Flask and Databases

To add database functionality to a Flask app, we will use SQLAlchemy.

SQLAlchemy is a Python SQL toolkit and object relational mapper (ORM) that enables Python to communicate with the SQL database system you prefer: MySQL, PostgreSQL, SQLite, and others. An ORM converts data between incompatible systems (object structure in Python, table structure in SQL database). SQLAlchemy is basically a **bridge** between Python and a SQL database.

Flask-SQLAlchemy is an *extension* for Flask that adds SQLAlchemy to your Flask app.

- SQLAlchemy documentation
- Flask-SQLAlchemy documentation
- Code for this chapter

Setup: Flask-SQLAlchemy

We will install the **Flask-SQLAlchemy** extension to enable us to work with a SQL database in Flask. There are many extensions for Flask; each one adds a different set of functions and capabilities. See the list of Flask extensions for more.

In Terminal, change into your Flask projects folder and **activate your virtual environment** there. Then install the extension at the command prompt — where you see `$` (Mac) or `C:\Users\yourname>` (Windows) —

```
pip install flask-sqlalchemy
```

We will use SQLite for database examples here. Although it's not necessary to use SQLAlchemy to interact with a SQLite database, learning to use SQLAlchemy gives you a skill set that can be applied to *any* SQL database system.

SQLAlchemy can bridge between Python and various different SQL database systems — some of which need an additional module, or library, to be installed. SQLite *does not* require an additional module — the `sqlite3` module is included in Python 3.x.

- Find other modules for other SQL databases.

Important

If you're using a MySQL or PostgreSQL database, you will need to install a DBAPI module such as `psycopg2` (PostgreSQL) or `PyMySQL` (MySQL).

Basics of using a database with Flask

You'll *connect* your Flask app to an existing SQL database. Connecting will require your own database username and database password, *unless* using SQLite.

Note

You *can* create the SQL database using Python, but *that is not required*. If you already have a database, all you need to worry about is how to connect it. If you *do* use Python to create a SQL database (and that's an "if," not a necessity), you will only do it once. You don't create the same database again and again. Yes, this seems like a no-brainer — but you need to think about what your code is *doing*.

Your database may have one table, or more than one table. That depends on what you need, or the structure of the existing SQL database. You'll need to know the table name(s). You'll need to know the field names (column headings) in each table.

Your app might only *read from* your SQL database. You can write SQL queries to accomplish this — using Flask-SQLAlchemy commands to do so. Note that you won't write a straightforward SQL query. Here is an example of Flask-SQLAlchemy syntax:

```
socks = Sock.query.filter_by(style='knee-high').order_by(Sock.name).all()
```

The Flask-SQLAlchemy statement *to the right of the equals sign*, above, is equivalent to this standard SQL statement:

```
SELECT * FROM socks WHERE style="knee-high" ORDER BY name
```

It is assumed you are familiar with how to write basic SQL queries.

- Details about writing queries with Flask-SQLAlchemy.

In addition to *reading from* your SQL database, your Flask app might allow people to *write to* the database. In that case, you will probably want people to log in securely. Alternatively, you might set up a Python script that updates your database on a regular schedule (e.g., writing in new records from a monthly data dump).

You might write a Python script to populate your database from the contents of a CSV file. This would be fairly simple if you only need to run it once. If you need to add records repeatedly (say, once per month) to an existing database, you might need to check whether you are *duplicating records that are already there*. If you need to check for existing records and update them, that's more challenging. You can handle each of these tasks within Flask, using route functions.

If people are *writing into* your database, you will want to give them a web form, or forms, for doing so. See Flask: Web Forms if you need to create a web form in your Flask app.

You will not necessarily need forms if your app only *reads from* the database, but it is possible you'll want to allow people to search for content, or to choose content from a menu using a `<select>` element in a form that queries the database. Then a form or forms will be required. Again, you will handle these tasks within Flask, using route functions.

Of course, you'll be using templates and all the other aspects of Flask covered in previous chapters here.

This overview should get you thinking about what you will need *your* Flask app to do, and how you will be querying the database, inserting new records, etc.

For all Python and SQL commands, refer to the links listed under "User's Guide" in the Flask-SQLAlchemy documentation.

How to connect a database to a Flask app

The first step, assuming you have a database, is getting your app (or a starter script for your app) to **connect** to the database. **Do this first.**

Here's a starter script for testing whether you can connect:

../python_code_examples/flask/databases/test_local_sqlite_db.py

```
1  """
2      test a SQLite database connection locally
3      assumes database file is in same location
4      as this .py file
5  """
6
7  from flask import Flask
8  from flask_sqlalchemy import SQLAlchemy
9  from sqlalchemy.sql import text
10
11 app = Flask(__name__)
12
13 # change to name of your database; add path if necessary
14 db_name = 'sockmarket.db'
15
16 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + db_name
17
18 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
19
20 # this variable, db, will be used for all SQLAlchemy commands
21 db = SQLAlchemy(app)
22
23 # NOTHING BELOW THIS LINE NEEDS TO CHANGE
24 # this route will test the database connection and nothing more
25 @app.route('/')
26 def testdb():
27     try:
28         db.session.query(text('1')).from_statement(text('SELECT 1')).all()
29         return '<h1>It works.</h1>'
30     except Exception as e:
31         # e holds description of the error
32         error_text = "<p>The error:<br>" + str(e) + "</p>"
33         hed = '<h1>Something is broken.</h1>'
34         return hed + error_text
35
36 if __name__ == '__main__':
37     app.run(debug=True)
```

The script above assumes that a SQLite database file (see filename on line 14) is in the same directory as the `.py` file. The script will work with any SQLite database file.

- Starter scripts for a MySQL database connection are here. More about this below.

Note that line 16 is the key to the connection — it contains the **database connection string,** which will be *different* depending on which SQL database system you are connecting to. The SQLite connection string is simpler than the others, so be sure to read on if you're using MySQL, PostgreSQL, or another system.

Run the script above:

```
python test_local_sqlite_db.py
```

Open `http://localhost:5000` in your web browser. If you see the text “It works.” — then all is well. Otherwise, you’ll see an error message that should enable you to resolve the problem.

SQLite resources

SQLite is a SQL database engine that is especially easy to work with because the database — regardless of its size or how many tables it might include — is in a single `.db` file. You can copy the file, upload it to a server, and so on — it is a standalone file.

Unlike other database systems, a SQLite database does not have a username or password for access to the database itself.

- SQLite homepage
- Download and install SQLite for your operating system (note: MacOS already has SQLite)
- Download the free DB Browser for SQLite (you can easily create tables by importing CSV files) — use this to create a new database, add tables, set data types for columns, etc.
- SQLite Tutorial
- SQLite can be used *without* SQLAlchemy: Using SQLite3 with Flask

Connecting to a MySQL database

Two scripts are provided to test a connection to a MySQL database. They are here. An additional Python module must be installed — PyMySQL — and a username and password **must** be included in the connection string (even an *empty* password has a place).

In addition, when running the MySQL database locally, a socket string must be included. This string will be very different on MacOS and Windows.

If you do not want to run the database locally, but instead you have the database on a remote server — while you are writing your Flask app (and testing it) on your own computer — you will need to remotely connect to your MySQL database.

- See other examples of connection strings.

The connection string

The database connection string requires a **strict syntax,** or it will not work. The SQLite string is by far the simplest:

```
'sqlite:///' + db_name
```

That assumes the `.db` file is adjacent to your `.py` file. If not, write the path as needed.

For other database systems, the connection string will be more complex:

```
'mysql+pymysql://' + username + ':' + password + '@' + server + database
```

Note, that example is for MySQL only; the protocol will be different for, say, PostgreSQL.

Setting environment variables

Instead of including username, password, and the whole database connection string *within a Python script*, you can set the complete string as an **environment variable.** Note that the connection string must be complete and correct for your configuration, as discussed in the previous section.

If you set an environment variable for the connection string, then add the following lines near the top of your script:

```
import os
# check for environment variable
if not os.getenv("DATABASE_URL"):
    raise RuntimeError("DATABASE_URL is not set")
```

Eliminate all lines that refer to username, password, server, and database name.

Change the `app.config` statement to this:

```
app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv("DATABASE_URL")
```

Look up how to set an environment variable for your operating system.

> **Note**
>
> `os` is a built-in Python module.

When setting up a Flask app on a server, there will be an option to set environment variables there. The lines in the Flask script referring to the environment variable will not change.

> **Important**
>
> Only one environment variable on your computer, or in an app, can be named `DATABASE_URL` — it's a variable name, so you can change the string to something else, and you can set as many different variables as you need.

> **Note**
>
> If you're using a SQLite database, don't bother with environment variables.

What could go wrong?

If you cannot get your Flask app to connect to your database, check the following:

- You forgot to install something (Flask-SQLAlchemy, or PyMySQL, etc.) in your Python virtual environment.
- Your virtual environment has not been activated.
- Your username and/or password for the database are wrong.
- Your database name is incorrect.
- On a remote server, permissions for the database user are not set correctly.
- For a local database, the socket does not match what you need on your computer.
- For a local MySQL database, you have not started the MySQL server.

Summary

This has been a basic introduction to *getting started* with Flask-SQLAlchemy and databases in Flask. The first step is to make sure you are able to **connect** successfully to the database you want to use.

A successful connection depends on **which type of SQL system** your database was built in. In this chapter, SQLite and MySQL are covered. PostgreSQL is another popular option but not covered here.

Creating a database from scratch is not covered here.

The following two chapters cover **reading from** or **writing to** the database.

.