Шпаргалка по работе на сервере: Часть 2-я

Докер

Установка докера на компьютер

Windows: установите WSL по инструкции, после чего установите Docker Desktop.

macOS: установите Docker Desktop

Linux: установите Docker Engine, используя готовый скрипт от Docker:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh ./get-docker.sh
```

Управление докер-контейнерами

Запуск контейнера

```
docker run --rm -p порт_хоста:порт_контейнера --name имя_контейнера имя_образа
```

- run: команда для запуска контейнера
- --rm: указывает, что контейнер надо удалить после остановки (опционально)
- -p порт_хоста:порт_контейнера: перенаправляет запросы с указанного порта хоста на указанный порт контейнера (опционально)
- --name имя_контейнера: присваивает контейнеру заданное имя (опционально). Если имя не задано — оно будет сгенерировано автоматически
- имя_образа: указывает, на основе какого образа запустить контейнер

Остановка контейнера и повторный запуск; удаление контейнера

```
docker container stop имя_контейнера
docker container start имя_контейнера
docker container rm имя_контейнера
```

Команды Dockerfile

```
# Создать образ на основе базового слоя,
# который содержит файлы ОС и интерпретатор Python 3.9.
FROM python:3.9

# Перейти в образе в директорию /app: в ней будет храниться код проекта.
# Если директории с таким именем нет, она будет создана.
# Название директории может быть любым.
WORKDIR /app
# Дальнейшие инструкции будут выполняться в директории /app
```

```
# Скопировать с локального компьютера файл зависимостей
# в текущую директорию образа (текущая директория — это /app).
COPY requirements.txt .

# Выполнить в текущей директории образа команду терминала
# для установки зависимостей.
RUN pip install -r requirements.txt --no-cache-dir

# Скопировать всё необходимое содержимое
# той директории локального компьютера, где сохранён Dockerfile,
# в текущую рабочую директорию образа — /app.
COPY . .

# При старте контейнера запустить сервер разработки.
CMD ["python", "manage.py", "runserver", "0:8000"]
```

.dockerignore

Чтобы случайно не скопировать в образ файлы, которые там не должны быть (например, файлы с переменными окружения или файлы библиотек), используется файл .dockerignore. Его нужно расположить рядом с Dockerfile. Синтаксис аналогичен синтаксису .gitignore.

Типичный файл .dockerignore для Django-проекта:

```
venv
.git
.env
db.sqlite3
```

Docker Volume

Docker volume позволяет хранить данные вне контейнера так, чтобы к ним был доступ из контейнера. При этом данные в volume будут сохранены и после удаления контейнера; доступ к одному volume может быть у нескольких контейнеров.

Команда для создание volume:

```
docker volume create имя_volume
```

Использование volume при запуске контейнера:

```
docker run -v имя_volume:путь_к_папке_в_контейнере имя_образа
```

Docker Compose

Синтаксис docker-compose.yml

Пример файла docker-compose.yml:

```
# Файл docker-compose.yml

# Версия docker-compose:
version: '3'

# Перечень volume:
volumes:
  pg_data:

# Перечень контейнеров:
services:
  # Имя и описание первого контейнера; имя выбирает разработчик.
  # Это контейнер с базой данных:
  db:
    # Из какого образа запустить контейнер:
    image: postgres:13.10
    # Файл (или список файлов) с переменными окружения:
    env_file: .env
    # Какой volume подключить для этого контейнера:
    volumes:
      - pg_data:/var/lib/postgresql/data
  # Имя и описание контейнера с бэкендом:
  backend:
    # Из какого Dockerfile собирать образ для этого контейнера:
    build: ./backend/
    env_file: .env
    # Какие контейнеры нужно запустить до старта этого контейнера:
    depends_on:
      - db
  # Имя третьего контейнера. Это контейнер с фронтендом:
  frontend:
    env_file: .env
    build: ./frontend/
```

version — версия спецификации файла docker-compose.yml. Обязательный параметр. От версии к версии набор доступных команд меняется, и какие-то команды из новых версий могут не поддерживаться старыми версиями Docker Compose. Узнать, какая версия Docker Compose установлена на компьютере, можно с помощью команды docker compose version.

В документации описано соответствие версий Docker Compose версиям файла docker-compose.yml. В Практикуме мы работаем с Docker Compose 1.10.

volumes — перечень volumes для докера, это необязательный параметр. Для каждого имени volume через двоеточие можно указать его подробные настройки. Их можно и не указывать — докер применит настройки по умолчанию.

services — названия и описания контейнеров, которые должны быть запущены. В листинге описаны три контейнера: db, backend и frontend.

Ключи в конфигурации можно указывать в любом порядке. В примере сначала указаны volumes, а потом services: при описании контейнеров удобнее видеть, какие volumes уже созданы.

Описание каждого контейнера — это YAML-словарь, значения в этом словаре похожи на параметры запуска, которые вы применяли при ручном старте контейнеров.

В описании контейнера объявляется:
1. image или build: <address> (одно из двух):
   - image — из какого образа создать и запустить контейнер;
   - build: <address> — создать образ из докерфайла, который лежит в директории <address>, и запустить контейнер из этого образа.

2. volumes — список подключаемых к контейнеру volumes (опциональный):

```
volumes:
  - имя_volume:директория_контейнера
```

Ещё вариант — просто указать директорию контейнера, для которой будет создан volume:

```
volumes:
  - директория_контейнера
```

Будет создан анонимный volume, у него не будет имени, его не нужно описывать в общем блоке volumes. Другие способы создания volumes можно посмотреть в документации.

3. env_file указывает один или несколько файлов с переменными окружения для контейнера (опционально).

4. depends_on — список контейнеров, которые должны быть запущены перед запуском описываемого контейнера (опционально). Значение ключа depends_on — список: иногда запускаемый контейнер зависит не от одного, а от нескольких контейнеров. В листинге указано, что контейнер backend должен быть запущен после контейнера db: при старте Django-приложения база данных должна быть уже доступна.

Дополнительно в depends_on можно указать состояние предыдущего контейнера, при котором можно запустить текущий контейнер, — это описано в документации.

Управление контейнерами с Docker Compose

Запуск всех описанных в docker-compose.yml контейнеров:

```
docker compose up
```

Запуск всех описанных в docker-compose.yml контейнеров в фоновом режиме:

```
docker compose up -d
```

Остановка всех контейнеров:

```
docker compose stop
```

Остановка и удаление всех контейнеров:

```
docker compose down
```

Остановка и удаление всех контейнеров и volume:

```
docker compose down -v
```

Команда для запуска новой команды в запущенном контейнере:

```
docker compose exec имя_контейнера команда
```

Если файл называется не docker-compose.yml, то в каждой команде после compose нужно указывать параметр -f имя_файла, например:

```
docker compose -f имя_файла up
```

GitHub Actions

Синтаксис. Пример workflow

```yaml
# Файл main.yml
# Имя workflow:
name: Main Workflow
# Перечень событий-триггеров, при которых должен запускаться workflow:
on:
  # Событие push возникает, когда изменения кода приходят на сервер GitHub.
  push:
    # Отслеживаем изменения только в ветке main:
    branches:
      - main
# Перечень задач:
jobs:
  # Единственная задача — склонировать код и вывести в консоль дерево файлов.
  checkout-and-print-tree:
    runs-on: ubuntu-latest
    steps:
      # Применим готовое описание шага для получения исходного кода:
      - name: Check out repository code  # Имя шага задаём сами.
        uses: actions/checkout@v3  # Готовое решение из библиотеки GitHub Actions.
      # Выполняем команду tree в текущей директории:
      - name: Print project tree
        run: tree .
```

- name — имя workflow, оно будет использоваться в интерфейсе GitHub Actions;
- on — события-триггеры, после которых должен срабатывать workflow; триггеров может быть несколько;
- jobs — список действий, которые должны выполниться после срабатывания триггера.

Каждая задача (job) описывается набором параметров:

```
...
jobs:
  checkout-and-print-tree:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository code
        uses: actions/checkout@v3
      - name: Print project tree
        run: tree .
```

Первый параметр, runs-on, определяет, в каком окружении будут запущены все команды этой задачи. Окружение создаётся сервисом GitHub Actions на его сервере.

В приведённом примере в runs-on указана ubuntu-latest: последняя версия Ubuntu.

Каждая отдельная задача делится на шаги — steps. Каждый шаг — отдельная команда. Перечень шагов форматируется в виде списка словарей; в начале каждого шага ставится -.

Шагам можно дать имя с помощью ключа name. Именовать шаги не обязательно, но при отладке процесса имя поможет понять, на каком шаге возникла проблема.

В ключе run хранится команда, она будет выполнена в терминале окружения на раннере.

Для подключения стороннего workflow вместо ключа run применяется ключ uses. Более подробное описание работы с uses есть в документации. В примере использован готовый workflow **actions/checkout@v3**: он клонирует текущий коммит репозитория, в котором запущен workflow в текущую рабочую директорию раннера. Таким образом, раннеру становится доступен исходный код проекта.

Базовый Workflow для тестов

```yaml
# .github/workflows/main.yml
name: Main workflow

on:
  push:
    branches:
      - main

jobs:
  tests:
    # Разворачиваем окружение:
    runs-on: ubuntu-latest

    steps:
    # Копируем код проекта:
    - name: Check out code
      uses: actions/checkout@v3
    # Устанавливаем Python с помощью action:
    - name: Set up Python
      uses: actions/setup-python@v4
    # В action setup-python@v4 передаём параметр — версию Python:
      with:
        python-version: 3.9
    # Обновляем pip, устанавливаем flake8 и flake8-isort,
    # устанавливаем зависимости проекта:
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8==6.0.0 flake8-isort==6.0.0
    # Запускаем flake8:
    - name: Test with flake8
      # Вызываем flake8 и указываем ему,
      # что нужно проверить файлы только в папке backend/:
      run: python -m flake8 backend/
```

Базовый Workflow для сборки образов

```
# .github/workflows/main.yml

# Тут задачи тестирования и сборки образа.
# ...
# Добавляем новую задачу: деплой приложения:
build_and_push_to_docker_hub:
    name: Push Docker image to DockerHub
    runs-on: ubuntu-latest
    needs: tests
    steps:
      - name: Check out the repo
        # Получение исходного кода из репозитория:
        uses: actions/checkout@v3
      - name: Set up Docker Buildx
        # Установка сборщика контейнеров Docker:
        uses: docker/setup-buildx-action@v2
      - name: Login to Docker
        # Авторизация на Docker Hub:
        uses: docker/login-action@v2
        # При помощи with передаём в action параметры username и password:
        with:
          username: <ваш_username_docker_hub>
          password: <ваш_пароль_docker_hub>
          # Хорошо ли держать логин и пароль прямо в коде workflow?
          # Нет, это нехорошо. Хранить пароль надо в Secrets.
      - name: Push to DockerHub
        # Одновременный билд и пуш образа в Docker Hub:
        uses: docker/build-push-action@v4
        with:
          # Параметр context: ./backend/ указывает, что нужный Dockerfile
          # находится в ./backend/
          context: ./backend/
          # Параметр push: true указывает, что образ нужно не только собрать,
          # но и отправить на Docker Hub:
          push: true
          # В параметре tags задаётся название и тег для образа.
          # Для каждого пересобранного образа
          # устанавливаем тег latest, чтобы потом
          # на сервере и в docker-compose.yml не указывать версию образа:
          tags: ваш-логин-на-docker-hub/имя_образа:latest
```

Базовый Workflow для выкладки на сервер

```
deploy:
    runs-on: ubuntu-latest
    needs:
      # Дождёмся билда контейнеров фронтенда, бэкенда и гейтвея:
      - build_and_push_to_docker_hub
      - build_frontend_and_push_to_docker_hub
      - build_gateway_and_push_to_docker_hub
    steps:
    - name: Checkout repo
      uses: actions/checkout@v3
    # Копируем docker-compose.production.yml на продакшен-сервер:
    - name: Copy docker-compose.yml via ssh
      uses: appleboy/scp-action@master
    # Передаём параметры для action appleboy/scp-action:
      with:
        host: ${{ secrets.HOST }}
        username: ${{ secrets.USER }}
        key: ${{ secrets.SSH_KEY }}
        passphrase: ${{ secrets.SSH_PASSPHRASE }}
        source: "docker-compose.production.yml"
        target: "имя_директории_на_сервере"
    - name: Executing remote ssh commands to deploy
      uses: appleboy/ssh-action@master
      with:
        host: ${{ secrets.HOST }}
        username: ${{ secrets.USER }}
        key: ${{ secrets.SSH_KEY }}
        passphrase: ${{ secrets.SSH_PASSPHRASE }}
        # Параметр script передаёт в action appleboy/ssh-action команды,
        # которые нужно выполнить на сервере,
        # с которым установлено соединение:
        script: |
          cd имя_директории_на_сервере
          # Выполняет pull образов с Docker Hub:
          sudo docker compose -f docker-compose.production.yml pull
          # Перезапускает все контейнеры в Docker Compose:
          sudo docker compose -f docker-compose.production.yml down
          sudo docker compose -f docker-compose.production.yml up -d
          # Выполняет миграции и сбор статики.
          # Все команды начинаются так:
          # Вместо ... — sudo docker compose -f docker-compose.production.yml.
          ... exec backend python manage.py migrate
          ... exec backend python manage.py collectstatic
          ... exec backend cp -r /app/collected_static/. /backend_static/static/
```