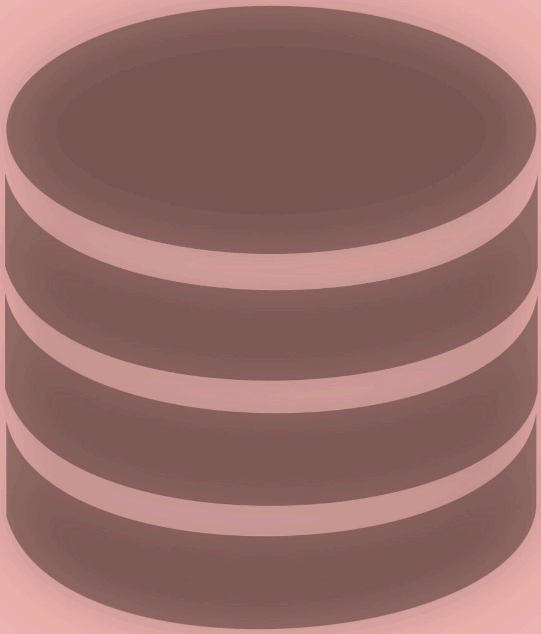
SQLAlchemy 2
In Practice



Learn to program relational databases in Python step by step

Miguel Grinberg

SQLAlchemy 2 In Practice

Miguel Grinberg

Jan 17, 2026

Contents

Preface

It was almost eleven years ago in 2012 that I started to write the Flask Mega-Tutorial. Initially published as a collection of blog articles over a period of a year, and later revised into an e-book and a video course, this work allowed an uncountable number of developers to take their first steps into the fascinating world of web development. If I'd ask you to guess which chapter in this tutorial is the most visited on my blog, you'd likely guess correctly that it is Chapter One. But could you guess which chapter is the second-most popular?

Interestingly, the second most-read chapter of the Flask Mega-Tutorial isn't Chapter Two. It is Chapter Four, which includes an introduction to databases and the SQLAlchemy library. If you browse through the hundreds of questions that people have left on my blog for this chapter, you'll realize that there is a common theme. Many of the questions are from developers that need to know how to do something a little more complex than what I present in the article, but are unable to figure out how to do it on their own. Over the years I have written standalone articles about some of the most asked database questions, but with such a vast topic, database questions have never stopped coming.

The book you are now reading is my attempt to address the large database knowledge gap that many developers have. I'm going to follow a similar approach to that of the Flask Mega-Tutorial, in that I will take you on a journey in which you will develop a real-world project that starts small and then evolves and gets more complex as you make progress through the chapters. The difference with the Flask Mega-Tutorial is that in this book the goal is not to create a complete web application but to build a flexible, efficient, real-world relational database, along with its associated processes and scripting, and with a focus on creating efficient queries and reports.

Because SQLAlchemy is an extensive framework with lots of options and ways of doing things, you should not expect this book to teach you every feature and every available trick. While I have made an effort to cover a wide variety of use cases, you should also become familiar with the SQLAlchemy official documentation, which is the ultimate reference from where you can pick up more ideas and techniques. My hope is that after working through this book you will be better prepared to navigate the official documentation and continue your learning process.

In this book you will learn how to work with relational databases in a way that is generic and apt to be applied to any web application framework, or to other types of applications not related to the web. You will even be able to incorporate what you learn in this book into modern asynchronous applications based on the `asyncio` package, such as those you can build with the FastAPI framework.

What You Will Build

The project that you will work on with the help of this book has the goal of supporting many of the aspects of the operation of a fictional vintage home computer store that I'm going to call RetroFun.

RetroFun offers an impressive collection of home computers from the 1980s and 1990s for sale. In this book you will learn how to build some database operations for this made-up company, including:

- Maintaining a list of products categorized by several attributes such as year of release, manufacturer, country of origin or CPU.
- Keeping track of customers and their orders.
- Maintaining star ratings and reviews made by customers.
- Tracking page views for articles published on the company's blog.
- Generating a lot of reports, both simple and complex, always keeping an eye towards efficiency and performance.

Prerequisites

You should be aware that this isn't a book for the complete beginner. To make the most out of it you should have some previous experience writing Python, and ideally also some basic relational database knowledge. If you have learned to work with databases with my Flask Mega-Tutorial[1], or with any other introductory Python course, you should be at the right level.

I recommend that you don't just read this book, but also work on all the exercises along with me. For this you will need a recent Python interpreter installed on your computer, and a text editor or IDE in which you feel comfortable writing Python code. Basic knowledge of the terminal or command prompt in your operating system would also help.

How To Work With The Example Code

You are encouraged to write and try all the code examples as you read this book. The code has been thoroughly tested using SQLAlchemy 2.0 on the three major open-source databases:

- SQLite[2]
- MySQL[3]

[1] https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-iv-database
[2] https://www.sqlite.org/
[3] https://www.mysql.com/

- PostgreSQL[4]

Other databases are likely to work as well, as long as SQLAlchemy has support for them.

The code from this book should work with SQLAlchemy 1.4 with relatively minor changes, but the effort to back port the book examples is left to the reader. Support for SQLAlchemy versions 1.3 and older was not taken into consideration.

I have released the complete source code for this book on a GitHub repository[5]. In addition to source code, this repository contains data files that can be used to populate the database you will build. At the appropriate times you will be given instructions on how to use these data files.

Conventions Used In This Book

This book frequently includes commands that you need to type in a terminal session. For these commands, a $ will be shown as a command prompt. This is a standard prompt for many Linux shells, but may look unfamiliar to Microsoft Windows users. For example:

```
$ python hello.py
hello
```

In a lot of the terminal examples, you are going to be required to have an activated *virtual environment* (do not worry if you don't know or remember what this is, you will find out very soon!). For those examples, the prompt will appear as `(venv) $`:

```
(venv) $ pip install sqlalchemy
```

You will also need to interact with the Python REPL, or interactive prompt. Examples that show statements that need to be entered in a Python interpreter session will use a >>> prompt, as in the following example:

```
>>> print('hello!')
hello
```

In all cases, lines that are not prefixed with a >>> prompt are output printed by the command right above, and should not be typed.

Many of the statements that you will need to type in the REPL are database queries that are formed by a call to SQLAlchemy's `select()` function followed by an often long sequence of method calls. Here is an example of how these queries might look when typed in a single long line:

```
>>> q = select(select_expression).method1(expression1).method2(expression2)
```

[4] https://www.postgresql.org/
[5] https://github.com/miguelgrinberg/retrofun

These statements can be quite unreadable when shown as above, so the convention used in this book is to show them broken up into multiple lines as follows:

```
>>> q = (select(select_expression)
        .method1(expression1)
        .method2(expression2))
```

Note that to be able to break long lines on the periods as shown above, Python requires the entire expression to be enclosed in parentheses. You can collapse these multi-line statements into single-line when you type them in the REPL.

Acknowledgements

This book was inspired by many of the questions I have received over the years from readers of my Flask Mega-Tutorial, so I'm extremely thankful to them for engaging with me and sharing their problems and ideas. To them, I also owe the realization that Python database programming is an area that hasn't been well explored in technical literature or video content.

Writing a technical book is hard, especially when the book is structured as a tutorial with detailed steps that the reader is expected to follow. While I have put a lot of care and attention in creating this content so that readers can have a smooth experience as they move through the chapters, I relied on reviewers to alert me of mistakes and inconsistencies that I inadvertently introduced. I would like to recognize the work of Martin Bell, Rostislav Roznoshchik and my son Dylan Grinberg as technical reviewers.

Finally, I would like to thank Mike Bayer and Federico Caselli. Mike is the creator of SQLAlchemy and Alembic. He and Federico are the current maintainers, and both have been extremely helpful and patient with my questions. Their assistance gave me a greater understanding of the major changes that have been introduced in releases 1.4 and 2.0 of SQLAlchemy. Mike was also kind enough to review the draft of this book and has made some useful suggestions.

1. Database Setup

Welcome! This is the start of a journey which I hope will provide you with many new tricks to improve how you work with relational databases in your Python applications. Given that this is a hands-on book, this first chapter is dedicated to help you set up your system with a database, so that you can run all the examples and exercises.

1.1. Project Directory

Your first task is to create a project directory where you will store files associated with the project featured in this book.

Open a terminal or command prompt, find a suitable parent directory and create a directory there. Then change into that directory.

```
$ mkdir retrofun
$ cd retrofun
```

Note: "RetroFun" is the name of the fictional company for which the database project featured in this book is for.

1.2. Python and SQLAlchemy Installation

As is standard in Python projects, you should create a Python virtual environment[1] where all the dependencies can be installed. The command to do this is:

```
$ python -m venv venv
```

[1] https://docs.python.org/3/library/venv.html

The `python -m venv` portion is what tells Python to create the virtual environment, by running the `venv` module that is part of the Python standard library. The second and final `venv` included in the command is the name I have chosen for the virtual environment. You are welcome to use a different name if you prefer.

After this command completes, your project will have a subdirectory named *venv*, containing a private copy of your Python interpreter.

Whenever you are ready to start working on this project, you have to tell your terminal session that you want to use the virtual environment. This action is called "activating" the virtual environment.

If you are using a UNIX based shell such as `bash`, regardless of operating system, the activation command is:

```
$ source venv/bin/activate
```

If you are using a Command Prompt on Microsoft Windows, the activation command is different:

```
$ venv\Scripts\activate
```

Finally, if you are using a PowerShell terminal the following is the activation command:

```
$ venv\Scripts\activate.ps1
```

Regardless of the activation command that you use, your shell prompt should change to indicate that the virtual environment has been activated. The prompt should look more or less as follows:

```
(venv) $ _
```

Note: Virtual environment activations only affect the shell session in which they are issued. If you have multiple terminals open, the activation command must be given for each terminal session. Activations must be issued again after a computer reboot or restart.

You can now install SQLAlchemy[2] in the virtual environment:

```
(venv) $ pip install sqlalchemy
```

Version 2.0 or newer of SQLAlchemy is required for the code featured in this book.

[2] https://www.sqlalchemy.org/

1.3. Database Choices

The code featured in this book is generic enough to be used with any relational database system supported by SQLAlchemy. The code examples have been tested against three popular open-source databases:

- SQLite[3]
- MySQL[4]
- PostgreSQL[5]

If you are interested in a particular database system, and it is not in the list above, then you should ensure that SQLAlchemy supports it[6], either through a built-in or a third-party integration.

If, on the other side, you have no particular preference, then my recommendation is that you start with SQLite, which is by far the easiest to set up and manage. Since the code uses common features present in all the database systems, you can switch to a different database when and if needed.

Have you made a decision? Now it is time to create a brand-new database for use with this book. If you have your preferred set of tools to do this you are welcome to use them, but in case you need some guidance, the sections that follow offer step-by-step instructions for the three databases listed above.

1.4. SQLite Database Installation

SQLite is a C-language library that implements a small, fast and self-contained relational database engine. This database is particularly interesting because it does not require a separate server process to run.

The SQLite library is bundled with the Python interpreter, so support for this database is available to use in Python and SQLAlchemy without installing any additional software or performing any configuration.

If you would like to have a tool that you can use to inspect and manage SQLite databases outside of Python and SQLAlchemy, you can download the `sqlite3` command-line shell for your operating system from the official SQLite download page[7].

[3] https://www.sqlite.org/
[4] https://www.mysql.com/
[5] https://www.postgresql.org/
[6] https://docs.sqlalchemy.org/en/latest/dialects/index.html
[7] https://www.sqlite.org/download.html

1.5. MySQL Database Set Up

MySQL is an open-source relational database owned by Oracle Corporation. Unlike SQLite, this database includes server and client components, both of which need to be installed.

1.5.1 MySQL Server

If you have access to a running MySQL server, then you can just create a new database and database user and skip to the client section below. The installation instructions in this section demonstrate how to install a server along with the popular phpMyAdmin[8] management application.

If you don't already have access to a MySQL installation, the easiest way to get one up and running is to use Docker. If you would like to follow the instructions below to install MySQL, first install Docker Desktop[9].

Copy the following definitions to a file named *docker-compose.yml* in your project directory:

```yaml
version: '3'

services:
  db:
    image: mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: changethis!
    ports:
      - "3306:3306"
    volumes:
      - db-data:/var/lib/mysql
  admin:
    image: phpmyadmin
    restart: always
    environment:
      - PMA_ARBITRARY=1
    ports:
      - 8080:80
volumes:
  db-data:
```

This Docker Compose configuration file starts a service called `db` that runs a MySQL server connected to port 3306 of your computer, plus a second service called `admin` that runs phpMyAdmin on port 8080. The database storage is configured on a separate volume called `db-data`, to make it possible to upgrade the database container without losing data.

Note the `MYSQL_ROOT_PASSWORD` line, which has the value `changethis!`. This line defines the administrator password for the MySQL server. Edit this line to set a secure password of your liking.

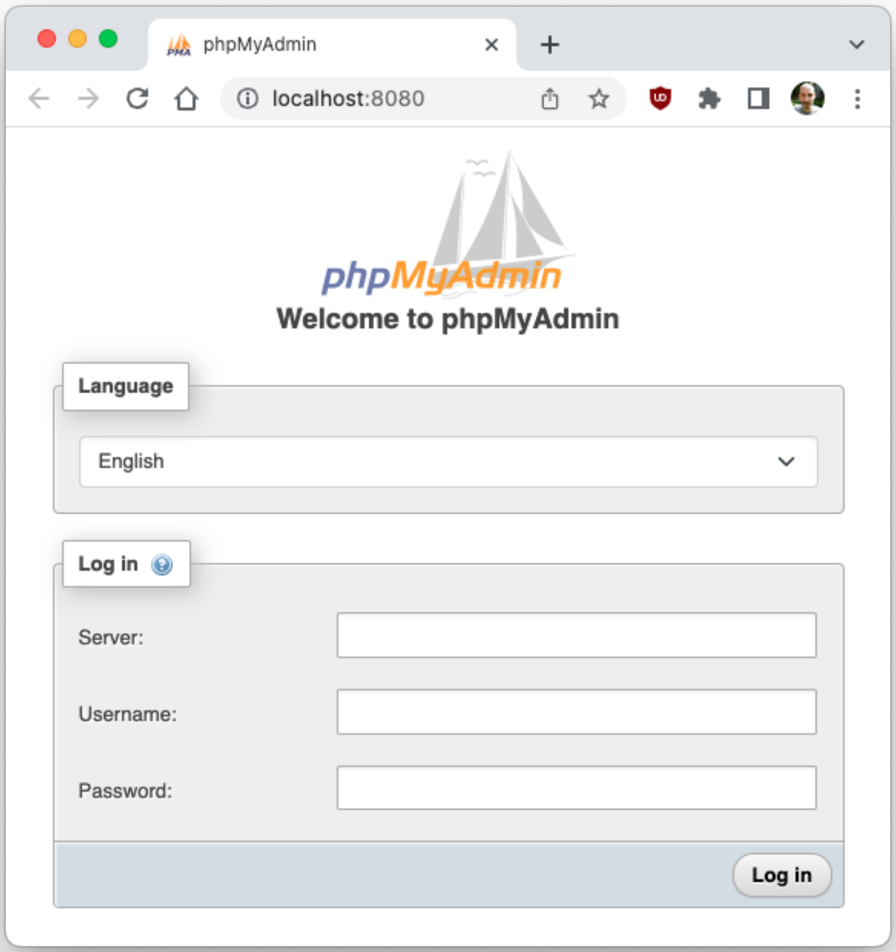
[8] https://www.phpmyadmin.net/
[9] https://www.docker.com/products/docker-desktop/

Once you have this file saved in your project directory, return to the terminal and run the following command to start your MySQL server:

```
$ docker-compose up -d
```

The first time you run this command it will take a while, as Docker has to download the MySQL and phpMyAdmin container images from the Docker Hub repository. Once the images are downloaded, it should take a few seconds for the containers to be launched, and at that point MySQL should be deployed on your computer and ready to be used.

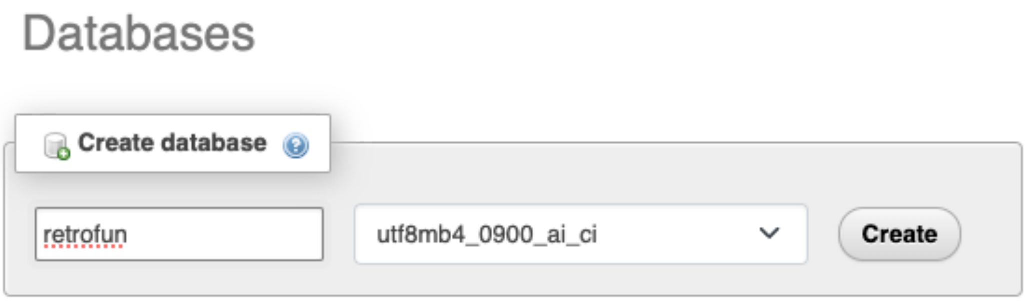
You can open the phpMyAdmin database management tool by typing *http://localhost:8080* in the address bar of your web browser.



To log in, enter the following credentials:

- Server: **db**
- Username: **root**
- Password: the root password that you entered in the *docker-compose.yml* file

Once you access the phpMyAdmin interface, click on the "Databases" tab. Near the top you should see the "Create Database" section.



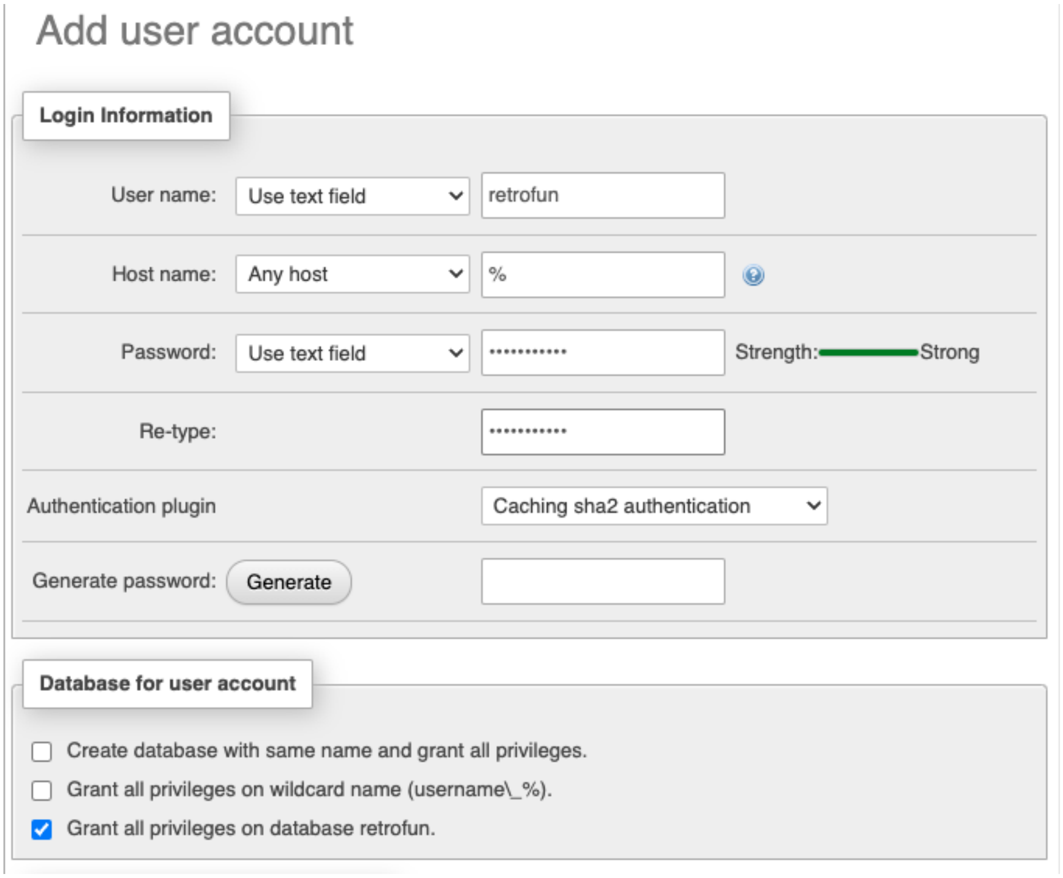
retrofun

utf8mb4_0900_ai_ci

Enter a name for the new database, such as **retrofun** and click the "Create" button.

A good practice when creating a new database is to also define a user specifically assigned to it. Using the root database user for day-to-day operations is too risky, because this account is too powerful and should only be used for important administration tasks.

Click on the "Privileges" tab for the new database. Near the bottom of the page there is a section titled "New" with an "Add user account" link. Click it to create a new user.



Add user account

Login Information

User name:	Use text field	retrofun	
Host name:	Any host	%	
Password:	Use text field	•••••••••••	Strength: Strong
Re-type:		•••••••••••	
Authentication plugin		Caching sha2 authentication	
Generate password:	Generate		

Database for user account

- ☐ Create database with same name and grant all privileges.
- ☐ Grant all privileges on wildcard name (username_%).
- ☑ Grant all privileges on database retrofun.

For the username you can choose any name that you like, but a naming convention that I find useful is to use the same name for the database and the user, so in this case it would be **retrofun**. Leave the host set to "%", then enter a password for the new user.

Confirm that the "Grant all privileges on database retrofun" option is enabled, and then scroll all the way to the bottom of the page and click the "Go" button to create the user. This user will have full access to the database, but it will not be able to access or create other databases, which is a good security principle to follow.

From now on, you can log in to phpMyAdmin using the user you just created, and your view of the database server will be constrained to only what's relevant to manage this particular database.

If you'd like to stop the MySQL server, you can do so with this command, issued from the directory in which you have your *docker-compose.yml* file:

```
$ docker-compose down
```

To start the server again, repeat the "up" command as before:

```
$ docker-compose up -d
```

Stopping and restarting the server as shown above does not cause any data loss.

1.5.2 MySQL Client

To access your MySQL database you have to install a Python client, sometimes also called driver. There are several MySQL drivers for Python[10] that can be used here, so as before, you should use your favorite if you have one.

If you need a recommendation, my driver of choice is pymysql[11], which you can install into your Python virtual environment as follows:

```
(venv) $ pip install pymysql cryptography
```

The `cryptography` package installed above is an optional dependency of `pymysql` that is needed to perform authentication against the MySQL database.

Congratulations! You now have a complete set up, including a blank MySQL database that is ready to be used. If you followed the installation procedure described above, the connection settings for your database are:

- Hostname: `localhost` (but use `db` as hostname to connect from the phpMyAdmin container)
- Port: 3306
- Database: `retrofun`
- Username: `retrofun`
- Password: the password that you selected for the user
- Python driver: `pymysql`

1.6. PostgreSQL Database Set Up

PostgreSQL (often shortened to Postgres) is yet another major open-source relational database system, similar to MySQL in the sense that it also requires separate server and client.

[10] https://docs.sqlalchemy.org/en/latest/dialects/mysql.html#dialect-mysql
[11] https://github.com/PyMySQL/PyMySQL/

1.6.1 PostgreSQL Server

If you have access to a PostgreSQL server then you can create a database and user to use with this book and skip to the next section to set up your client.

This section provides instructions to install PostgreSQL in your computer, along with the pgAdmin[12] administration application. These instructions are based on Docker containers, and are compatible with the three major operating systems. The easiest way to install Docker and Docker Compose on your computer is through Docker Desktop[13].

Copy the following Docker Compose configuration file to a file named *docker-compose.yml* in your project directory:

```
version: '3'

services:
  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: changethis!
    ports:
      - "5432:5432"
    volumes:
      - db-data:/var/lib/postgresql/data
  admin:
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: changethis!
    ports:
      - 8080:80
    volumes:
      - admin-data:/var/lib/pgadmin
volumes:
  db-data:
  admin-data:
```

This configuration file defines a service called `db` with the PostgreSQL server running on port 5432 of your computer, and a second service called `admin` that runs pgAdmin on port 8080. Both services require storage, so two volumes are also created for them. Using separate volumes for storage allows the data to persist if the containers are stopped and restarted.

There are three lines in the above configuration that need to be reviewed and edited:

- Change `POSTGRES_PASSWORD` to the PostgreSQL administrator password of your liking.
- Change `PGADMIN_DEFAULT_EMAIL` to your own email address (used only to log in).

[12] https://www.pgadmin.org/
[13] https://www.docker.com/products/docker-desktop/

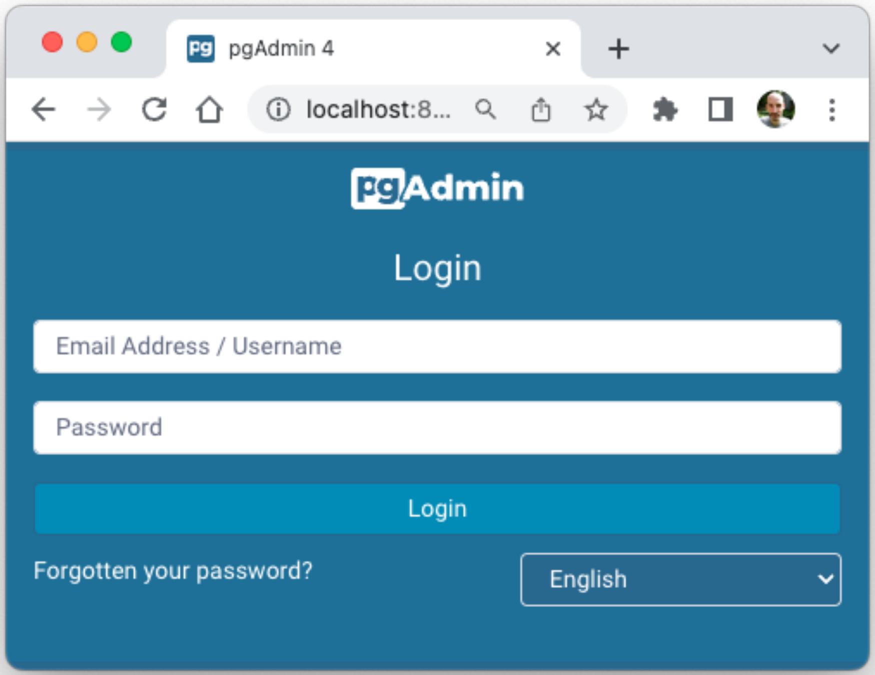
- Change PGADMIN_DEFAULT_PASSWORD to the pgAdmin administrator password of your liking.

Once you have the *docker-compose.yml* file ready, you can start the services with the following command:

```
$ docker-compose up -d
```

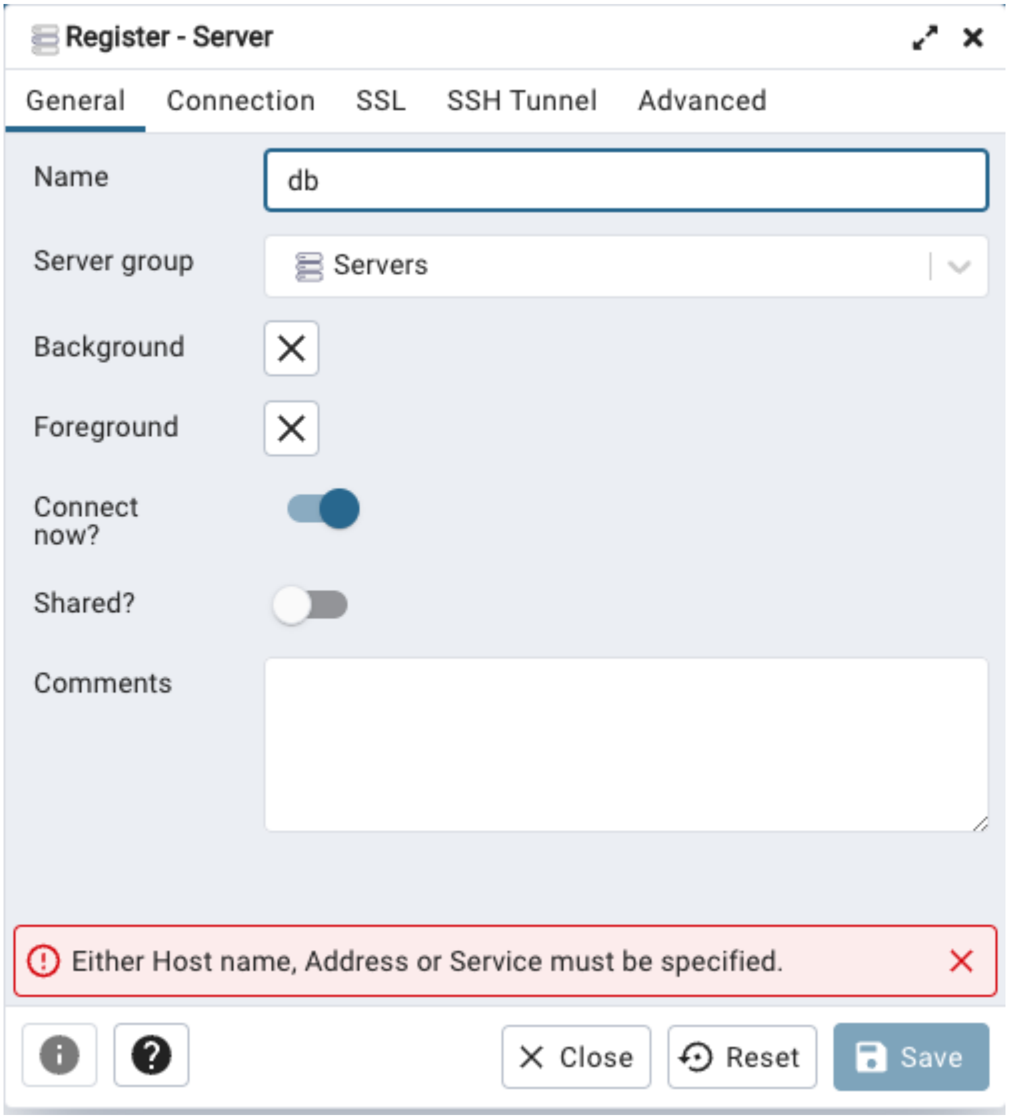
The first time you run this command Docker will have to download the Docker images for PostgreSQL and pgAdmin, so that may take a while. Once these images are downloaded, starting the services should take just a few seconds.

After the above command completes, give your computer a minute or two to get everything started and then connect to pgAdmin by typing *http://localhost:8080* on the address bar of your web browser.



You can log in to the administration interface with the email and password that you selected for the PGADMIN_DEFAULT_EMAIL and PGADMIN_DEFAULT_PASSWORD settings in your *docker-compose.yml* configuration file.

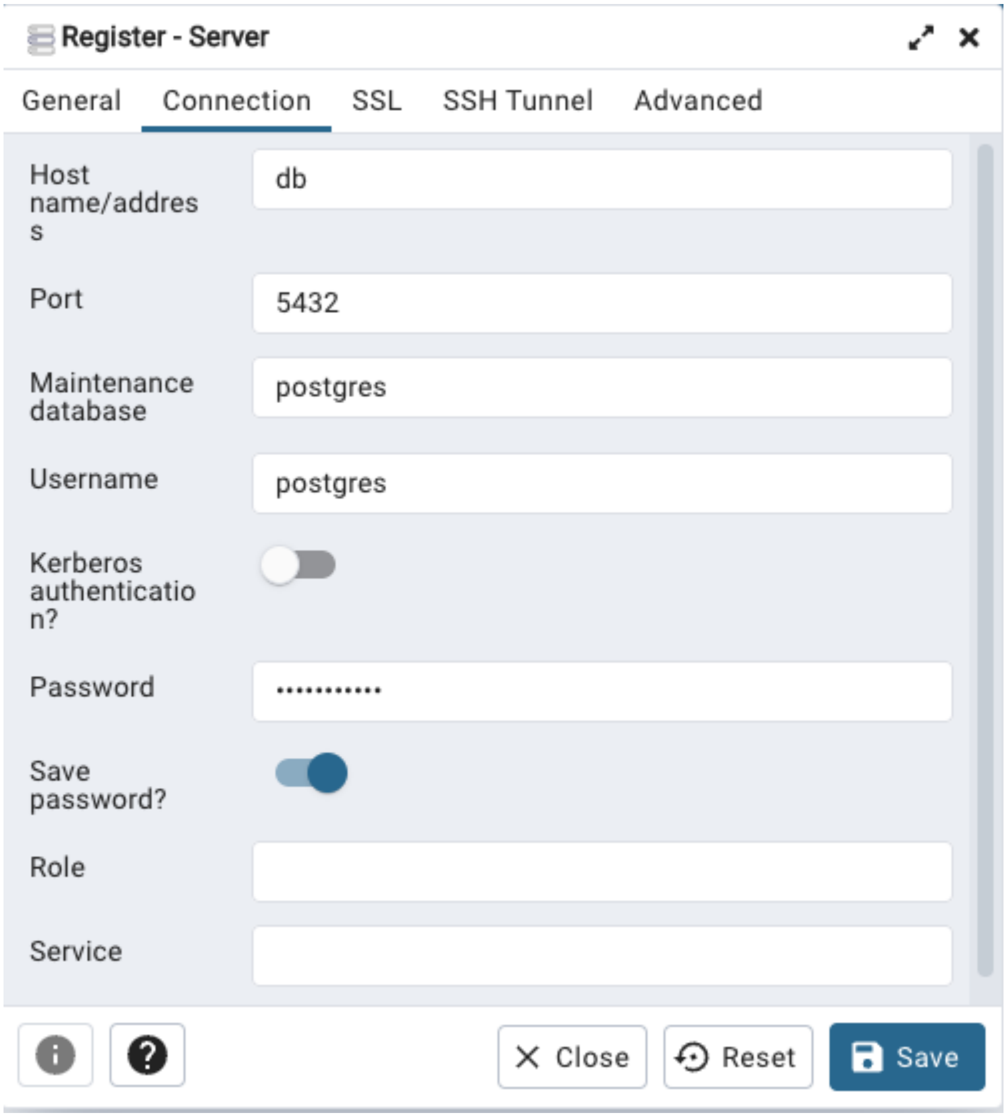
The first task is to tell pgAdmin about the PostgreSQL server. Click the "Add New Server" icon to do this. In the "General" tab, enter a name for the server such as **db** in the "Name" field.



Name: db
Server group: Servers
Background
Foreground
Connect now?
Shared?
Comments

Either Host name, Address or Service must be specified.

Then in the "Connection" tab, set "Host name" to **db**, which is the name of the PostgreSQL service as defined in the Docker Compose configuration. Leave the "Port" and "Maintenance Database" settings with their default values. Change "Username" to **postgres**, and write the password that you entered for the `POSTGRES_PASSWORD` setting in the "Password" field. You can enable the "Save password?" option if you don't want to have to re-enter the password in the future.


Register - Server

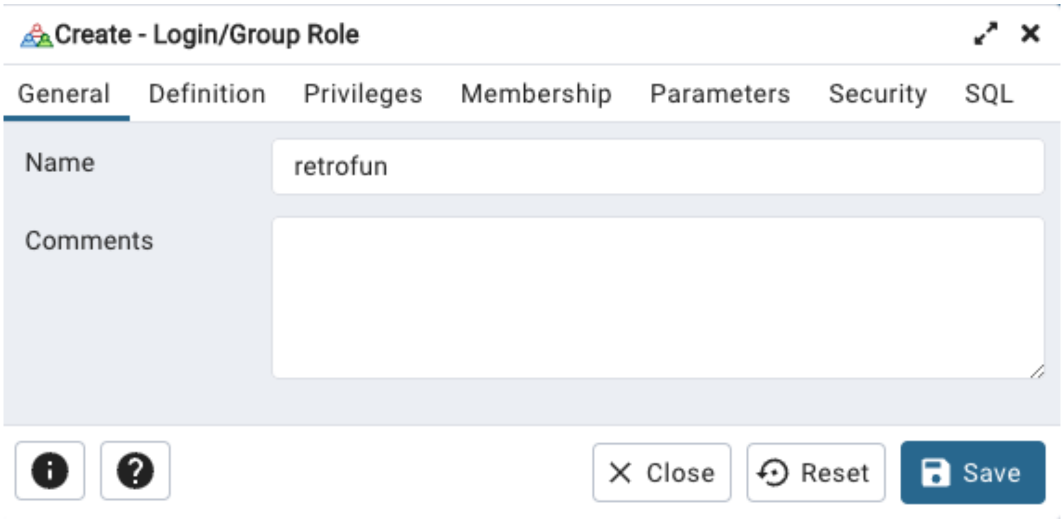
General | Connection | SSL | SSH Tunnel | Advanced

Host name/address	db
Port	5432
Maintenance database	postgres
Username	postgres
Kerberos authentication?	off
Password	•••••••••••
Save password?	on
Role	
Service	

After you click the "Save" button, pgAdmin will add the server to the left sidebar, and will start showing you live statistics about its operation.

The next step is to create a brand-new database that you can use to run the examples in this book. As with MySQL, it is a good practice to create a dedicated user for each database. To create the user, right-click on the db name in the sidebar and select "Create", and then "Login/Group Role...".

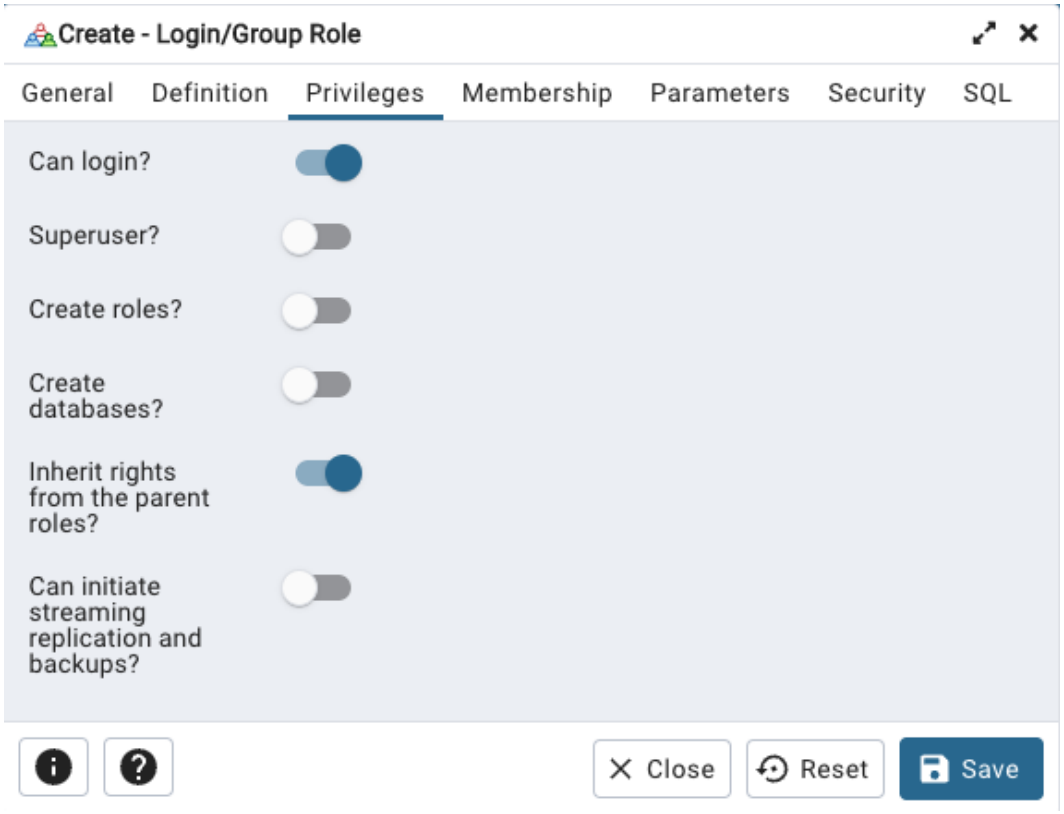
In the "General" tab, enter a name for the new user, such as **retrofun**.



Name: retrofun

Comments:

Switch to the "Definition" tab, and enter a password for the user. Then switch to the "Privileges" tab.



Can login? [✓ On]

Superuser? Off

Create roles? Off

Create databases? Off

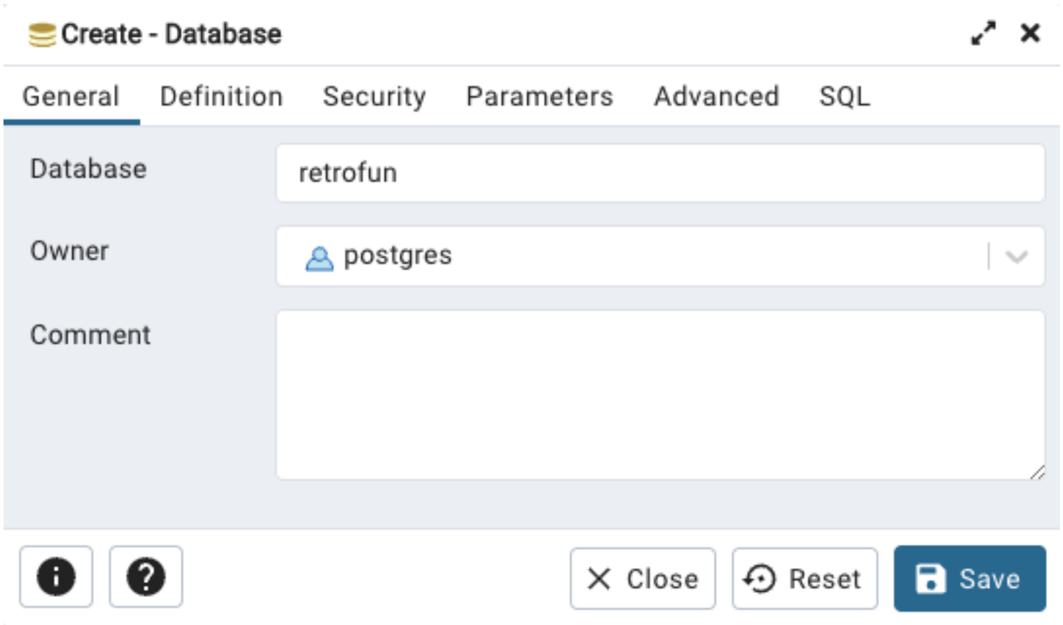
Inherit rights from the parent roles? [✓ On]

Can initiate streaming replication and backups? Off

This user should have the "Can login?" and "Inherit rights from the parent roles?" options enabled. To increase security it is best to have all other privileges disabled, as they are not needed.

Click the "Save" button to add the user.

Then right-click on the database on the left once again, then select "Create", and then pick "Database...".


Create - Database

General | Definition | Security | Parameters | Advanced | SQL

Database: retrofun

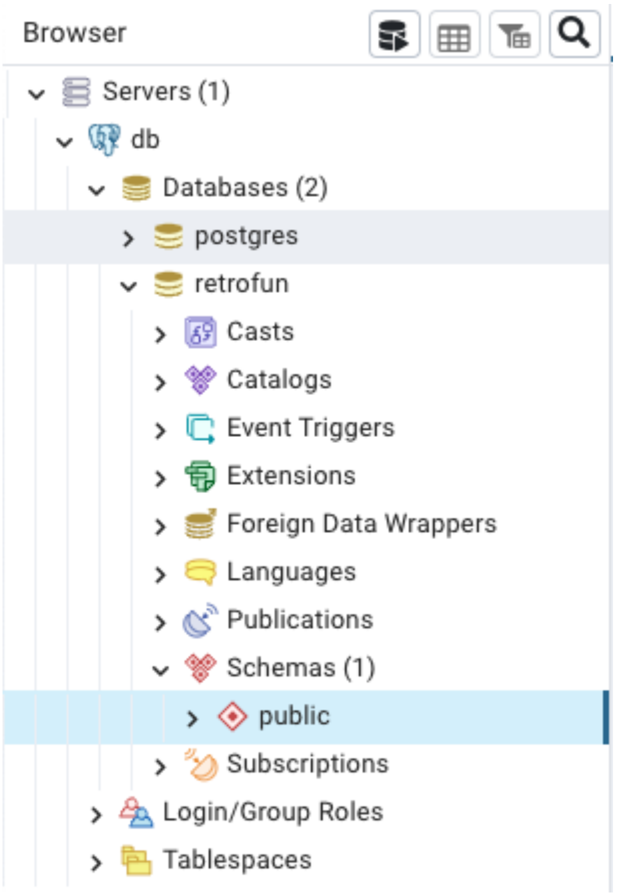
Owner: postgres

Comment:

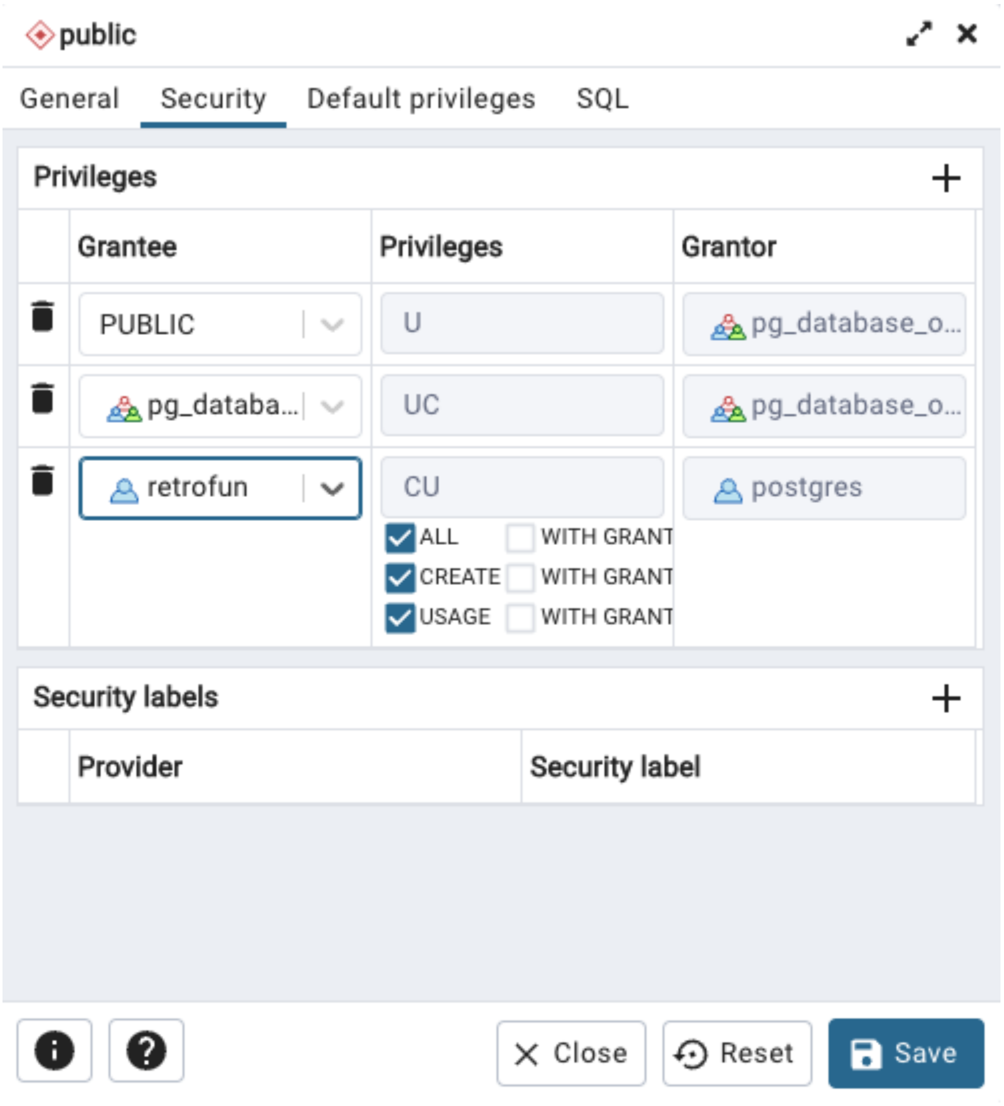
Give the new database a name, such as also **retrofun**. Naming the user and the database the same is a naming convention that I find convenient, since each user will be dedicated to only one database. The owner of the database should be the `postgres` user, which is the administrator.

Click "Save" to create your new database.

The next step is to configure the privileges of the `retrofun` user so that it can have full access to the new database. In the left sidebar, expand the tree view starting from the db server and continuing on to "Databases", the `retrofun` database, "Schemas", and finally "public".



Right-click on the `public` schema and select "Properties...". Then select the "Security" tab.



Privileges

Grantee	Privileges	Grantor
PUBLIC	U	pg_database_o...
pg_databa...	UC	pg_database_o...
retrofun	CU	postgres

[✓] ALL [ ] WITH GRANT
[✓] CREATE [ ] WITH GRANT
[✓] USAGE [ ] WITH GRANT

Security labels

Provider	Security label

Click the "+" in the "Privileges" table to add a new entry. Under "Grantee", select the `retrofun` user. In the "Privileges" column check the "ALL" option to give the user full access to the schema.

Click "Save" to store the new privileges.

To stop the PostgreSQL server you can issue the following command from the directory in which you have your *docker-compose.yml* file:

```
$ docker-compose down
```

To start the server again, repeat the "up" command as before:

```
$ docker-compose up -d
```

Thanks to the data being stored in standalone volumes, you can freely stop and restart the server without losing any data.

1.6.2 PostgreSQL Client

The final step is to install a PostgreSQL driver for Python. SQLAlchemy supports a few PostgreSQL drivers[14], and you can choose any of them if you have a preference.

A driver that is extremely popular and has proven to be very stable is psycopg2[15], which you can install with this command:

```
(venv) $ pip install psycopg2-binary
```

To connect to your database from Python you will later need to know the connection details. If you followed the instructions above, then these are:

- Hostname: `localhost` (but use db as hostname to connect from the pgAdmin container)
- Port: 5432
- Database: `retrofun`
- Username: `retrofun`
- Password: the password that you selected for the user
- Python driver: `psycopg2`

1.7. Database Connection URLs

When using SQLAlchemy, a database to connect to is represented by a URL that has the following structure:

```
{dialect}{+driver}://{username}:{password}@{hostname}:{port}/{database}
```

[14] https://docs.sqlalchemy.org/en/latest/dialects/postgresql.html#dialect-postgresql
[15] https://www.psycopg.org/

URLs for MySQL and PostgreSQL are built using `mysql` or `postgresql` as dialect respectively, plus the connection details for your database.

The following examples assume that the user password is `my-password`:

```
# MySQL with pymysql
url = 'mysql+pymysql://retrofun:my-password@localhost:3306/retrofun'

# PostgreSQL with psycopg2
url = 'postgresql+psycopg2://retrofun:my-password@localhost:5432/retrofun'
```

Database URLs for SQLite are a bit different, because this is an in-process database without the concept of users or servers. For this database, the dialect name is `sqlite` and the driver can be omitted. The username, password, hostname and port are also omitted, since they do not have any meaning for this database. Finally, instead of a database name, a path to the database file is given.

The following examples show some possible database URLs for a SQLite database named `retrofun.sqlite`:

```
# database file in the current directory
url = 'sqlite:///retrofun.sqlite'

# database file in /home/miguel/retrofun directory
url = 'sqlite:////home/miguel/retrofun/retrofun.sqlite'

# database file in C:\users\miguel\retrofun directory (Microsoft Windows)
url = 'sqlite:///c:\\users\\miguel\\retrofun\\retrofun.sqlite'
```

If you look at these URLs carefully, you may think that they have too many forward slashes right after the `sqlite:` prefix, but these are all correct.

The first example uses a relative location (the current directory) for the database file. In this URL, the first two forward slashes are part of the `sqlite://` URL prefix, and the third slash is the one that comes after the username, password, hostname and port, only in this case these four are empty so only the slash separator needs to be included.

In the second example there are four forward slashes after the dialect and driver. The first three slashes have the same purpose as in the first example. The fourth slash is the start of an absolute path for the SQLite database file, which in this example is */home/miguel/retrofun/retrofun.sqlite*.

The third and final example shows how an absolute path can be given when using the Microsoft Windows operating system. Here what follows the three forward slashes is an absolute path that starts with a disk drive and uses backslashes as path component separators. Python strings need the backslash character to be escaped by entering a second backslash.

The SQLite database provides one additional option: an in-memory database. This is useful for temporary databases, such as those used in unit tests. With an in-memory database, all the data is kept in the memory of the process, without persistence. This option is not going to be used in this book, but it is useful to be aware of it.

```
url = 'sqlite://'
```

In all the examples in this book, the database URL will be configured externally, in an environment variable named `DATABASE_URL`. To avoid having to set this variable in every shell session, create a file named *.env* (a dot followed by *env*, often called a "dotenv" file), open it in your text editor, and write the database URL that you would like to use in it as follows:

```
DATABASE_URL=sqlite:///retrofun.sqlite
```

The above example configures a SQLite database named *retrofun.sqlite* in the current directory.

The python-dotenv[16] package allows an application to read variables from a *.env* file. Install it with `pip`:

```
(venv) $ pip install python-dotenv
```

Below you can see an example of how to read the `DATABASE_URL` variable from a Python program. Copy this code to a file named *db.py* in the project directory to try it out on your computer.

Listing 1.1: *db.py*: Display the database URL

```python
import os
from dotenv import load_dotenv

load_dotenv()

print('Database URL:', os.environ['DATABASE_URL'])
```

Run this example to ensure that you have configured your database URL correctly:

```
(venv) $ python db.py
Database URL: sqlite:///retrofun.sqlite
```

[16] https://pypi.org/project/python-dotenv/

2. Database Tables

This chapter provides an overview of the most basic usage of the SQLAlchemy[1] library to create, update and query database tables.

2.1. SQLAlchemy Core and SQLAlchemy ORM

The SQLAlchemy library is divided into two modules called Core and ORM (short for Object-Relational Mapping).

The Core module contains the database integration logic for all the supported database dialects, a collection of classes to describe database tables, and a fairly sophisticated system for generating SQL statements using Python language constructs.

The ORM module introduces a layer of abstraction between the Python application and the database that allows many database operations to be automatically derived from actions performed on Python objects.

An application can choose to use SQLAlchemy Core exclusively, or it can combine elements from Core and ORM. In this book you will learn to use a combined approach.

2.2. The Database Engine

SQLAlchemy uses "engine" objects to manage connections to a database, both for Core and ORM applications. The `create_engine()` function constructs an engine given a database URL. Below you can see an updated version of the *db.py* file you created in the previous chapter, showing how to create an engine object from a `DATABASE_URL` environment variable imported from the *.env* file:

[1] https://www.sqlalchemy.org/

Listing 2.1: *db.py*: Create a SQLAlchemy engine object

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine

load_dotenv()

engine = create_engine(os.environ['DATABASE_URL'])
```

The `create_engine()` function can be passed additional keyword arguments to configure the engine. Interesting options include:

- `echo=True`, to have SQLAlchemy log every SQL statement issued to the database. This is a very useful option when debugging.
- `pool_size=<N>`, to specify a custom size for the connection pool SQLAlchemy maintains (the default is a pool size of up to 5 simultaneous connections).
- `max_overflow=<N>`, the maximum number of connections above the pool size that can be created during usage spikes (the default is 10).
- `future=True`, to tell SQLAlchemy 1.4 to use the newer 2.0 based APIs.

Consult the documentation for create_engine()[2] to see the complete list of options that are available.

2.3. Models

When using the ORM module, database tables are defined in the application as Python classes. The application must create a parent class for all these classes, where settings that are common to all the tables can be configured. This parent class, which SQLAlchemy calls the *declarative base* class, is often named `Model`, or in some cases `Base`. The collection of subclasses of the `Model` class represent the structure or schema of the database, and are generally referred to as the "models" of the application.

The `Model` class must inherit from SQLAlchemy's `DeclarativeBase` class. Here is an updated version of *db.py* that defines `Model` as an empty class, without any custom settings:

Listing 2.2: *db.py*: Create a declarative base class

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine
from sqlalchemy.orm import DeclarativeBase

```

(continues on next page)

[2] https://docs.sqlalchemy.org/en/latest/core/engines.html#sqlalchemy.create_engine

(continued from previous page)

```
class Model(DeclarativeBase):
    pass


load_dotenv()

engine = create_engine(os.environ['DATABASE_URL'])
```

To help keep things nicely organized, the models for the application you are going to build with the help of this book are going to be stored in their own file, which will be called *models.py* file. The next code example shows a first implementation of a model for a `products` database table:

Listing 2.3: *models.py*: Product model class

```
from sqlalchemy import String
from sqlalchemy.orm import Mapped, mapped_column
from db import Model


class Product(Model):
    __tablename__ = 'products'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(64))
    manufacturer: Mapped[str] = mapped_column(String(64))
    year: Mapped[int]
    country: Mapped[str] = mapped_column(String(32))
    cpu: Mapped[str] = mapped_column(String(32))

    def __repr__(self):
        return f'Product({self.id}, "{self.name}")'
```

If you have used older versions of SQLAlchemy, you may find the above model definition to be significantly different. In version 2.0, SQLAlchemy introduced an integration with Python type hints, and that is the reason why the currently recommended syntax for column definition has changed from older releases.

As indicated above, all application model classes must inherit from the `Model` declarative base class, which is imported from *db.py*.

Model subclasses are configured using class attributes. The `__tablename__` attribute defines the name of the database table the class represents. A very common naming convention for database tables is to use the plural form of the entity in lowercase, so in this case the table is given the `products` name. This contrasts with the convention used for the model class names, which prefers the singular form in camel case.

The remaining attributes defined in the class represent the columns of the table. The `Mapped[t]` type declaration is used to define each column, with `t` being the Python type assigned to the column, such as `int`, `str`, or `datetime`. For simple columns such as `year` above, this is all that is necessary. If the column needs to be given additional options, it is assigned to a `mapped_column()` constructor that provides those options.

In the `Product` model defined above, an option is used to identify the `id` column as a primary key, which means that the values in this column must uniquely identify each item stored in the table. Without any additional configuration, SQLAlchemy configures integer primary key columns with auto-incrementing numbers starting from 1. You will later learn other ways to define primary keys.

The remaining columns describe the attributes that products have. For columns that are of type `str`, a maximum length is added with a supplementary `String()` option. Not all databases require a length to be given for string columns, but it is best to always include a length just in case.

The `__repr__()` method included in this class is a special method that tells Python how an object of this class should be printed. Adding this method is optional, but it is useful as an aid when debugging or when trying things out in a Python shell, which is something you will often while working with this book.

To create an instance of a model class, a standard class constructor is used, passing the values for the model's attributes as keyword arguments. For example:

```
c64 = Product(name='Commodore 64', manufacturer='Commodore')
```

The above example initializes a new `Product` instance. SQLAlchemy provides a default constructor for all model classes that accepts value assignments for its columns. In this particular example, all the attributes of `c64` except `name` and `manufacturer` will be set to `None`, because they were not assigned a value when the object was created. Even though this object is a model instance, at this point it is just a plain Python object that is not stored or associated with any database.

Note: The concept of model classes is available only for applications that use the ORM module. When using Core, instances of the `Table` class are used to represent database tables.

2.4. Database Metadata

SQLAlchemy maintains the definitions of all the tables that make up a database in an object of class `MetaData`. For convenience, it initializes the declarative base class with a `metadata` attribute that has a default `MetaData` object. For the `Model` class, the metadata instance is available as `Model.metadata`. When a model class such as `Product` is defined, SQLAlchemy creates a corresponding table definition in this attribute.

The default `MetaData` configuration has one important limitation that is bound to cause problems when projects reach certain size or level of complexity. This is related to the `naming_convention` option, which tells SQLAlchemy how to name indexes and constraints it creates on a database. You will learn what these are later in this chapter, but for now, just consider that in the same way as tables, indexes and constraints need to have a name.

The default naming convention used in the `MetaData` object provides a naming rule for indexes, but not for constraints, so SQLAlchemy initializes all constraints without an explicit name, which results in them having arbitrary names chosen by the database. This is a problem if at some point a constraint needs to be modified or deleted, since SQLAlchemy wouldn't immediately know how to address the constraint by its name. To avoid this potential complication down the road, the `Model` declarative base can be initialized with a more complete set of naming conventions, as shown below:

Listing 2.4: *db.py*: Configure naming conventions for indexes and constraints

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import DeclarativeBase


class Model(DeclarativeBase):
    metadata = MetaData(naming_convention={
        "ix": "ix_%(column_0_label)s",
        "uq": "uq_%(table_name)s_%(column_0_name)s",
        "ck": "ck_%(table_name)s_%(constraint_name)s",
        "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
        "pk": "pk_%(table_name)s",
    })


load_dotenv()

engine = create_engine(os.environ['DATABASE_URL'])
```

The `MetaData` object has a `create_all()` method that has significant importance, as it creates all the database tables associated with the defined models. Here is an example usage:

```
Model.metadata.create_all(engine)
```

Here the `create_all()` method will issue SQL statements to the database represented by `engine` to create the database tables referenced by all the models. Following the code examples from previous sections, this call would create a `products` table, which is defined by the `Product` model.

An important limitation of `create_all()` is that it only creates tables that don't already exist in the database, which means that when a model class is changed, this method cannot be used to transfer the change to the corresponding database table.

A workaround that can be used to modify an existing table is to remove the old and outdated version of the table from the database before calling `create_all()` again. As a convenience, the `MetaData` object also has a `drop_all()` method, which removes all the tables from the database. The following example refreshes all the tables to their latest definitions:

```
Model.metadata.drop_all(engine)
Model.metadata.create_all(engine)
```

Unfortunately updating a database in this way is only practical for small tests or while prototyping, because `drop_all()` not only deletes the tables but also all the data stored in them. You will later learn how to use Alembic[3] to manage updates to the database in a much more effective way through *migration scripts*.

Note: When using Core, the database metadata object must be manually created by the application.

2.5. Sessions

Another important entity in ORM-based applications is the *session*. A session object maintains the list of new, read, modified and deleted model instances.

Changes that accumulate in a session are passed on to the database in the context of a database transaction when the session is *flushed*, which is an operation that in most cases is automatically issued by SQLAlchemy when it is needed. A flush operation writes the changes to the database, but keeps the database transaction open.

When the session is *committed*, the corresponding database transaction is committed as well, causing all the changes to be permanently written to the database.

Database transactions are one of the most important benefits of relational databases, designed to maintain the integrity of the data. The changes that are committed as part of a transaction are written as an atomic operation, so errors or unexpected interruptions will never result in partial or incomplete data being written. If an error or failure occurs while a session is active, a *rollback* operation on the session will roll the transaction back, and all the changes made up until that point in that session will be undone.

The following example shows how the `c64` object created in the previous section can be added to a database session and committed:

```
from sqlalchemy.orm import Session

with Session(engine) as session:
    try:
        session.add(c64)
        session.commit()
    except:
        session.rollback()
```

(continues on next page)

[3] https://alembic.sqlalchemy.org/en/latest/