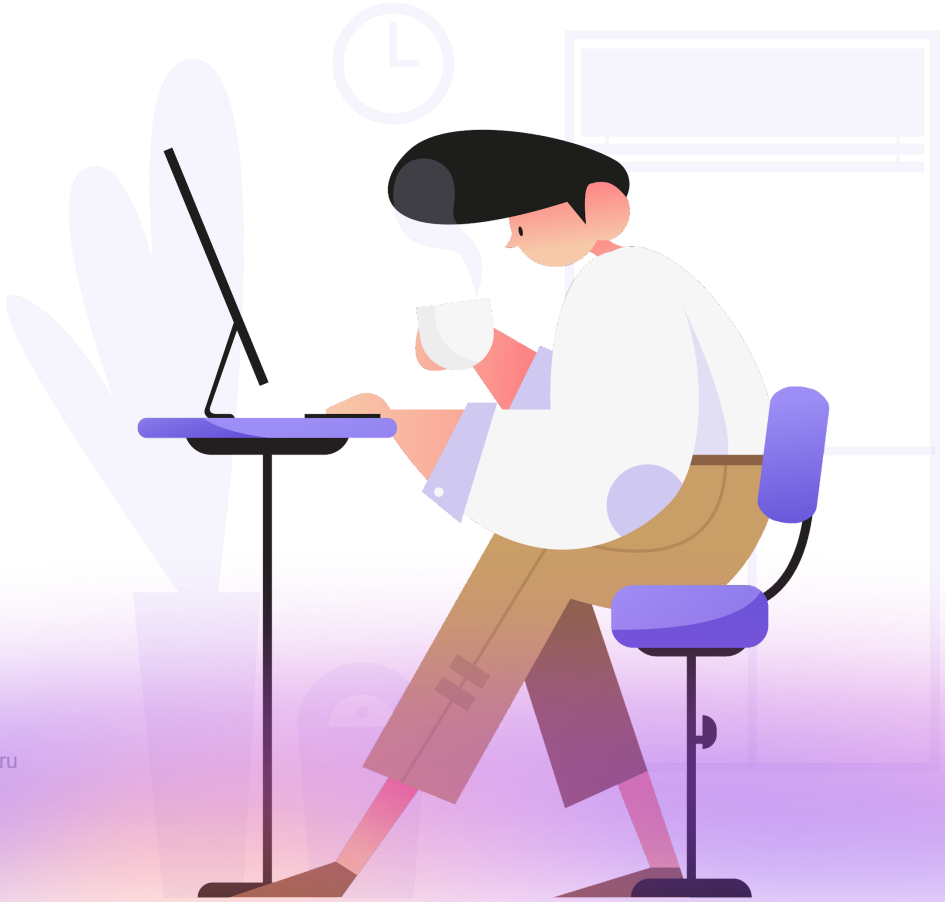
GeekBrains

Flask

Docker, docker-compose, Postgres. Миграции схем Flask-Migrate и alembic



На этом уроке

1. Создадим модуль configs и используем class-based подход для конфигурации сервера.
2. Завернём проект в докер, научимся использовать docker-compose.
3. Изучим процессы миграции в Flask (migrate, upgrade, downgrade).

Оглавление

Теория

Docker

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему с поддержкой cgroups в ядре, а также предоставляет среду по управлению контейнерами.

Dockerfile — файлы, которые сообщают Docker о том, как собирать образы, на основе которых создаются контейнеры. Каждому образу Docker соответствует файл, который называется Dockerfile. Его имя записывается именно так — без расширения. При запуске команды docker build для создания нового образа подразумевается, что Dockerfile находится в текущей рабочей директории.

PostgreSQL, или **Postgres** — свободная объектно-реляционная система управления базами данных. Существует в реализациях для множества UNIX-подобных платформ, включая AIX, различные BSD-системы, HP-UX, IRIX, Linux, macOS, Solaris/OpenSolaris, Tru64, QNX, а также для Microsoft Windows.

Docker network — возможность подключения контейнера к сети/сетям.

Alembic — инструмент миграции баз данных.

Миграции — это правила обновления структуры базы, которые привязаны к версиям ПО. Утилита придумана для автоматизированного обновления и отката обновлений структуры БД на серверах. Alembic позволяет разработчикам через миграции обновлять базу данных при помощи ORM, не используя SQL

Flask-Migrate

Позволяет настроить миграции для ORM SQLAlchemy. Для использования миграций необходимо подключить команду, произвести начальную инициализацию, выполнив flask db init, затем использовать действия migrate, upgrade и downgrade данной команды для управления миграциями. Список действий для команды и их краткое описание можно получить, выполнив flask db help:

- flask db init
- flask db migrate
- flask db upgrade
- flask db downgrade
- flask db help

Практическое задание

Переносим конфигурации в классы

Это нужно для объектно-ориентированного подхода к конфигурациям: таким образом мы сможем переопределять нужные параметры и при этом не дублировать повторяющиеся.

Создаём blog/configs.py

```
import os


class BaseConfig(object):
    DEBUG = False
    TESTING = False
    SQLALCHEMY_DATABASE_URI = "sqlite:///:memory:"
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = "abcdefg123456"


class DevConfig(BaseConfig):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = os.environ.get("SQLALCHEMY_DATABASE_URI")


class TestingConfig(BaseConfig):
    TESTING = True
```

В blog/app.py используем нужный конфиг. Параметры берём из переменных окружения (для среды разработки устанавливаем переменную окружения CONFIG_NAME = DevConfig)

```
import os

cfg_name = os.environ.get("CONFIG_NAME") or "ProductionConfig"
app.config.from_object(f"blog.configs.{cfg_name}")
```

Собираем Docker контейнер

Dockerfile:

```
FROM python:3.9.1-buster

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt

COPY wsgi.py wsgi.py
COPY blog ./blog

EXPOSE 5000

CMD ["python", "wsgi.py"]
```

docker-compose.yaml

```
version: '3.5'

services:
  app:
    build:
      dockerfile: ./Dockerfile
      context: .
    environment:
      SQLALCHEMY_DATABASE_URI: postgresql://user:password@pg:5432/blog
      CONFIG_NAME: DevConfig
    volumes:
    - ./blog:/app/blog
    ports:
      - 5000:5000
    depends_on:
      - pg

  pg:
    image: postgres:12
    environment:
      POSTGRES_DB: blog
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      PGDATA: /var/lib/postgresql/data/pgdata
    volumes:
      - ${HOME}/dbs/flask-lesson/pgdata_dev:/var/lib/postgresql/data/pgdata
    ports:
      - 5432:5432
```

Обращаем внимание на wsgi.py: нам необходимо указать host="0.0.0.0", потому что мы будем обращаться к приложению из внешней сети.

```
    app.run(
        host="0.0.0.0",
        debug=True,
    )
```

Теперь можно собрать и запустить приложение следующими командами:

- `docker-compose build app`
- `docker-compose up app` (или `docker-compose up -d app`, чтобы запустить приложение в фоне)

Устанавливаем Flask-Migrate, автоматически подтянется alembic

```
pip install Flask-Migrate
```

И в blog/app.py подключаем к основному приложению

```
from flask_migrate import Migrate

migrate = Migrate(app, db)
```

Создание первой миграции

Выполняем flask db migrate -m “create user model”, будет создана автоматическая миграция на создание пользователя. Выполняем её при помощи команды flask db upgrade. Можем убедиться, что в базе данных создана новая таблица для хранения пользователей.

Продолжаем знакомиться с миграциями. Добавляем поле в модели: blog/models/user.py

```
email = Column(String(255), nullable=False, default="", server_default="")
```

И проводим миграцию снова, выполняя

```
flask db migrate -m "add email field to user model"
flask db upgrade
```

Чтобы Flask-Migrate замечал разницу при изменении свойств колонок — например, максимальная длина строчки, — необходимо указать `compare_type=True` в инициализатор:

```
migrate = Migrate(app, db, compare_type=True)
```

Итоги

Мы создали модуль `configs` и использовали class-based подход для конфигурации приложения.

Упаковали проект в докер, научились использовать команду `docker-compose`.

Изучили процессы миграции в Flask при помощи Flask-Migrate.

Практическое задание

1. Собрать докер-образ со своим проектом.
2. Подключить Postgres к своему Flask проекту.
3. Обновить модель пользователя.
4. Сгенерировать автоматическую миграцию на создание схемы пользователя.
5. Добавить миграцию данных для создания стандартного админа.

Дополнительные материалы

Знакомство с Docker (habr).

Docker network (habr).

Flask-Migrate.

alembic.