

Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Лабораторная работа №1
по дисциплине
«Облачные и туманные вычисления»

Выполнил студент группы Р3415
Барсуков Максим Андреевич

Преподаватель:
Перл Ольга Вячеславовна

Санкт-Петербург
2025 г.

Содержание

- Задание.....2
- Приложение.....3
 - Frontend.....3
 - Backend.....3
- Ход работы.....4
 - Контейнеризация.....4
 - Docker-Compose.....5
 - Инициализация Swarm.....7
 - Развертывание стека.....7
 - Тест на отказ.....8
 - Масштабирование.....8
- Заключение.....9

Задание

Цель: Изучить разницу между простым запуском контейнера и оркестрацией сервиса. Эта работа закрывает пробел между «работает на моем ноутбуке» и «работает в кластере».

Сценарий: Вы выступаете в роли DevOps-инженеров простого интернет-магазина, которому нужно справиться с наплывом трафика (например, в «Черную пятницу»).

Приложение:

- **Frontend:** Веб-страница на Python/Flask, отображающая счётчик посещений.
- **Backend:** База данных Redis для хранения количества хитов.

Ход работы:

1. **Контейнеризация:** Написать Dockerfile для Python-приложения.
2. **Docker Compose:** Написать docker-compose.yml для локального запуска связки App + Redis. Убедиться, что они «видят» друг друга.
3. **Инициализация Swarm:** Перевести Docker в режим Swarm (docker swarm init).
4. **Развертывание стека (Deploy Stack):** Конвертировать файл compose в деплой стека (Stack Deploy).
5. **Тест на отказ (Self-healing):** Вы вручную «убиваете» (docker kill) активный контейнер и наблюдают, как Swarm автоматически пересоздаёт его.
6. **Масштабирование:** Выполнить команду docker service scale frontend=5.

Ключевые знания: Обнаружение сервисов (Service Discovery) — как приложение находит Redis по имени), реплики и самовосстанавливающиеся системы.

Приложение

Frontend

```
import os
from flask import Flask, jsonify
import redis

redis_client = redis.Redis(
    host=os.getenv("REDIS_HOST", "redis"),
    port=int(os.getenv("REDIS_PORT", "6379")),
    db=int(os.getenv("REDIS_DB", "0")),
    password=os.getenv("REDIS_PASSWORD"),
    decode_responses=True,
)

app = Flask(__name__)

@app.route("/")
def index():
    count = redis_client.incr("hits")
    return jsonify(visit_count=count), 200

@app.route("/health")
def health():
    try:
        redis_client.ping()
        return jsonify(status="ok"), 200
    except redis.RedisError as exc:
        return jsonify(status="failed", detail=str(exc)), 500

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Backend

Хранилище представлено Redis. Для локального стенда пароль берётся из файла `.env`, порт Redis не открывается наружу. Данные в именованном volume `redis-data`.

Ход работы

Контейнеризация

Для Python-сервиса собран образ на базе python:3.11-slim. Переменные окружения задают адрес Redis и параметры запуска Flask. Зависимости ставятся отдельно.

```
FROM python:3.11-slim
```

```
ENV PYTHONDONTWRITEBYTECODE=1 \  
    PYTHONUNBUFFERED=1 \  
    FLASK_APP=app.py \  
    FLASK_RUN_HOST=0.0.0.0 \  
    FLASK_RUN_PORT=5000
```

```
WORKDIR /app
```

```
COPY app/requirements.txt ./
```

```
RUN pip install --no-cache-dir -r requirements.txt
```

```
COPY app/ ./
```

```
EXPOSE 5000
```

```
CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

Список зависимостей приложения в requirements.txt:

```
Flask==2.3.3  
redis==5.0.1
```

Docker-Compose

Файл `docker-compose.yml` описывает два сервиса: `frontend` и `redis`. Пароль Redis и остальные переменные читаются из `.env`. Порт Redis наружу не открывается, фронтэнд доступен на `localhost:5002`.

```
services:
  frontend:
    build: .
    image: lab1-frontend:latest
    ports:
      - "5002:5000"
    env_file: .env
    environment:
      - REDIS_HOST=${REDIS_HOST:-redis}
      - REDIS_PORT=${REDIS_PORT:-6379}
      - REDIS_DB=${REDIS_DB:-0}
      - REDIS_PASSWORD=${REDIS_PASSWORD}
    depends_on:
      - redis
    deploy:
      replicas: 1
      restart_policy:
        condition: any
  redis:
    image: redis:7-alpine
    env_file: .env
    command: ["redis-server", "--appendonly", "yes", "--requirepass", "${REDIS_PASSWORD}"]
    volumes:
      - redis-data:/data
    deploy:
      replicas: 1
      restart_policy:
        condition: any

volumes:
  redis-data:
```

Пример конфигурации окружения (в репозитории хранится `.env.example`):

```
REDIS_PASSWORD=password
REDIS_HOST=redis
REDIS_PORT=6379
REDIS_DB=0
```

Локальный запуск и проверка (до перехода в Swarm), как показано на рисунках 1, 2:

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

```
docker compose up --build
[+] Building 0.4s (12/12) FINISHED
=> [internal] load local bake definitions
=> reading from stdin 592B
=> [internal] load build definition from Dockerfile
=> transferring dockerfile: 372B
=> [internal] load metadata for docker.io/library/python:3.11-slim
=> [internal] load dockerignore
=> transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.11-slim@sha256:8b2cfd528ad7c43ba8de95e2788393cf545aaed73868e9e1454bd38a76ba
=> transferring context: 94B
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY app/requirements.txt ./
=> CACHED [4/5] RUN pip install --no-cache-dir -r requirements.txt
=> CACHED [5/5] COPY app/ ./
=> exporting layers
=> writing image sha256:4420f07cb91934d4e878d18e58a2dd40edc8a984063dd618fde60cf9052ac1ea
=> naming to docker.io/library/lab1-frontend:latest
=> resolving provenance for metadata file
[+] Running 2/2
  lab1-frontend:latest Built
  Container lab1-frontend-1 Recreated
Attaching to frontend-1, redis-1
redis-1 11C 30 Dec 2025 15:35:28.335 # WARNING Memory overcommit must be enabled! Without it, a background save or replication may fail under low memory condition. Being d
jemalloc/issues/1328. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take
redis-1 11C 30 Dec 2025 15:35:28.335 * oOoOoOoOoOoOo Redis is starting oOoOoOoOoOoOo
redis-1 11C 30 Dec 2025 15:35:28.335 * Redis version=7.4.7, bits=64, commit=00000000, modified=0, pid=1, just started
redis-1 11C 30 Dec 2025 15:35:28.335 * Configuration loaded
redis-1 11M 30 Dec 2025 15:35:28.335 * Increased maximum number of open files to 10032 (it was originally set to 1024).
redis-1 11M 30 Dec 2025 15:35:28.335 * Monotonic clock: POSIX clock_gettime
redis-1 11M 30 Dec 2025 15:35:28.336 * Running mode=standalone, port=6379.
redis-1 11M 30 Dec 2025 15:35:28.336 * Server initialized
redis-1 11M 30 Dec 2025 15:35:28.336 * Reading RDB base file on AOF loading...
redis-1 11M 30 Dec 2025 15:35:28.336 * Loading RDB produced by version 7.4.7
redis-1 11M 30 Dec 2025 15:35:28.336 * RDB age 7 seconds
redis-1 11M 30 Dec 2025 15:35:28.336 * RDB memory usage when created 0.91 Mb
redis-1 11M 30 Dec 2025 15:35:28.336 * RDB is base AOF
redis-1 11M 30 Dec 2025 15:35:28.336 * Done loading RDB, keys loaded: 0, keys expired: 0.
redis-1 11M 30 Dec 2025 15:35:28.336 * DB loaded from base file appendonly.aof.1.base.rdb: 0.000 seconds
redis-1 11M 30 Dec 2025 15:35:28.336 * DB loaded from append only file: 0.000 seconds
redis-1 11M 30 Dec 2025 15:35:28.336 * Opening AOF incr file appendonly.aof.1.incr.aof on server start
redis-1 11M 30 Dec 2025 15:35:28.336 * Ready to accept connections tcp
frontend-1 * Serving Flask app "app.py"
frontend-1 * Debug mode: off
frontend-1 WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
frontend-1 * Running on all addresses (0.0.0.0)
frontend-1 * Running on http://127.0.0.1:5000
frontend-1 * Running on http://172.20.0.3:5000
frontend-1 Press CTRL-C to quit
frontend-1 172.20.0.1 - - [30/Dec/2025 15:36:06] "GET / HTTP/1.1" 200 -
frontend-1 172.20.0.1 - - [30/Dec/2025 15:36:13] "GET / HTTP/1.1" 200 -
frontend-1 172.20.0.1 - - [30/Dec/2025 15:36:13] "GET / HTTP/1.1" 200 -
[+] Enable Watch
```

Рисунок 1 – Запуск команды `docker compose up --build`

```
curl -s http://localhost:5002/
```

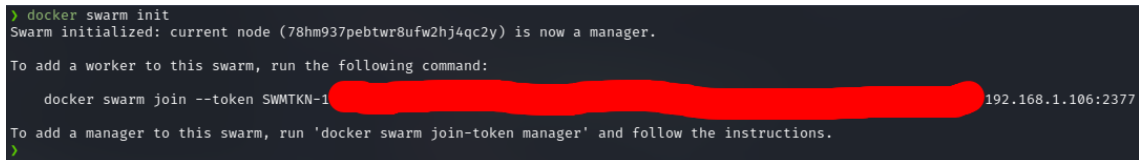
```
> curl -s http://localhost:5002/
{"visit_count":1}
>
> curl -s http://localhost:5002/
{"visit_count":2}
> curl -s http://localhost:5002/
{"visit_count":3}
>
>
```

Рисунок 2 – Проверка работы сервиса

Инициализация Swarm

Перевод Docker в режим оркестрации. Команда выводит токены для присоединения узлов, как показано на рисунке 3:

```
docker swarm init
```

A terminal window showing the output of the 'docker swarm init' command. The output indicates that the current node is now a manager and provides a token for adding workers. The token is partially obscured by a red redaction bar. The IP address 192.168.1.106:2377 is visible at the end of the line.

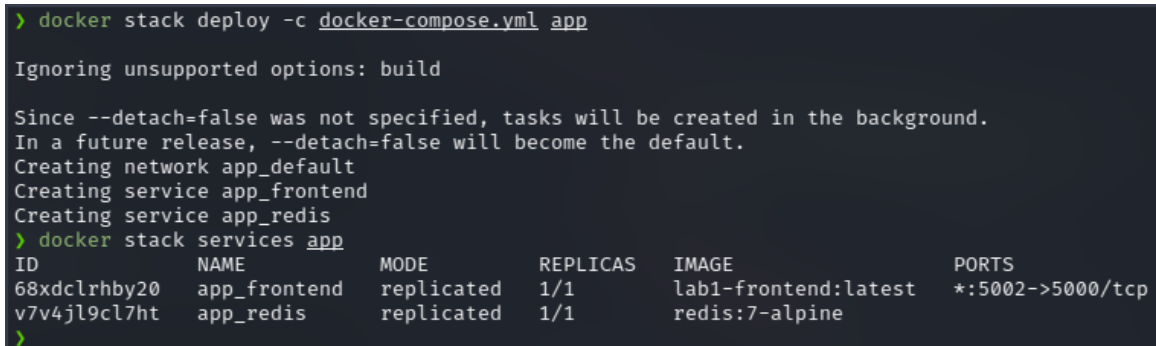
```
> docker swarm init
Swarm initialized: current node (78hm937pebtwr8ufw2hj4qc2y) is now a manager.
To add a worker to this swarm, run the following command:
    docker swarm join --token SWMTKN-1-[REDACTED] 192.168.1.106:2377
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
>
```

Рисунок 3 – Запуск docker swarm

Развертывание стека

Перед деплоем образ фронтенда собирается и тегируется, затем запускается стек app. Список сервисов показывает доступный порт фронта, как показано на рисунке 4.

```
docker compose build frontend
docker stack deploy -c docker-compose.yml app
docker stack services app
```

A terminal window showing the output of the 'docker stack deploy' and 'docker stack services' commands. The first command shows the stack being deployed with services 'app_frontend' and 'app_redis'. The second command shows the details of the services, including their IDs, names, modes, replicas, images, and ports.

```
> docker stack deploy -c docker-compose.yml app
Ignoring unsupported options: build
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
Creating network app_default
Creating service app_frontend
Creating service app_redis
> docker stack services app
ID            NAME          MODE           REPLICAS  IMAGE              PORTS
68xdclrhby20 app_frontend  replicated     1/1        lab1-frontend:latest *:5002->5000/tcp
v7v4jl9cl7ht app_redis     replicated     1/1        redis:7-alpine
```

Рисунок 4 – Развертывание стека

Тест на отказ

Одна из реплик фронтенда намеренно завершается, после чего планировщик автоматически поднимает новую задачу, что видно по списку задач сервиса на рисунке 5.

```
docker kill app_frontend.1.ow8qi7jca4i2mqlksutx6gfsf
docker service ps app_frontend
```

```
> docker kill app_frontend.1.ow8qi7jca4i2mqlksutx6gfsf
app_frontend.1.ow8qi7jca4i2mqlksutx6gfsf
>
>
> docker service ps app_frontend
ID            NAME                IMAGE                NODE    DESIRED STATE  CURRENT STATE          ERROR                                     PORTS
6l0eecthr7pc  app_frontend.1      lab1-frontend:latest laptop  Ready          Ready 1 second ago
ow8qi7jca4i2  _ app_frontend.1    lab1-frontend:latest laptop  Shutdown      Failed 4 seconds ago   "task: non-zero exit (137)"
> docker service ps app_frontend
ID            NAME                IMAGE                NODE    DESIRED STATE  CURRENT STATE          ERROR                                     PORTS
6l0eecthr7pc  app_frontend.1      lab1-frontend:latest laptop  Running        Running 13 seconds ago
ow8qi7jca4i2  _ app_frontend.1    lab1-frontend:latest laptop  Shutdown      Failed 18 seconds ago   "task: non-zero exit (137)"
```

Рисунок 5 – Тест на отказ

Масштабирование

Масштабирование выполняется командой `docker service scale`. После этого у сервиса стало 5 реплик.

```
docker service scale app_frontend=5
```

```
> docker service scale app_frontend=5
app_frontend scaled to 5
overall progress: 5 out of 5 tasks
1/5: running [=====>]
2/5: running [=====>]
3/5: running [=====>]
4/5: running [=====>]
5/5: running [=====>]
verify: Service app_frontend converged
```

Рисунок 6 – Масштабирование сервиса

Заключение

В ходе выполнения лабораторной работы было разработано и развернуто контейнеризованное веб-приложение на Flask с БД в Redis. Проект был первоначально настроен для локальной разработки с использованием Docker Compose, после чего та же конфигурация была успешно применена для развертывания в кластере Docker Swarm. В ходе работы были продемонстрированы ключевые возможности оркестратора: отказоустойчивость (автоматическое восстановление сервиса после сбоя) и горизонтальное масштабирование (увеличение количества экземпляров сервиса Flask до пяти). Конфигурационные параметры централизованы в файле `.env`, а доступ к Redis ограничен внутренней сетью. В результате получен опыт переноса приложения из среды одиночных контейнеров в полноценную оркестрируемую инфраструктуру.