yactouat
Posted on 31 de ago. de 2022

2

flask + postgres + sqlalchemy migrations dockerized intro

#python #postgres #docker #beginners

practical Python (2 Part Series)

1 Connect to a Postgres SQL database with a Python Flask app' o…
2 **flask + postgres + sqlalchemy migrations dockerized intro**

When developing a new Python Flask web API or adding new functionalities to an existing one, it's really useful to be able to generate migrations on the fly, it's even better if we can do that in a dockerized environment !

In this post, we are going to do the following:

1. explain what database migrations are
2. set up a Python Flask project using a postgres db in Docker
3. create the first migrations of the project

So let's get to it !

prerequisites

- a basic knowledge of PostgreSQL
- a basic knowledge of Python
- a basic understanding of Docker
- having a working standard Docker install on your development machine

what migrations are

A database of an organization being the reflect of its business logic in the real world, this business logic may evolve; therefore, these changes will eventually be reflected in the database. For instance, let's say that a cie x has implemented an orders full for consumers who want to buy its products; in the future, this cie may lease some of its expensive products... This is a change in the business logic that would certainly be reflected in the database schema.

Data migrations are the process of making the existing data schema evolve. These migrations should allow for a quick rollback in case mistakes are made; in other words, migrations act as a version control system for your database.

As data migrations encapsulate successive versions of a database; they are typically stored as local files in the project's repository, so the versioning of your schemas is also versioned in git 😎.
That way, any developer working on the project can read the project's business logic history and better understand the context of the app' she/he is working on.

Database migrations provide better maintainability of the whole domain logic of an application, but for that to work well, there should be a 1 to 1 mapping between the changes made to your database, and the migration files that exist in your migrations/ folder.

Data migrations should be uniquely named and sequentially ordered; also, their naming should indicate what action has been done on the database schema (it is a common practive in frameworks such as Laravel, but not much in Flask).
Generally, database migrations are created/applied/rollbacked using command line scripts created for that task. Luckily, there are many open-source packages for various languages/frameworks that help us do that. In our case, we are going to use Flask-Migrate since we are going to use Flask, a micro framework with which we will get a classical todo web app' up and running.

set up a Python Flask project using a postgres db in Docker

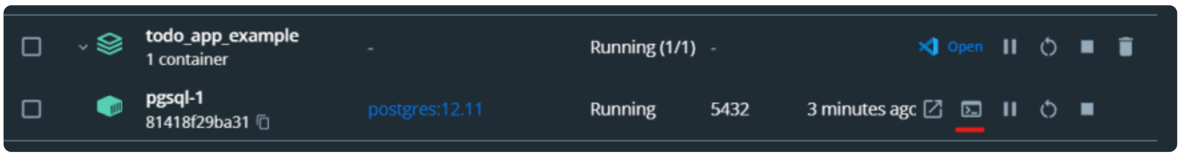
Ok, let's get this web app' running.

We are going to set the foundation of our application stack with a database: create a new project in your favorite IDE, with a `docker-compose.yml` file at root level of this project =>

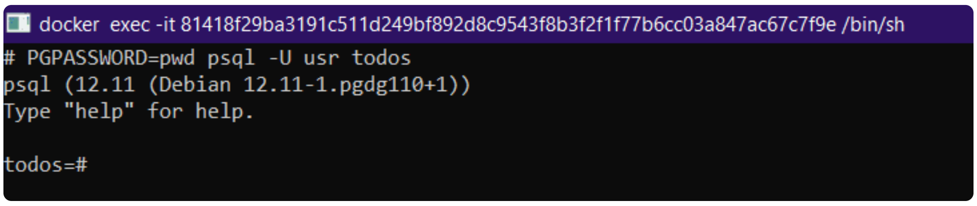
```yaml
# creating a volume to be able to persist data between Postgres container restarts
volumes:
  todos-vol:

services:

  pgsql:
    image: postgres:12.11
    restart: always
    environment:
      POSTGRES_PASSWORD: pwd # environment variable that sets the superuser password for PostgreSQL
      POSTGRES_USER: usr # variable that will create the specified user
      POSTGRES_DB: todos # the name of your db
    volumes:
      - todos-vol:/var/lib/postgresql/data
    ports:
      - 5432:5432
```

Now, when running `docker compose up` from the root of the project, you should be able to access your Postgres instance from within the container using `psql`.

I personally like to use the Docker GUI to access my containers because it's easy and fast :)



Once inside the container, you can access your instance shell directly wired to your db by running =>



```
docker exec -it 81418f29ba3191c511d249bf892d8c9543f8b3f2f1f77b6cc03a847ac67c7f9e /bin/sh
# PGPASSWORD=pwd psql -U usr todos
psql (12.11 (Debian 12.11-1.pgdg110+1))
Type "help" for help.

todos=#
```

Good ! we now have a working instance of PostgresSQL, we can get on with the second main moving part of our application stack: the Python Flask app'.

First let's reference our Python app' in our application stack inside the `./docker-compose.yml` file =>

```yaml
# creating a volume to be able to persist data between Postgres container restarts
volumes:
  todos-vol:

services:

  pgsql:
    image: postgres:12.11
    restart: always
    environment:
      POSTGRES_PASSWORD: pwd # environment variable that sets the superuser password for PostgreSQL
      POSTGRES_USER: usr # variable that will create the specified user with superuser power and a database with the sa
      POSTGRES_DB: todos
    volumes:
      - test-vol:/var/lib/postgresql/data
    ports:
      - 5432:5432

  python:
    # we are not going to use the Python image as is but rather tweak one to our needs
    build:
      context: .
      dockerfile: ./docker/Dockerfile
    depends_on:
      - pgsql
    # using port 80 for convenience so we can access localhost directly without specifying the port
    ports:
      - 80:5000
    # the Flask app' code will go into the `app` folder of your project and be mapped to `/usr/src/app` in the contain
    volumes:
      - ./app:/usr/src/app
```

Here, we are referencing 2 things that don't exist yet in our project:

- an `app` folder
- a `./docker/python.Dockerfile`

Create both `app` and `docker` folders at the root of your project.

Since we are going to use Flask and Flask-Migrate libs in our todos app', we need a `requirements.txt` file, at the root of our project, that we will use in our Dockerfile to tell our dockerized environment which dependencies need to be installed beforehand to make our app' runnable =>

```
# ./requirements.txt
psycopg2-binary==2.9.3
Flask==2.1.2
Flask-SQLAlchemy==2.5.1
Flask-Migrate==3.1.0
```

- `psycopg2-binary` is a package to install if you need to quickly whip up a PostgreSQL adapter without having to worry too much about the environment of the server; e.g. you'll be able to talk to a PostgreSQL DB in Python without having to install any other lib in C in your system (suitable for dev)
- `Flask` contains the core of our Flask web micro framework
- `Flask-SQLAlchemy` is a popular implementation of the famous `SQLAlchemy` (https://docs.sqlalchemy.org/en/14/) ORM in Flask; modern ORMs (**object relational mapper***) consist of an elaborated piece of software that allows you to translate your code in actionable queries against a wide array of database systems, they usually also provide optimizations of various kinds (caching, lazy loading, etc.) that make your interaction with the db smoother
- `Flask-Migrate` (https://github.com/miguelgrinberg/flask-migrate) is the package that will help us do our migrations

Let's write our starter Python code, we want a minimal app' in the `app` folder so we can put into practice executing migrations =>

```
# ./app/routes.py
from flask import jsonify

def init_routes(app):

    @app.route("/api", methods=["GET"])
    def get_api_base_url():
        return jsonify({
            "msg": "todos api is up",
            "success": True,
            "data": None
        }), 200
```

Here, we just define a function to set up a base route for our API (basically see if it works when going to `/api` endpoint).

```
# ./app/init.py
from flask import Flask, jsonify

from routes import init_routes


def create_app(test_config=None):

    # creates an application that is named after the name of the file
    app = Flask(__name__)

    app.config["SECRET_KEY"] = "some_dev_key"
    app.config["SQLALCHEMY_DATABASE_URI"] = "postgresql://usr:pwd@pgsql:5432/todos"

    # initializing routes
    init_routes(app)

    return app
```

This is a very simple implementation of an app' factory, that contains the initialization logic of our Flask app'.
It is to be used in the entry point of our app', that we will simply call `app.py`

```
# ./app/app.py
from init import create_app
app = create_app()

if __name__ == "__main__":
    app.run(host="0.0.0.0")
```

That's it for the boilerplate code !

Now, there is only one piece missing before having a functioning Flask app' in Docker: the Dockerfile =>

```
# ./docker/Dockerfile
FROM python:3.9.13

# specifying the working directory inside the container
WORKDIR /usr/src/app

# installing the Python dependencies
COPY ./requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# copying the contents of our app' inside the container
COPY ./app .

# defining env vars
ENV FLASK_APP=app.py
# watch app' files
ENV FLASK_DEBUG=true
ENV FLASK_ENV=development

# running Flask as a module, we sleep a little here to make sure that the DB is fully instanciated before running our a
CMD ["sh", "-c", "sleep 5 \
    && python -m flask run --host=0.0.0.0"]
```

Now, if you run a `docker compose up` from the root of the project in your terminal, you should be able to visit `localhost/api` in your browser and see the first output of your Flask web API.

create the first migrations of the project

Now let's get to the meat of our migrations: models. Models are the representation in code of the actual entities that will populate your database; in the context of our RDBMS, they will represent the different tables that you are going to create inside PostgreSQL.

For now, let's keep it simple and assume that there will be only one entity in our app': the `Todo` entity; later on, you can put all your models within the same `models.py` file =>

```
# ./app/models.py
from flask_sqlalchemy import SQLAlchemy

# session_options={"expire_on_commit": False} =>
# would allow to manipulate out of date models
# after a transaction has been committed
# ! be aware that the above can have unintended side effects
db = SQLAlchemy()


class Todo(db.Model):
    __tablename__ = "todos"

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    completed = db.Column(db.Boolean, nullable=False, default=False)
    description = db.Column(db.String(), nullable=False)
    due_date = db.Column(db.DateTime, nullable=True)

    def __repr__(self):
        return f"<Todo {self.id}, {self.completed}, {self.description}>"
```

In here, we do 2 things:

- we instanciate our SQLAlchemy main object (`db`)
- we define what a `Todo` should look like

But to be able to reflect this entity in our database, we need to make a little (but capital) change in our `./app/app.py` entry point file =>

```
# ./app/app.py
from init import create_app
app = create_app()
# bootstrap database migrate commands
db.init_app(app)
migrate = Migrate(app, db)

if __name__ == "__main__":
    app.run(host="0.0.0.0")
```

What we did here is:

- wiring our app' to the ORM with `db.init_app(app)`
- wire our migrations, which are the link between the code and the actual database schema, to our app with `migrate = Migrate(app, db)`

What does that mean ? It means that now, `Flask-Migrate` will be able, among other things, to:

- auto detect changes between versions of SQLAlchemy models (say, if you add a new model or change what a `Todo` consists of, for instance)
- create scripts that resolve the differences between models versions

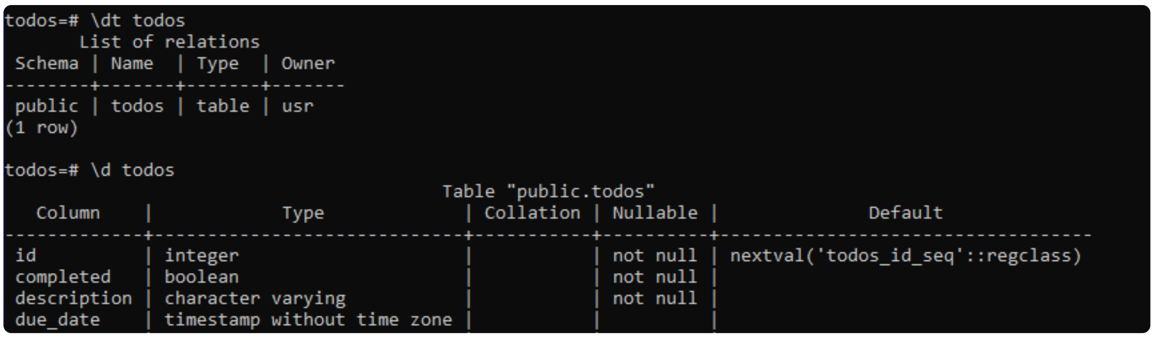
However, migrations still need to be manually run to be applied: this is what gives you control of what changes happen to your database.

So, now that everything is wired up, we need to talk about the migrations flow:

- you write (creating or updating) models that represent the business logic entities of your application
- you need to set up the migrations folder **once** with one simple command; go ahead try to run `flask db init` from your Python container after having killed the previously running application stack and re run `docker compose up --build --force-recreate` => this will create a `migrations` folder inside your `app` folder

- now, when you'll run `flask db migrate`, changes in your SQLAlchemy models will be detected and corresponding migrations will be created in the `./app/migrations/version` folder
- finally, running `flask db upgrade` will actually implement the changes in the DB

This is what you should see now in your PostgreSQL container =>



```
todos=# \dt todos
       List of relations
 Schema | Name  | Type  | Owner
--------+-------+-------+-------
 public | todos | table | usr
(1 row)

todos=# \d todos
                                         Table "public.todos"
   Column    |            Type             | Collation | Nullable |              Default
-------------+-----------------------------+-----------+----------+-----------------------------------
 id          | integer                     |           | not null | nextval('todos_id_seq'::regclass)
 completed   | boolean                     |           | not null |
 description | character varying           |           | not null |
 due_date    | timestamp without time zone |           |          |
```

Now you're all set up in implementing a migrations workflow in your Flask app', enjoy !

You can find a repo that implements all the concepts that have been tackled in this article @ https://github.com/yactouat/todo_app_example

Please don't hesitate to give your feedback in the comments section below.

👋

practical Python (2 Part Series)

1 Connect to a Postgres SQL database with a Python Flask app' o…

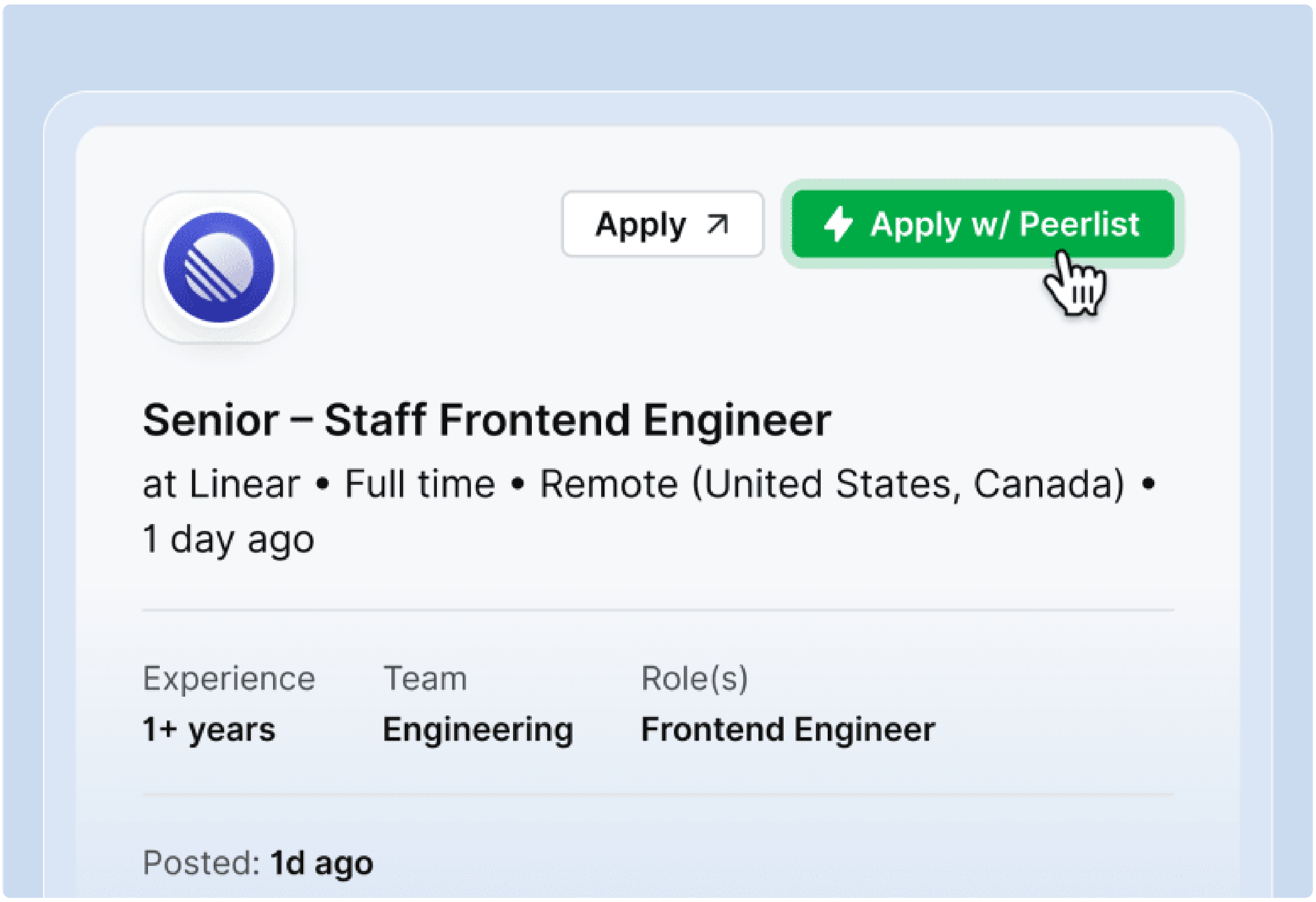
2 **flask + postgres + sqlalchemy migrations dockerized intro**

Top comments (0)

Code of Conduct • Report abuse

Peerlist PROMOTED

...


Apply
Apply w/ Peerlist
Senior – Staff Frontend Engineer
at Linear • Full time • Remote (United States, Canada) • 1 day ago
Experience
1+ years
Team
Engineering
Role(s)
Frontend Engineer
Posted: 1d ago

yactouat

I'm a generalist web developer who is driven by curiosity, positivity, and a can-do attitude; I like to design full stack solutions with various technologies.

WORK
web developer / web developement mentor

JOINED
6 de dez. de 2021

More from yactouat

MongoDB Node.js Connection
#node #mongodb #beginners

a gentle introduction to client-side data fetching and rendering
#beginners #webdev #javascript #php

Setting up dockerized cron jobs: example using PHP and MySQL
#docker #cronjobs #beginners

DEV Community ···