

Департамент образования и науки города Москвы  
Государственное автономное образовательное учреждение высшего  
образования города Москвы  
«Московский городской педагогический университет»  
Институт цифрового образования  
Департамент информатики, управления и технологий

**ДИСЦИПЛИНА:**

Интеграция и развертывание программного обеспечения с помощью  
контейнеров

**Лабораторная работа №3.2**

**Тема:**

«Развертывание приложения в Kubernetes»

Выполнил(а): st\_98, группа: АДЭУ-211

Преподаватель:

Москва

2025

**Цель работы:** освоить процесс развертывания приложения в Kubernetes с использованием Deployments и Services

**Задачи:**

1. Создать Deployment для указанного приложения.
2. Создать Service для обеспечения доступа к приложению.
3. Проверить доступность приложения через созданный Service.
4. Выполнить индивидуальное задание.

**Вариант 11 (st\_98):**

Разверните приложение на Flask, использующее базу данных SQLite и Gunicorn в качестве сервера приложений, в Kubernetes. Создайте Deployment для Flask и Gunicorn, а также Service для доступа к приложению.

## Ход работы:

В первую очередь устанавливаем и запускаем Minikube. На рисунке 1 можем увидеть, что кластер запущен.



```
dev@dev-vm: ~/lr3_1$ minikube start --memory=2048mb --driver=docker
minikube v1.35.0 on Ubuntu 22.04 (vbox/amd64)
Using the docker driver based on user configuration
Using Docker driver with root privileges
Starting "minikube" primary control-plane node in "minikube" cluster
Pulling base image v0.0.46 ...
Downloading Kubernetes v1.32.0 preload ...
> preloaded-images-k8s-v18-v1...: 333.57 MiB / 333.57 MiB 100.00% 9.93 Mi
> gcr.io/k8s-minikube/kicbase...: 500.25 MiB / 500.31 MiB 99.99% 12.04 Mi
Creating docker container (CPUs=2, Memory=2048MB) ...
Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
  ■ Generating certificates and keys ...
  ■ Booting up control plane ...
  ■ Configuring RBAC rules ...
Configuring bridge CNI (Container Networking Interface) ...
Verifying Kubernetes components...
  ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
Enabled addons: storage-provisioner, default-storageclass
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
dev@dev-vm: ~/lr3_1$
```

Рисунок 1 – Запуск Minikube

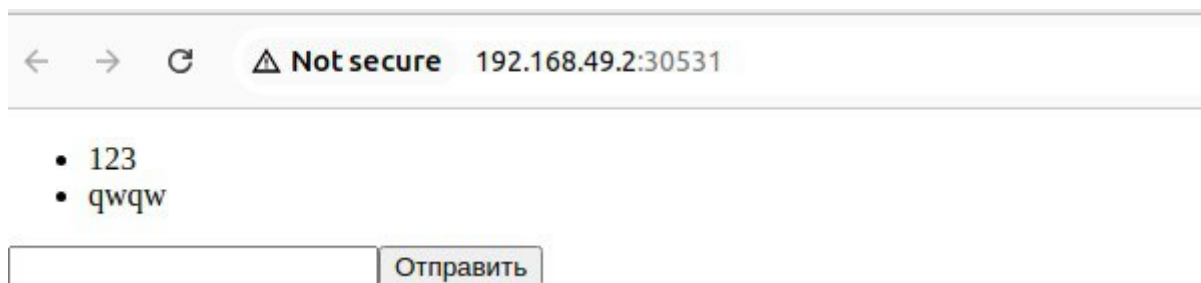
Далее применяем конфигурацию Deployment и выводим поды для просмотра их статуса. Результат представлен на рисунке 2.



```
dev@dev-vm: ~/nodejs-chat-app$ kubectl apply -f deployment.yaml
deployment.apps/nodejs-chat-app created
dev@dev-vm: ~/nodejs-chat-app$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nodejs-chat-app-545ccb79d4-8v8jm    1/1     Running   0           10s
nodejs-chat-app-545ccb79d4-mfdxz    1/1     Running   0           10s
nodejs-chat-app-545ccb79d4-tsphm    1/1     Running   0           10s
dev@dev-vm: ~/nodejs-chat-app$
```

Рисунок 2 – Созданные модули

Чат доступен ссылке, что доказывает работоспособность (см. рисунок 3).



← → ↻ ⚠ Not secure 192.168.49.2:30531

- 123
- qwqw

Рисунок 3 – Чат

Переходим к выполнению индивидуального задания.

Сначала были созданы файлы: dockerfile, flask-deployment.yaml, flask-service.yaml. Их код представлен на картинках 4-6 соответственно.

```
task > Dockerfile > FROM
1 FROM python:3.9-slim
2
3 WORKDIR /app
4
5 COPY requirements.txt /app/
6
7 RUN pip install --no-cache-dir -r requirements.txt
8
9 COPY . /app/
10
11 EXPOSE 5000
12
13 CMD ["gunicorn", "-b", "0.0.0.0:5000", "app:app"]
```

Рисунок 4 – dockerfile

```
task > ! flask-deployment.yaml
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: flask-app
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: flask-app
10  template:
11    metadata:
12      labels:
13        app: flask-app
14    spec:
15      containers:
16      - name: flask-app
17        image: flask-app:local
18      ports:
19      - containerPort: 5000
20
```

Рисунок 5 – flask-deployment.yaml

```

task > ! flask-service.yaml
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: flask-app-service
5  spec:
6    selector:
7      app: flask-app
8    ports:
9      - protocol: TCP
10        port: 80
11        targetPort: 5000
12    type: LoadBalancer

```

Рисунок 6 – flask-service.yaml

Запускаем flask-deployment и flask-service, проверяем, что модули начали работу и получаем ссылку для внешнего подключения. Результаты представлены на рисунке 7.

```

-- -- naming to docker.io/library/task-app:local
• (venv) dev@dev-vm:~/nodejs-chat-app/task$ kubectl apply -f flask-deployment.yaml
deployment.apps/flask-app created
• (venv) dev@dev-vm:~/nodejs-chat-app/task$ kubectl apply -f flask-service.yaml
service/flask-app-service created
• (venv) dev@dev-vm:~/nodejs-chat-app/task$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
flask-app-5b7664b767-bgk2l         1/1     Running   0           46s
flask-app-5b7664b767-dgk2s         1/1     Running   0           46s
flask-app-5b7664b767-psrqf         1/1     Running   0           46s
• (venv) dev@dev-vm:~/nodejs-chat-app/task$ minikube service flask-app-service --url
http://192.168.49.2:31504

```

Рисунок 7 – Запуск модулей

Перейдя по ссылке, отработал endpoint (рисунок 8) приложения flask, на рисунке 9 можно увидеть тестовых пользователей.

```

    return jsonify({"message": str(e)}), 400

@app.route('/get_users', methods=['GET'])
def get_users():
    users = User.query.all()
    return jsonify([{'id': user.id, 'name': user.name, 'email': user.email} for user in users])

if __name__ == '__main__':
    app.run(debug=True)

```

Рисунок 8 – Метод получения пользователей

```

← → ↻ ⚠ Not secure 192.168.49.2:31504/get_users
pretty print ☐
[{"email": "alice@example.com", "id": 1, "name": "Alice"}, {"email": "bob@example.com", "id": 2, "name": "Bob"}, {"email": "charlie@example.com", "id": 3, "name": "Charlie"}]

```

Рисунок 9 – Тестовые пользователи Flask

## **Вывод:**

1. Освоены основные этапы деплоя приложений в Kubernetes.
2. Приобретены навыки работы с Deployment (управление подами) и Service (обеспечение доступа к приложению).
3. Успешно развернуто Flask-приложение с использованием Gunicorn и SQLite, что подтверждает его работоспособность в Kubernetes-кластере.

## **Контрольные вопросы:**

### 1. Что такое Pod, Deployment и Service в Kubernetes?

Pod: наименьшая разворачиваемая единица в Kubernetes. Pod может содержать один или несколько контейнеров, которые совместно используют одно и то же сетевое пространство имен и хранилище. Pod являются эфемерными и могут быть созданы, уничтожены или реплицированы на основе желаемого состояния, определенного пользователем.

Deployment: абстракция более высокого уровня, которая управляет набором идентичных Pod. Он обеспечивает запуск нужного количества Pod и может беспрепятственно обрабатывать обновления приложения.

Сервис: абстракция, которая определяет логический набор Pod и политику доступа к ним. Сервисы обеспечивают связь между различными частями вашего приложения и предоставляют стабильные конечные точки для доступа к Pod.

### 2. Каково назначение Deployment в Kubernetes?

Deployment нужен для:

- Декларативного управления подами (Kubernetes сам поддерживает желаемое состояние).
- Масштабирования (можно увеличить или уменьшить количество реплик).
- Обновления приложений (rolling updates, blue-green deployments).
- Отката при неудачном обновлении.

### 3. Каково назначение Service в Kubernetes?

Service решает следующие задачи:

- Постоянный доступ к динамически меняющимся подам (поды могут пересоздаваться, но сервис остается).
- Балансировка нагрузки между подами.
- Предоставление единой точки входа (вместо прямого доступа к подам).

### 4. Как создать Deployment в Kubernetes?

- 1) Создать YAML-файл (например, deployment.yaml).
- 2) Запустить:

```
kubectl apply -f deployment.yaml
```

- 3) Проверить:

```
kubectl get deployments
```

```
kubectl get pods
```

## 5. Как создать Service в Kubernetes и какие типы Services существуют?

- 1) Создать YAML-файл (например, service.yaml).
- 2) Запустить:

```
kubectl apply -f service.yaml
```

- 3) Проверить:

```
kubectl get services
```

### **Типы сервисов:**

1. ClusterIP (по умолчанию)

Назначение: внутренний сервис, доступный только внутри кластера.

Как работает:

- Присваивает сервису внутренний IP-адрес.
- Другие поды или сервисы в кластере могут обращаться к нему по этому IP или DNS-имени.

### 2. NodePort

Назначение: открывает статический порт на каждой ноде (узле) кластера, позволяя обращаться к сервису через IP любой ноды.

Как работает:

- Kubernetes назначает порт из диапазона 30000-32767 (или можно указать вручную).
- Запросы на <NodeIP>:<NodePort> перенаправляются в сервис.

### 3. LoadBalancer

Назначение: создает внешний балансировщик нагрузки (обычно в облачных провайдерах: AWS, GCP, Azure).

Как работает:

- Kubernetes автоматически запрашивает у облачного провайдера балансировщик.
- Внешний IP балансировщика направляет трафик на поды.

### 4. ExternalName

Назначение: связывает сервис с DNS-именем вне кластера (например, внешний API или база данных).

Как работает:

- Вместо селектора (selector) указывается externalName.
- Запросы к сервису перенаправляются на указанное DNS-имя.