GeekBrains

Django REST Framework

Развёртывание проекта с Docker и Docker Compose

На этом уроке

1. Научимся разворачивать проект в нескольких Docker-контейнерах.
2. Научимся настраивать взаимодействие контейнеров между собой.
3. Узнаем различные варианты сборок проекта.
4. Научимся поднимать проект на «боевом» VPS-сервере.

Оглавление

Введение

Наш демонстрационный проект написан и протестирован. Он состоит из backend-части на Django и frontend-части на React. На этом занятии мы рассмотрим процесс развёртывания проекта на «боевом» сервере с использованием Docker и Docker Compose.

План развёртывания

Количество контейнеров для развёртывания проекта может быть разным. На практике обычно используют от трёх и более контейнеров. Важный момент — создание отдельного контейнера с frontend или сборка frontend вместе с backend.

Мы будем использовать следующие контейнеры:

- `db` — база данных postgres;
- `backend` — Django-проект, запущенный с помощью Gunicorn;
- `nginx` — веб-сервер для backend;
- `frontend` — веб-сервер для frontend.

Мы рассмотрим самый гибкий вариант развёртывания с созданием отдельного контейнера для frontend.

Мы будем делать промежуточные шаги для самопроверки и исправления возникающих ошибок. Когда вы попрактикуетесь работать с Docker, можно будет их пропустить и сразу создать четыре финальных контейнера.

1. Запустим базу данных и образ `adminer` для проверки её работоспособности.
2. Соединим базу данных с тестовым Django-сервером и сделаем миграции.
3. Запустим Django-проект с помощью Gunicorn.
4. Добавим контейнер с nginx для backend.
5. Добавим контейнер с nginx для frontend.

После локальной проверки мы перенесём проект на боевой VPS-сервер.

Настройка рабочей среды

Для работы нам понадобится Linux-система с установленными Docker и Docker Compose.

Внимание! *Есть варианты использования Docker в ОС Windows и macOS. На данный момент они не всегда работают стабильно. В качестве альтернативы можно использовать виртуальную машину с Linux-системой.*

Установка Docker описана в официальной документации, как и установка Docker Compose. Рекомендуем пользоваться именно этими инструкциями.

Структура проекта

Наш Django-проект находится в папке library, там же лежит файл manage.py. Мы будем работать на уровень выше. Создадим папку examples и поместим в неё папку library:

examples

→ **library**

→ **manage.py**

→ **…**

Папка examples будет корнем, она обозначена слешем (/). Все последующие действия относятся к ней.

Запуск базы данных postgres в отдельном контейнере

Для базы данных postgres мы будем использовать готовый официальный образ postgres с Docker Hub.

Также для проверки работоспособности возьмём образ adminer с Docker Hub. Это позволит подключиться к базе данных после запуска контейнера.

docker-compose.yml

Создадим файл docker-compose.yml со следующим содержанием:

```
version: '3.1'

services:
  db:
    image: postgres:13.1
      privileged: true
      ports:
        - 5432:5432
      environment:
       POSTGRES_USER: dante
       POSTGRES_PASSWORD: dante123456
        POSTGRES_DB: library
  adminer:
    image: adminer:4.7.8-standalone
      restart: always
      ports:
        - 8080:8080
      depends_on:
        - db
```

/docker-compose.yml

Рассмотрим основные настройки.

Мы описываем два контейнера `db`: для базы данных `postgres` и `adminer` для админки. Контейнер `adminer` зависит от контейнера `db`:

```
depends_on:
  - db
```

Для базы данных мы пробрасываем порт 5432, а для `adminer` — 8080:

```
ports:
  - 5432:5432
```

Для базы данных указываем имя пользователя, его пароль и название базы данных:

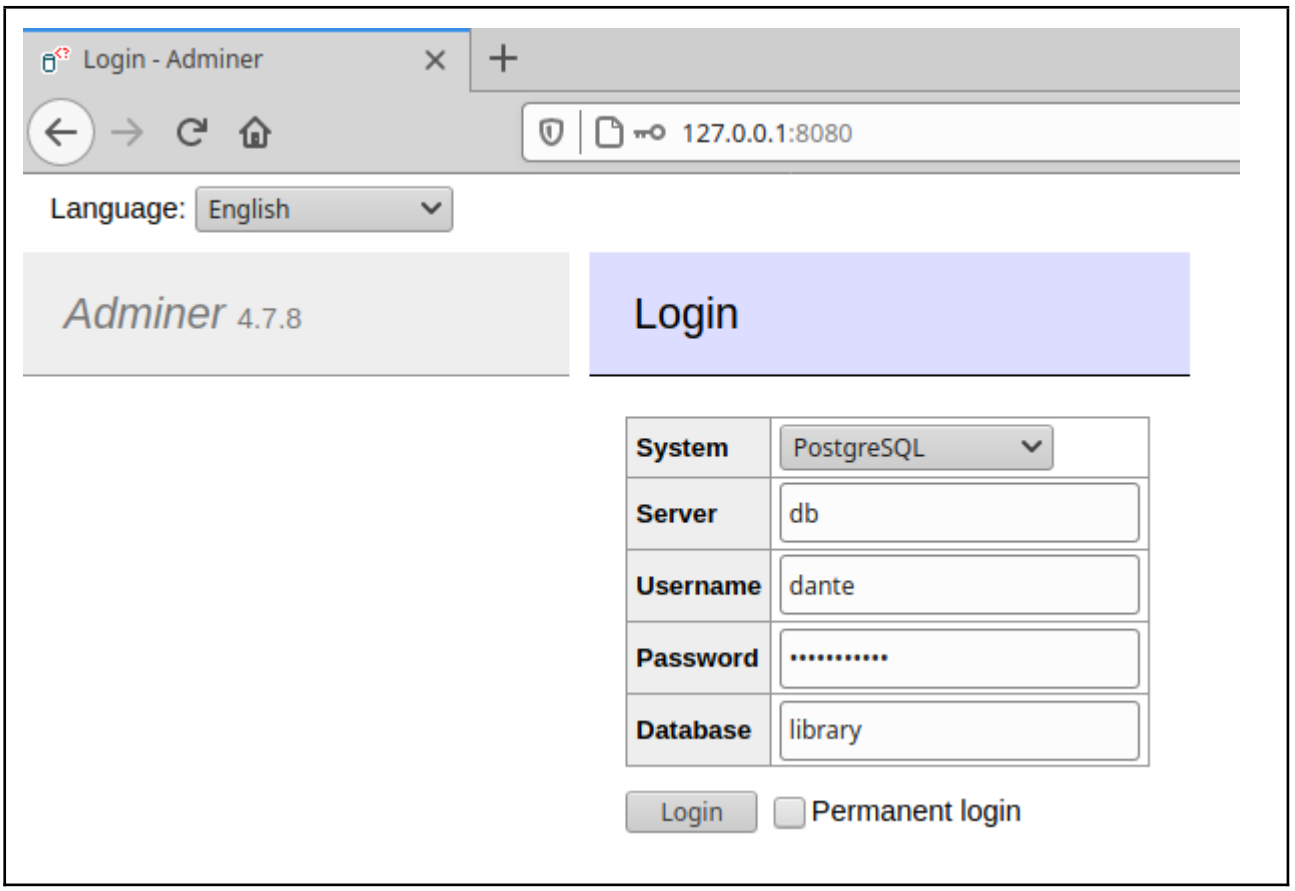
```
environment:
  POSTGRES_USER: dante
  POSTGRES_PASSWORD: dante123456
  POSTGRES_DB: library
```

Проверка работоспособности

После выполнения команды:

```
docker-compose up --build
```

переходим по адресу http://127.0.0.1:8080/ и пробуем подключиться к базе с помощью `adminer`.



Важно в поле `Server` вводить `db` — это хост, на котором находится контейнер с базой данных. Если после входа мы попадаем в панель управления базой данных, значит, всё работает без ошибок и можно переходить к следующему шагу.

Запуск тестового Django-сервера и выполнение миграций

Теперь соединим Django-проект с базой данных postgres и выполним миграции. Для начала воспользуемся тестовым сервером Django.

Контейнер `adminer` нам больше не нужен. Можно его оставить для прямого подключения к базе данных.

docker-compose.yml

Изменим файл docker-compose.yml следующим образом:

```
version: '3.1'

services:
  db:
    image: postgres:13.1
    privileged: true
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: dante
      POSTGRES_PASSWORD: dante123456
      POSTGRES_DB: library
  backend:
    build:
      context: .
    ports:
      - 8080:8080
    command: bash -c "
      ./wait-for-postgres.sh db
      && python manage.py migrate
      && python manage.py create_data
      && python manage.py runserver 0.0.0.0:8080
      "
    depends_on:
      - db
```

/docker-compose.yml

Контейнер с базой данных остался без изменений:

```
  backend:
    build:
      context: .
```

Контейнер `backend` будет собираться с помощью `Dockerfile`, который должен находиться в корне проекта:

```
ports:
  - 8080:8080
```

Пробрасываем порт 8080:

```
command: bash -c "
      ./wait-for-postgres.sh db
      && python manage.py migrate
      && python manage.py create_data
      && python manage.py runserver 0.0.0.0:8080
      "
```

После создания образа мы будем выполнять миграции, заполнять базу данных тестовыми данными и запускать тестовый сервер Django на порту 8080.

Выполнение скрипта `wait-for-postgres.sh` с параметром `db` нужно, чтобы дождаться готовности базы данных перед выполнением миграций.

```
 depends_on:
   - db
```

Контейнер `backend` зависит от контейнера с базой данных.

Dockerfile

Далее в корне проекта создадим файл Dockerfile со следующим содержанием:

```
from python:3.8.6

RUN apt-get update \
&& apt-get install -y postgresql postgresql-contrib libpq-dev python3-dev

RUN pip3 install --upgrade pip

COPY ./library/ ./
RUN pip3 install -r requirements.txt

COPY wait-for-postgres.sh .
RUN chmod +x wait-for-postgres.sh
```

/Dockerfile

Мы будем использовать официальный образ Python с Docker Hub:

```
from python:3.8.6
```

```
RUN apt-get update \
&& apt-get install -y postgresql postgresql-contrib libpq-dev python3-dev
```

Устанавливаем `postgres` и библиотеки, необходимые для работы с ней:

```
RUN pip3 install --upgrade pip
```

Обновляем `pip`:

```
COPY ./library/ ./
RUN pip3 install -r requirements.txt
```

Копируем файлы из проекта library в корень Docker-контейнера. И устанавливаем зависимости из файла requirements.txt

Внимание! *Для работы с базой данных `postgres` необходимо установить библиотеку `psycopg2`. Для этого нужно добавить соответствующую зависимость (`psycopg2==2.8.6`) в файл requirements.txt перед установкой зависимостей.*

```
COPY wait-for-postgres.sh .
RUN chmod +x wait-for-postgres.sh
```

Копируем скрипт `wait-for-postgres.sh` в корень Docker-контейнера и выдаём права на его запуск.

wait-for-postgres.sh

Проблема одновременного запуска базы данных `postgres` и выполнения миграций описана в официальной документации Docker. Дело в том, что хоть контейнер `backend` и стартует после контейнера `db`, но для полного запуска базы нужно определённое время. При попытке выполнить миграции может возникнуть ситуация, когда база данных ещё не готова, и возникнет ошибка.

В документации также описаны варианты решения этой проблемы. Мы будем использовать вариант проверки готовности базы данных с помощью sh-скрипта.

В корне проекта создадим файл wait-for-postgres.sh со следующим содержанием:

```
#!/bin/sh
# wait-for-postgres.sh

set -e

host="$1"
shift
cmd="$@"

until PGPASSWORD="dante123456" psql -h "$host" -d "library" -U "dante" -c '\q';>
  >&2 echo "Postgres is unavailable - sleeping"
  sleep 1
done

>&2 echo "Postgres is up - executing command"
exec $cmd
```

/wait-for-postgres.sh

Скрипт в цикле будет пробовать подключаться к базе данных и, когда произойдёт удачное подключение, совершит выход из бесконечного цикла.

settings.py

Для подключения к базе данных `postgres` необходимо в файле settings.py изменить настройки базы данных:

```
DATABASES = {
      'default': {
          'ENGINE': 'django.db.backends.postgresql',
          'NAME': 'library',
          'USER': 'dante',
          'PASSWORD': 'dante123456',
          'HOST': 'db',
```

```
            'PORT': '5432',
        }
}
```

/library/library/settings.py

Важный момент — указание `db` в качестве `HOST`.

Проверка работоспособности

Выполняем команду:

```
docker-compose up --build --remove-orphans
```

Ключ `remove-orphans` нужен, потому что мы удалили контейнер `adminer` и Docker не знает, переименовали мы контейнер или создали новый.

После запуска контейнеров переходим по адресу http://127.0.0.1:8080.

Если наш сайт на Django открылся и работает, можно переходить к следующему шагу.

Запуск Django-проекта с помощью Gunicorn

У нас готова dev-сборка с помощью Docker. Можно использовать такой вариант сборки для разработки проекта на локальной машине. Для production этот билд не подходит, так как используется тестовый сервер Django. Он не предназначен для многопользовательской работы сайта.

Нам нужен балансировщик нагрузки (например, nginx) для обработки множества запросов. А для связи Django-проекта и nginx мы будем использовать Gunicorn.

Dockerfile

В конце Dockerfile добавим одну строку для установки Gunicorn:

```
...
RUN pip3 install gunicorn
```

/Dockerfile

docker-compose.yml

В файле docker-compose.yml заменим одну строку в команде запуска сервера:

```
...
command: bash -c "
      ./wait-for-postgres.sh db
      && python manage.py migrate
      && python manage.py create_data
      && gunicorn library.wsgi -b 0.0.0.0:8080
      "
...
```

/docker-compose.yml

Команда `gunicorn library.wsgi -b 0.0.0.0:8080` запустит Django-приложение с помощью Gunicorn.

Проверка работоспособности

После запуска:

```
docker-compose up --build
```

переходим по адресу http://127.0.0.1:8080. Должен открыться сайт на Django, как и на предыдущем шаге, но уже без статики.

Контейнер с nginx для backend

Добавим контейнер для балансировщика нагрузки nginx. Для него мы будем использовать официальный образ с Docker Hub.

docker-compose.yml

Изменим код файла docker-compose.yml следующим образом:

```
version: '3.1'

services:
  db:
    image: postgres:13.1
    privileged: true
```

```
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: dante
      POSTGRES_PASSWORD: dante123456
      POSTGRES_DB: library
  backend:
    build:
    context: .
    expose:
      - 8080
    command: bash -c "
      ./wait-for-postgres.sh db
      && python manage.py migrate
      && python manage.py create_data
      && gunicorn library.wsgi -b 0.0.0.0:8080
      "
    depends_on:
      - db
  nginx:
    build: ./nginx
    ports:
      - 8000:80
    depends_on:
      - backend
```

/docker-compose.yml

В контейнере `backend` мы изменили настройку `ports` на `expose`. Порт 8080 будет доступен из внутренней сети и недоступен снаружи:

```
    expose:
      - 8080
```

Далее в конце мы добавили контейнер `nginx`:

```
nginx:
    build: ./nginx
    ports:
      - 8000:80
    depends_on:
      - backend
```

Он будет собираться из Dockerfile, который должен находиться в папке nginx. 80-й порт пробрасываем на 8000-й, так как на 80-м порту будет контейнер `frontend`. Контейнер `nginx` зависит от `backend`.

Dockerfile

В корне создаём папку nginx. В ней будут находиться два файла: Dockerfile и nginx.conf.

В Dockerfile добавим следующий код:

```
FROM nginx:1.19.6-alpine

RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d
```

/nginx/Dockerfile

Мы используем готовый образ nginx и просто копируем настройки из локальной папки в настройки nginx в Docker-контейнере.

nginx.conf

Далее создаём файл nginx.conf со следующим содержимым:

```
upstream library {
      server backend:8080;
}

server {
      listen 80;
      location / {
      proxy_pass http://library;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $host;
      proxy_redirect off;
      }
}
```

/nginx/nginx.conf

Nginx будет перенаправлять запросы к Gunicorn на 8080-м порту. Для этого мы указали:

```
upstream library {
      server backend:8080;
}
```

и

```
proxy_pass http://library;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
proxy_set_header Host $host;
```

Nginx будет находиться на 80-м порту:

```
listen 80;
```

Проверка работоспособности

Выполняем команду:

```
docker-compose up --build
```

Теперь при переходе по адресу http://127.0.0.1:8080 мы ничего не получим, так как этот порт скрыт внутри Docker'а. При переходе по адресу http://127.0.0.1:8000 увидим работающий сайт.

Контейнер с nginx для frontend

Мы собрали проект на Django с помощью Docker и Docker Compose. Такой вариант сборки уже можно использовать для production. Чтобы подключить frontend-часть, мы будем использовать отдельный контейнер.

Структура

В нашем проекте папка frontend находилась в папке library (/library/frontend). Так как мы хотим полностью разделить frontend и backend, перенесём эту папку в корень проекта (/frontend).

Перейдём в папку frontend:

```
cd frontend
```

И выполним команду для сборки:

```
npm run build
```

Теперь в папке frontend находятся файлы исходного кода (в папке src) и файлы сборки (в папке build). В неё же мы добавим Dockerfile и nginx.conf

docker-compose.yml

Для контейнера frontend мы будем использовать тот же образ nginx, что и для контейнера `"nginx"`. Внесём в файл docker-compose.yml следующие изменения:

```
...
frontend:
  build: ./frontend
  ports:
    - 80:80
  depends_on:
    - nginx
```

/docker-compose.yml

Dockerfile для сборки будет находиться в папке frontend. Открываем 80-й порт. Контейнер зависит от nginx.

Dockerfile

В папке frontend создадим Dockerfile со следующим содержимым:

```
FROM nginx:1.19.6-alpine
COPY ./build /var/www
COPY nginx.conf /etc/nginx/nginx.conf
```

/frontend/Dockerfile

Мы используем официальный образ nginx. Копируем содержимое папки build в /var/www и копируем файл nginx.conf с настройками сервера.

nginx.conf

В папке frontend создадим файл nginx.conf со следующим содержимым:

```
# auto detects a good number of processes to run
worker_processes auto;

#Provides the configuration file context in which the directives that affect
connection processing are specified.
events {
      # Sets the maximum number of simultaneous connections that can be opened
by a worker process.
```

```
        worker_connections 8000;
        # Tells the worker to accept multiple connections at a time
        multi_accept on;
}


http {
        # what times to include
        include       /etc/nginx/mime.types;
        # what is the default one
        default_type  application/octet-stream;

        # Sets the path, format, and configuration for a buffered log write
        log_format compression '$remote_addr - $remote_user [$time_local] '
        '"$request" $status $upstream_addr '
        '"$http_referer" "$http_user_agent"';

        server {
        # listen on port 80
        listen 80;
        # save logs here
        access_log /var/log/nginx/access.log compression;

        # where the root here
        root /var/www;
        # what file to server as index
        index index.html index.htm;

        location / {
                # First attempt to serve request as file, then
                # as directory, then fall back to redirecting to index.html
                try_files $uri $uri/ /index.html;
        }

        # Media: images, icons, video, audio, HTC
        location ~*
\.(?:jpg|jpeg|gif|png|ico|cur|gz|svg|svgz|mp4|ogg|ogv|webm|htc)$ {
        expires 1M;
        access_log off;
        add_header Cache-Control "public";
        }

        # Javascript and CSS files
        location ~* \.(?:css|js)$ {
                try_files $uri =404;
                expires 1y;
                access_log off;
                add_header Cache-Control "public";
        }

        # Any route containing a file extension (e.g. /devicesfile.js)
        location ~ ^.+\..+$ {
                try_files $uri =404;
```

```
        }
        }
}
```

/frontend/nginx.conf

В файле представлены различные настройки с соответствующими комментариями. Основная из них — указание статической страницы index.html:

```
        # where the root here
       root /var/www;
       # what file to server as index
       index index.html index.htm;

       location / {
              # First attempt to serve request as file, then
              # as directory, then fall back to redirecting to index.html
              try_files $uri $uri/ /index.html;
       }
```

CORS headers

После выполнения команды:

```
docker-compose up --build
```

мы можем проверить наличие backend-части проекта на порту 8000 и frontend-части на 80-м порту. Однако при попытке выполнить запрос от frontend к backend возникнет следующая ошибка:

Это происходит потому, что nginx-сервер на backend по умолчанию не принимает кроссдоменные запросы.

Отредактируем файл nginx/nginx.conf с настройками nginx для backend следующим образом:

```
upstream library {
      server backend:8080;
}

server {

      listen 80;

      location / {
      proxy_pass http://library;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $host;
      proxy_redirect off;

      # add_header Access-Control-Allow-Origin *;
      if ($request_method = 'OPTIONS') {
      add_header 'Access-Control-Allow-Origin' '*';
      #
      # Om nom nom cookies
      #
      add_header 'Access-Control-Allow-Credentials' 'true';
      add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, DELETE';

      #
      # Custom headers and headers various browsers *should* be OK with but
aren't
      #
      add_header 'Access-Control-Allow-Headers'
'DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cac
he-Control,Content-Type,Authorization';

      #
      # Tell client that this pre-flight info is valid for 20 days
      #
      add_header 'Access-Control-Max-Age' 1728000;
      add_header 'Content-Type' 'text/plain charset=UTF-8';
      add_header 'Content-Length' 0;
      return 204;
      }

      add_header 'Access-Control-Allow-Origin' '*';
      add_header 'Access-Control-Allow-Credentials' 'true';
      add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, DELETE';
      add_header 'Access-Control-Allow-Headers'
'DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cac
he-Control,Content-Type,Authorization';
```

```
        }
}
```

/nginx/nginx.conf

Расширенный пример настроек для каждого типа запроса описан в [статье Wide open nginx CORS configuration](). Нам достаточно отдельно описать OPTION-запрос и все остальные.

Мы разрешили кроссдоменные запросы и указали типы запросов, доступные на нашем сервере:

```
add_header 'Access-Control-Allow-Origin' '*';
add_header 'Access-Control-Allow-Credentials' 'true';
add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, DELETE';
```

Внимание! *Для большей безопасности вместо * можно указать конкретные домены, для которых будут разрешены запросы. Если в приложении будут использоваться другие методы, например PUT, PATCH, необходимо будет добавить их в список доступных методов.*

Проверка работоспособности

Всё готово для запуска нашего проекта.

Выполняем:

```
docker-compose up --build
```

Нам будут доступны frontend-часть сайта на порту 80 и backend-часть на порту 8000. При желании можно скрыть 8000-й порт, заменив настройку `ports` на `expose`.

Перенос проекта на VPS

Для демонстрации мы будем использовать облачный сервер на [хостинге REG.RU]().

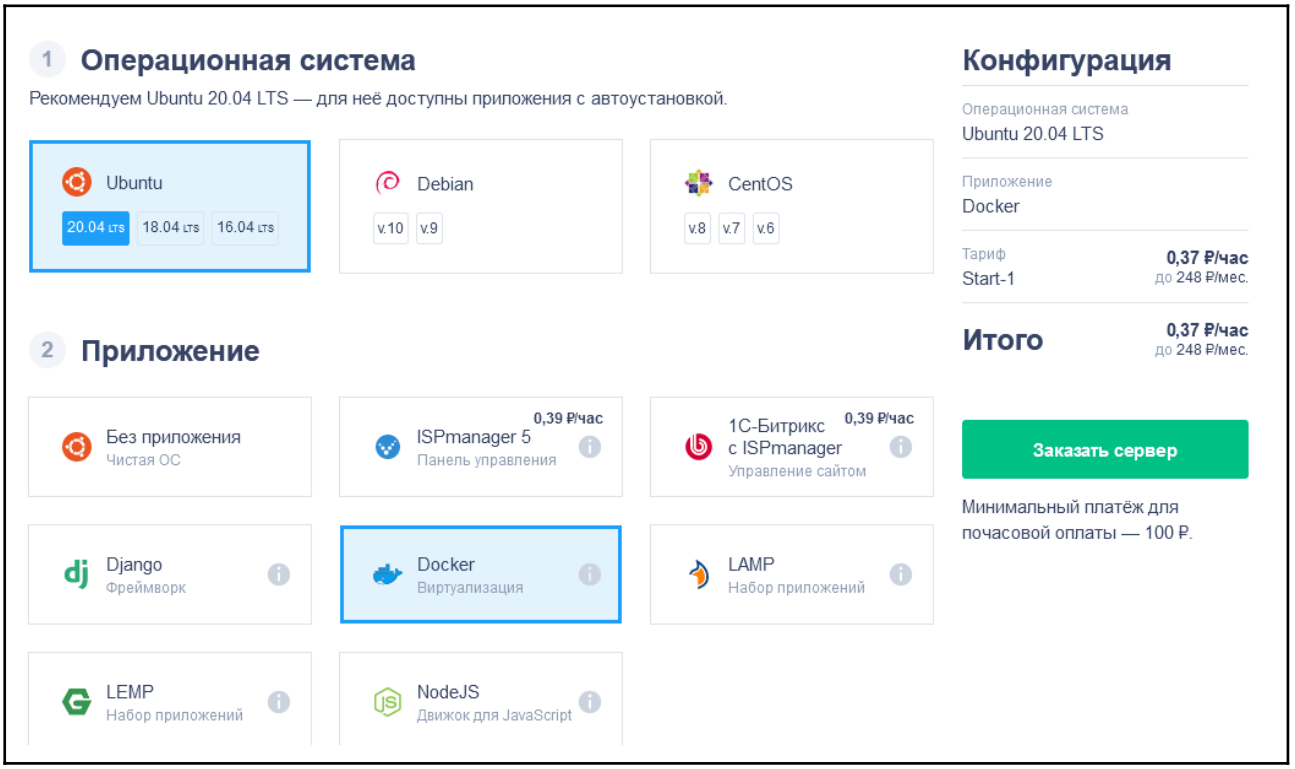
Регистрация на REG.RU

Если у вас уже есть аккаунт на REG.RU, нужно в него войти. Если аккаунта нет, [зарегистрироваться]().

После этого GeekBrains предоставит вам специальную ссылку на получение бонусов от REG.RU. При переходе по этой ссылке вы увидите сообщение о начислении бонусов и предложение создать облачный сервер.

Создание и настройка сервера

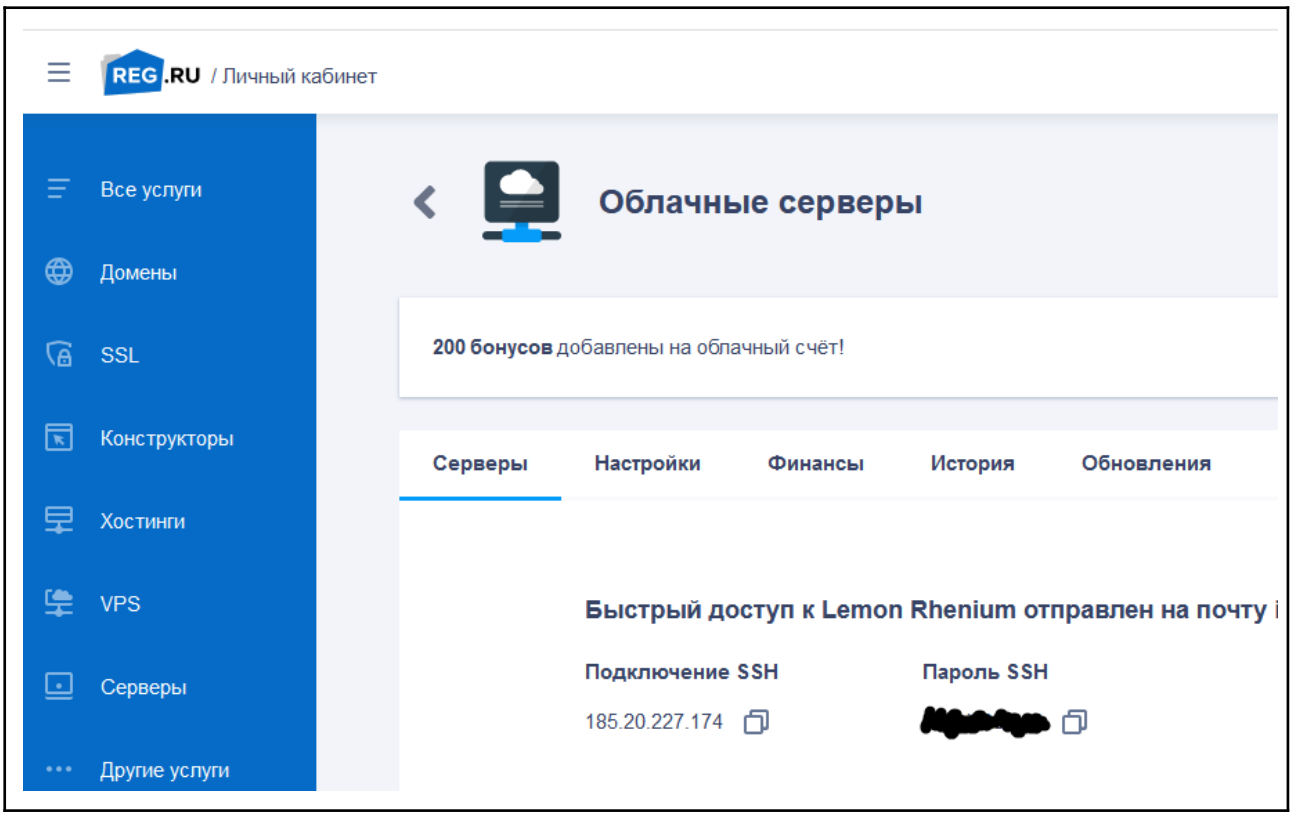
Выбираем «Создать сервер». И переходим на страницу настроек сервера:



Выбираем операционную систему. Также можно сразу выбрать предустановленный Docker, это удобно. Если выбрать «Без приложения», то Docker и Docker Compose нужно будет установить самостоятельно.

Ниже представлены различные конфигурации сервера (тарифы). От мощности зависит стоимость часа. На скриншоте был выбран самый простой тариф Start-1. Далее выбираем «Заказать сервер» и ждём некоторое время.

После создания сервера откроется страница с настройками сервера и параметрами подключения по SSH:



Подготовка проекта

В settings.py проекта в параметре `ALLOWED_HOSTS` нужно указать `'*'` либо IP-адрес сервера.

Далее в App.js заменяем IP-адрес всех запросов на IP-адрес сервера. Порт 8000 менять не нужно.

Выполняем команду для сборки frontend-части проекта в production:

```
npm run build
```

Перенос файлов на сервер

Подключаемся к серверу по SSH, используя данные в личном кабинете.

Далее нам нужно перекинуть файлы проекта. Это можно сделать любым удобным способом. При переносе через Git-репозиторий нужно убрать папку /build из .gitignore, так как по умолчанию она игнорируется. Файлы проекта можно поместить в любое место на сервере.

Развёртывание проекта

Переходим в корень проекта, где лежит файл docker-compose.yml.

Выполняем команду:

```
docker-compose up --build
```

Ждём, пока Docker создаст контейнеры и запустит серверы.

Далее переходим по IP-адресу сервера в браузере. Если ошибок нет, то наш сайт теперь работает в интернете.

Итоги

На этом занятии мы подготовили проект для production с помощью Docker и Docker Compose, развернули проект на облачном сервере на хостинге REG.RU.

Глоссарий

Облачный сервер — мощная физическая или виртуальная инфраструктура, выполняющая приложения и служащая хранилищем обрабатываемой информации. Облачные серверы создаются с помощью программного обеспечения для виртуализации, позволяющего разделить физический сервер на несколько виртуальных.

Дополнительные материалы

1. Настройки CORS на nginx.
2. Nginx и create-react-app.
3. Статья о развёртывании Django-приложения в Docker.
4. Официальный сайт Docker.
5. Docker Hub.

Практическое задание

Сборка проекта с помощью Docker и Docker Compose.

В этой самостоятельной работе мы тренируем умения:

- работать с Docker и Docker Compose;
- собирать проект на Django и React для production.

Смысл: использовать Docker и Docker Compose для разработки и развёртывания проектов.

Последовательность действий

1) Развернуть проект с помощью Docker и Docker Compose, используя один и более Docker-контейнер. Рекомендуются следующие отдельные контейнеры:
    - db-контейнер с базой данных postgresql;
    - nginx;
    - gunicorn (gunicorn + сайт на Django).
2) Разместить проект на хостинге REG.RU.
3) * Сделать отдельный контейнер для frontend-части приложения (nginx + frontend).