with CockroachDB and the psycopg2 driver.

Step 1. Start CockroachDB

Use CockroachDB Serverless Use a Local Cluster

Choose your installation method

You can create a CockroachDB Serverless cluster using either the CockroachDB Cloud Console, a web-based graphical user interface (GUI) tool, or `ccloud`, a command-line interface (CLI) tool.

Use the Cloud Console (GUI) **Use `ccloud` (CLI)**

Create a free cluster

Note:

Organizations without billing information on file can only create one CockroachDB Serverless cluster.

1. If you haven't already, sign up for a CockroachDB Cloud account.
2. Log in to your CockroachDB Cloud account.
3. On the **Clusters** page, click **Create Cluster**.
4. On the **Create your cluster** page, select **Serverless**.
5. Click **Create cluster**.

Your cluster will be created in a few seconds and the **Create SQL user** dialog will display.

Create a SQL user

The **Create SQL user** dialog allows you to create a new SQL user and password.

1. Enter a username in the **SQL user** field or use the one provided by default.
2. Click **Generate & save password**.
3. Copy the generated password and save it in a secure location.
4. Click **Next**.

   Currently, all new SQL users are created with admin privileges. For more information and to change the default settings, see [Manage SQL users on a cluster.

Get the root certificate

The **Connect to cluster** dialog shows information about how to connect to your cluster.

1. Select **General connection string** from the **Select option** dropdown.
2. Open a new terminal on your local machine, and run the **CA Cert download command** provided in the **Download CA Cert** section. The client driver used in this tutorial requires this certificate to connect to CockroachDB Cloud.

Get the connection string

Open the **General connection string** section, then copy the connection string provided and save it in a secure location.

Note:

The connection string is pre-populated with your username, password, cluster name, and other details. Your password, in particular, will be provided *only once*. Save it in a secure place (Cockroach Labs recommends a password manager) to connect to your cluster in the future. If you forget your password, you can reset it by going to the **SQL Users** page for the cluster, found at `https://cockroachlabs.cloud/cluster/<CLUSTER ID>/users`.

Step 2. Get the sample code

Clone the sample code's GitHub repo:

```
git clone https://github.com/cockroachlabs/hello-world-python-psycopg2
```

The sample code in `example.py` does the following:

- Creates an `accounts` table and inserts some rows
- Transfers funds between two accounts inside a transaction
- Deletes the accounts from the table before exiting so you can re-run the example code

To handle transaction retry errors, the code uses an application-level retry loop that, in case of error, sleeps before trying the funds transfer again. If it encounters another retry error, it sleeps for a longer interval, implementing exponential backoff.

Step 3. Install the psycopg2 driver

`psycopg2-binary` is the sample app's only third-party module dependency.

To install `psycopg2-binary`, run the following command:

```
pip install psycopg2-binary
```

For other ways to install psycopg2, see the official documentation.

Step 4. Run the code

1. Set the `DATABASE_URL` environment variable to the connection string to your cluster:

```
2. export DATABASE_URL="{connection-string}"
```

   Where `{connection-string}` is the connection string you copied earlier.

   The app uses the connection string saved to the `DATABASE_URL` environment variable to connect to your cluster and execute the code.

3. Run the code:

```
4. cd hello-world-python-psycopg2

5. python example.py
```

The output should show the account balances before and after the funds transfer:

```
Balances at Thu Aug  4 15:51:03 2022:

account id: 2e964b45-2034-49a7-8ab8-c5d0082b71f1  balance: $1000

account id: 889cb1eb-b747-46f4-afd0-15d70844147f  balance: $250

Balances at Thu Aug  4 15:51:03 2022:

account id: 2e964b45-2034-49a7-8ab8-c5d0082b71f1  balance: $900

account id: 889cb1eb-b747-46f4-afd0-15d70844147f  balance: $350
```

Build a Simple CRUD Python App with CockroachDB and SQLAlchemy

psycopg3 psycopg2 SQLAlchemy Django asyncpg

Tip:

Check out our developer courses at Cockroach University.
This tutorial shows you how build a simple CRUD Python application with CockroachDB and the SQLAlchemy ORM.

Step 1. Start CockroachDB

Use CockroachDB Serverless Use a Local Cluster

Choose your installation method

You can create a CockroachDB Serverless cluster using either the CockroachDB Cloud Console, a web-based graphical user interface (GUI) tool, or `ccloud`, a command-line interface (CLI) tool.

Use the Cloud Console (GUI) **Use `ccloud` (CLI)**

Create a free cluster

Note:

Organizations without billing information on file can only create one CockroachDB Serverless cluster.

1. If you haven't already, sign up for a CockroachDB Cloud account.
2. Log in to your CockroachDB Cloud account.
3. On the **Clusters** page, click **Create Cluster**.
4. On the **Create your cluster** page, select **Serverless**.
5. Click **Create cluster**.

Your cluster will be created in a few seconds and the **Create SQL user** dialog will display.

Create a SQL user

The **Create SQL user** dialog allows you to create a new SQL user and password.

1. Enter a username in the **SQL user** field or use the one provided by default.
2. Click **Generate & save password**.
3. Copy the generated password and save it in a secure location.
4. Click **Next**.

   Currently, all new SQL users are created with admin privileges. For more information and to change the default settings, see [Manage SQL users on a cluster.

Get the root certificate

The **Connect to cluster** dialog shows information about how to connect to your cluster.

1. Select **General connection string** from the **Select option** dropdown.
2. Open a new terminal on your local machine, and run the **CA Cert download command** provided in the **Download CA Cert** section. The client driver used in this tutorial requires this certificate to connect to CockroachDB Cloud.

Get the connection string

Open the **General connection string** section, then copy the connection string provided and save it in a secure location.

Note:

The connection string is pre-populated with your username, password, cluster name, and other details. Your password, in particular, will be provided *only once*. Save it in a secure place (Cockroach Labs recommends a password manager) to connect to your cluster in the future. If you forget your password, you can reset it by going to the **SQL Users** page for the cluster, found at `https://cockroachlabs.cloud/cluster/<CLUSTER ID>/users`.

Step 2. Get the code

Clone the code's GitHub repo:

```
git clone https://github.com/cockroachlabs/example-app-python-sqlalchemy/
```

The project has the following directory structure:

```
├── README.md

├── dbinit.sql

├── main.py

├── models.py

└── requirements.txt
```

The `requirements.txt` file includes the required libraries to connect to CockroachDB with SQLAlchemy, including the `sqlalchemy-cockroachdb` Python package, which accounts for some differences between CockroachDB and PostgreSQL:

```
psycopg2-binary

sqlalchemy

sqlalchemy-cockroachdb
```

The `dbinit.sql` file initializes the database schema that the application uses:

```
CREATE TABLE accounts (

    id UUID PRIMARY KEY,

    balance INT8

);
```

The `models.py` uses SQLAlchemy to map the `Accounts` table to a Python object:

```
from sqlalchemy import Column, Integer

from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.orm import declarative_base


Base = declarative_base()



class Account(Base):
    """The Account class corresponds to the "accounts" database table.
    """
    __tablename__ = 'accounts'
    id = Column(UUID(as_uuid=True), primary_key=True)
    balance = Column(Integer)

```

The `main.py` uses SQLAlchemy to map Python methods to SQL operations:

```
"""This simple CRUD application performs the following operations sequentially:
    1. Creates 100 new accounts with randomly generated IDs and randomly-computed balance amounts.
    2. Chooses two accounts at random and takes half of the money from the first and deposits it
     into the second.
    3. Chooses five accounts at random and deletes them.
"""

from math import floor
import os
```

```python
import random
import uuid

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy_cockroachdb import run_transaction
from sqlalchemy.orm.exc import NoResultFound, MultipleResultsFound

from models import Account

# The code below inserts new accounts.


def create_accounts(session, num):
    """Create N new accounts with random account IDs and account balances.
    """
    print("Creating new accounts...")
    new_accounts = []
    while num > 0:
        account_id = uuid.uuid4()
        account_balance = floor(random.random()*1_000_000)
        new_accounts.append(Account(id=account_id, balance=account_balance)
)
        seen_account_ids.append(account_id)
        print(f"Created new account with id {account_id} and balance {account_balance}.")

        num = num - 1

    session.add_all(new_accounts)




def transfer_funds_randomly(session, one, two):

    """Transfer money between two accounts.

    """

    try:

        source = session.query(Account).filter(Account.id == one).one()

    except NoResultFound:

        print("No result was found")

    except MultipleResultsFound:

        print("Multiple results were found")

    dest = session.query(Account).filter(Account.id == two).first()

    print(f"Random account balances:\nAccount {one}: {source.balance}\nAccount {two}: {dest.balance}")


    amount = floor(source.balance/2)

    print(f"Transferring {amount} from account {one} to account {two}...")


    # Check balance of the first account.

    if source.balance < amount:

        raise ValueError(f"Insufficient funds in account {one}")
    source.balance -= amount

    dest.balance += amount


    print(f"Transfer complete.\nNew balances:\nAccount {one}: {source.balan
ce}\nAccount {two}: {dest.balance}")




def delete_accounts(session, num):

    """Delete N existing accounts, at random.

    """

    print("Deleting existing accounts...")

    delete_ids = []

    while num > 0:

        delete_id = random.choice(seen_account_ids)

        delete_ids.append(delete_id)

        seen_account_ids.remove(delete_id)

        num = num - 1


    accounts = session.query(Account).filter(Account.id.in_(delete_ids)).al
l()


    for account in accounts:

        print(f"Deleted account {account.id}.")

        session.delete(account)

```

```python
if __name__ == '__main__':
    # For cockroach demo:
    # DATABASE_URL=postgresql://demo:<demo_password>@127.0.0.1:26257?sslmode=require
    # For CockroachCloud:
    # DATABASE_URL=postgresql://<username>:<password>@<globalhost>:26257/<cluster_name>.defaultdb?sslmode=verify-full&sslrootcert=<certs_dir>/<ca.crt>
    db_uri = os.environ['DATABASE_URL'].replace("postgresql://", "cockroachdb://")
    try:
        engine = create_engine(db_uri, connect_args={"application_name":"docs_simplecrud_sqlalchemy"})
    except Exception as e:
        print("Failed to connect to database.")
        print(f"{e}")


    seen_account_ids = []


    run_transaction(sessionmaker(bind=engine),
                    lambda s: create_accounts(s, 100))


    from_id = random.choice(seen_account_ids)
    to_id = random.choice([id for id in seen_account_ids if id != from_id])
    run_transaction(sessionmaker(bind=engine),
                    lambda s: transfer_funds_randomly(s, from_id, to_id))


    run_transaction(sessionmaker(bind=engine), lambda s: delete_accounts(s, 5))
```

`main.py` also executes the `main` method of the program.

Step 3. Install the application requirements

This tutorial uses `virtualenv` for dependency management.

1. Install `virtualenv`:

```
2. pip install virtualenv
```

3. At the top level of the app's project directory, create and then activate a virtual environment:

```
4. virtualenv env

5. source env/bin/activate
```

6. Install the required modules to the virtual environment:

```
7. pip install -r requirements.txt
```

Step 4. Initialize the database

1. Set the `DATABASE_URL` environment variable to the connection string for your cluster:

```
2. export DATABASE_URL="{connection-string}"
```

   Where `{connection-string}` is the connection string you copied earlier.

3. To initialize the example database, use the `cockroach sql` command to execute the SQL statements in the `dbinit.sql` file:

```
4. cat dbinit.sql | cockroach sql --url $DATABASE_URL
```

The SQL statement in the initialization file should execute:

```
CREATE TABLE



Time: 102ms
```

Step 5. Run the code

`main.py` uses the connection string saved to the `DATABASE_URL` environment variable to connect to your cluster and execute the code.

Note:

The example application uses the general connection string, which begins with `postgresql://` but modifies it so it uses the `cockroachdb://` prefix. It does this so SQLAlchemy will use the CockroachDB SQLAlchemy adapter.

```
db_uri = os.environ['DATABASE_URL'].replace("postgresql://", "cockroachdb:/
/")
```

Run the app:

```
python main.py
```

The application will connect to CockroachDB, and then perform some simple row inserts, updates, and deletes.

The output should look something like the following:

```
Creating new accounts...

Created new account with id 3a8b74c8-6a05-4247-9c60-24b46e3a88fd and balanc
e 248835.

Created new account with id c3985926-5b77-4c6d-a73d-7c0d4b2a51e7 and balanc
e 781972.

...

Created new account with id 7b41386c-11d3-465e-a2a0-56e0dcd2e7db and balanc
e 984387.
```

```
Random account balances:

Account 7ad14d02-217f-48ca-a53c-2c3a2528a0d9: 800795

Account 4040aeba-7194-4f29-b8e5-a27ed4c7a297: 149861

Transferring 400397 from account 7ad14d02-217f-48ca-a53c-2c3a2528a0d9 to ac
count 4040aeba-7194-4f29-b8e5-a27ed4c7a297...

Transfer complete.

New balances:

Account 7ad14d02-217f-48ca-a53c-2c3a2528a0d9: 400398

Account 4040aeba-7194-4f29-b8e5-a27ed4c7a297: 550258

Deleting existing accounts...

Deleted account 41247e24-6210-4032-b622-c10b3c7222de.

Deleted account 502450e4-6daa-4ced-869c-4dff62dc52de.

Deleted account 6ff06ef0-423a-4b08-8b87-48af2221bc18.

Deleted account a1acb134-950c-4882-9ac7-6d6fbdaaaee1.

Deleted account e4f33c55-7230-4080-b5ac-5dde8a7ae41d.
```

In a SQL shell connected to the cluster, you can verify that the rows were inserted, updated, and deleted successfully:

```
SELECT COUNT(*) FROM accounts;

  count

---------

     95

(1 row)
```

Best practices

Use the `run_transaction` function

We strongly recommend using the `sqlalchemy_cockroachdb.run_transaction()` function as shown in the code samples on this page. This abstracts the details of transaction retries away from your application code. Transaction retries are more frequent in CockroachDB than in some other databases because we use optimistic concurrency control rather than locking. Because of this, a CockroachDB transaction may have to be tried more than once before it can commit. This is part of how we ensure that our transaction ordering guarantees meet the ANSI SERIALIZABLE isolation level.

In addition to the above, using `run_transaction` has the following benefits:

- Because it must be passed a sqlalchemy.orm.session.sessionmaker object (*not* a session), it ensures that a new session is created exclusively for use by the callback, which protects you from accidentally reusing objects via any sessions created outside the transaction.
- It abstracts away the client-side transaction retry logic from your application, which keeps your application code portable across different databases. For example, the sample code given on this page works identically when run against PostgreSQL (modulo changes to the prefix and port number in the connection string).

For more information about how transactions (and retries) work, see Transactions.

Avoid mutations of session and/or transaction state inside `run_transaction()`

In general, this is in line with the recommendations of the SQLAlchemy FAQs, which state (with emphasis added by the original author) that

As a general rule, the application should manage the lifecycle of the session *externally* to functions that deal with specific data. This is a fundamental separation of concerns which keeps data-specific operations agnostic of the context in which they access and manipulate that data.

and

Keep the lifecycle of the session (and usually the transaction) **separate and external**.

In keeping with the above recommendations from the official docs, we **strongly recommend** avoiding any explicit mutations of the transaction state inside the callback passed to `run_transaction`, since that will lead to breakage. Specifically, do not make calls to the following functions from inside `run_transaction`:

- `sqlalchemy.orm.Session.commit()` (or other variants of `commit()`): This is not necessary because `cockroachdb.sqlalchemy.run_transaction` handles the savepoint/commit logic for you.
- `sqlalchemy.orm.Session.rollback()` (or other variants of `rollback()`): This is not necessary because `cockroachdb.sqlalchemy.run_transaction` handles the commit/rollback logic for you.
- `Session.flush()`: This will not work as expected with CockroachDB because CockroachDB does not support nested transactions, which are necessary for `Session.flush()` to work properly. If the call to `Session.flush()` encounters an error and aborts, it will try to rollback. This will not be allowed by the currently-executing CockroachDB transaction created by `run_transaction()`, and will result in an error message like the following: `sqlalchemy.orm.exc.DetachedInstanceError: Instance <FooModel at 0x12345678> is not bound to a Session; attribute refresh operation cannot proceed (Background on this error at: http://sqlalche.me/e/bhk3)`.

Break up large transactions into smaller units of work

If you see an error message like `transaction is too large to complete; try splitting into pieces`, you are trying to commit too much data in a single transaction. As described in our Cluster Settings docs, the size limit for transactions is defined by the `kv.transaction.max_intents_bytes` setting, which defaults to 256 KiB. Although this setting can be changed by an admin, we strongly recommend against it in most cases.

Instead, we recommend breaking your transaction into smaller units of work (or "chunks"). A pattern that works for inserting large numbers of objects using `run_transaction` to handle retries automatically for you is shown below.

```
from sqlalchemy import create_engine, Column, Float, Integer

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import sessionmaker

from cockroachdb.sqlalchemy import run_transaction

from random import random
```

```
Base = declarative_base()


# The code below assumes you have run the following SQL statements.


# CREATE DATABASE pointstore;


# USE pointstore;


# CREATE TABLE points (

#     id INT PRIMARY KEY DEFAULT unique_rowid(),

#     x FLOAT NOT NULL,

#     y FLOAT NOT NULL,

#     z FLOAT NOT NULL

# );


engine = create_engine(

    # For cockroach demo:

    'cockroachdb://<username>:<password>@<hostname>:<port>/bank?sslmode=req
uire',

    echo=True                   # Log SQL queries to stdout

)



class Point(Base):
    __tablename__ = 'points'

    id = Column(Integer, primary_key=True)

    x = Column(Float)

    y = Column(Float)

    z = Column(Float)



def add_points(num_points):

    chunk_size = 1000        # Tune this based on object sizes.


    def add_points_helper(sess, chunk, num_points):

        points = []

        for i in range(chunk, min(chunk + chunk_size, num_points)):

            points.append(

                Point(x=random()*1024, y=random()*1024, z=random()*1024)

            )

        sess.bulk_save_objects(points)


    for chunk in range(0, num_points, chunk_size):

        run_transaction(

            sessionmaker(bind=engine),

            lambda s: add_points_helper(

                s, chunk, min(chunk + chunk_size, num_points)

            )
        )



add_points(10000)
```

Use `IMPORT` to read in large data sets

If you are trying to get a large data set into CockroachDB all at once (a bulk import), avoid writing client-side code that uses an ORM and use the `IMPORT` statement instead. It is much faster and more efficient than making a series of `INSERT`s and `UPDATE`s such as are generated by calls to `session.bulk_save_objects()`.

For more information about importing data from PostgreSQL, see Migrate from PostgreSQL.

For more information about importing data from MySQL, see Migrate from MySQL.

Prefer the query builder

In general, we recommend using the query-builder APIs of SQLAlchemy (e.g., `Engine.execute()`) in your application over the Session/ORM APIs if at all possible. That way, you know exactly what SQL is being generated and sent to CockroachDB, which has the following benefits:

- It's easier to debug your SQL queries and make sure they are working as expected.
- You can more easily tune SQL query performance by issuing different statements, creating and/or using different indexes, etc. For more information, see SQL Performance Best Practices.

Joins without foreign keys

SQLAlchemy relies on the existence of foreign keys to generate `JOIN` expressions from your application code. If you remove foreign keys from your schema, SQLAlchemy will not generate joins for you. As a workaround, you can create a "custom foreign condition" by adding a `relationship` field to your table objects, or do the equivalent work in your application.