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

Макарова Екатерина Павловна

ЛАБОРАТОРНАЯ РАБОТА 3.2.

Развертывание приложения в Kubernetes

Интеграция и развертывание программного обеспечения с помощью контейнеров

Направление подготовки
38.03.05 Бизнес-информатика
Профиль подготовки
Аналитика данных и эффективное управление

Курс обучения: 4
Форма обучения: очная
Преподаватель: кандидат технических наук,
доцент Босенко Тимур Муртазович

Москва
2025

Цель: 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.

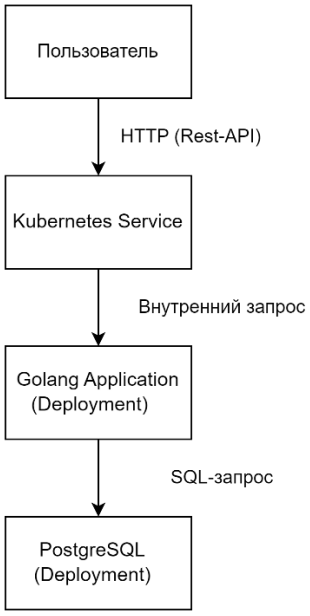
Вариант 8. Разверните приложение на Golang, использующее базу данных PostgreSQL, в Kubernetes. Создайте Deployment для Golang и PostgreSQL, а также Service для доступа к приложению.

Постановка задачи: приложение будет представлять собой API для управления списком задач (TODO list). Пользователи смогут добавлять, просматривать, обновлять и удалять задачи.

Технический стек:

1. Язык программирования: Golang
2. База данных: PostgreSQL
3. Оркестрация: Kubernetes
4. Контейнеризация: Docker
5. Маршрутизация API: Gorilla Mux
6. Сетевые протоколы: HTTP, TCP
7. Сервис обнаружения и балансировки нагрузки: Kubernetes Service

Архитектура



Функциональные требования

1. Управление задачами (TODO list):

- Добавление новой задачи.
- Просмотр списка всех задач.
- Обновление существующей задачи.

- Удаление задачи.

2. API Endpoints:

- GET /tasks — возвращает список всех задач.
- POST /tasks — создает новую задачу.
- PUT /tasks/{id} — обновляет задачу по ID.
- DELETE /tasks/{id} — удаляет задачу по ID.

3. Хранение данных:

- Все задачи должны храниться в базе данных PostgreSQL.

Ход работы

1. Подготовка к работе:

   Включение K8S в Docker Desktop (Рисунок 1).

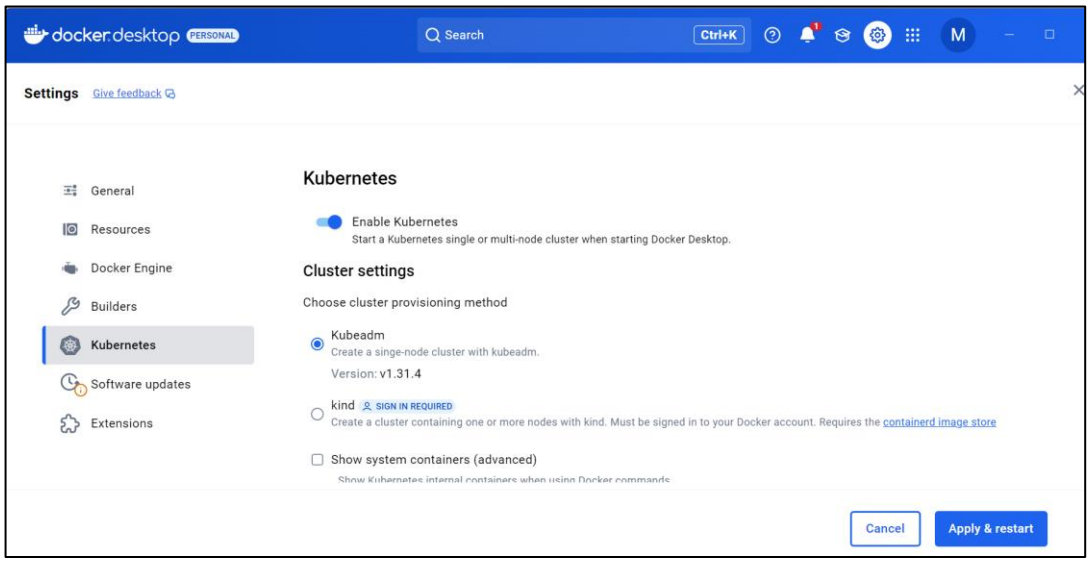


Рисунок 1

Кластер запущен (Рисунок 2).

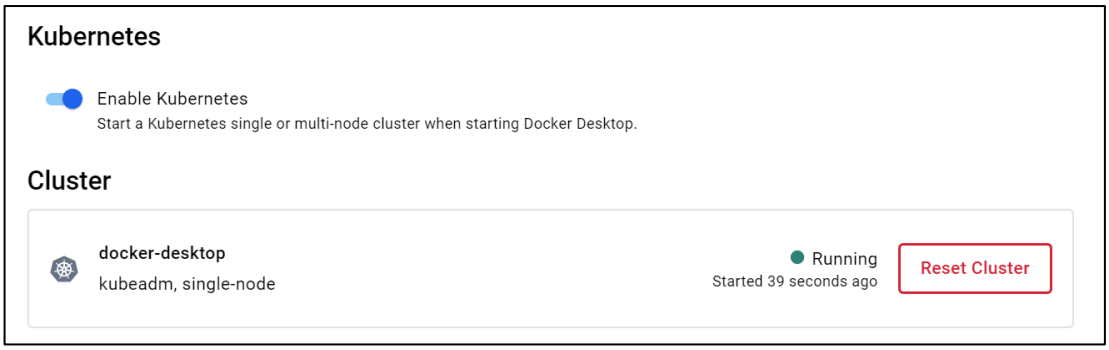


Рисунок 2

Проверка подключения kubectl к кластеру k8s (Рисунок 3).

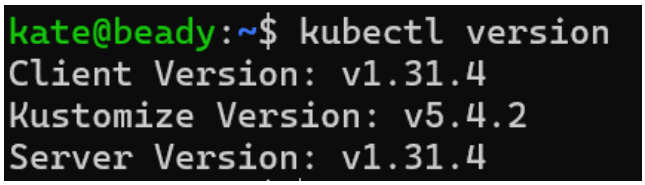


Рисунок 3

Проверка статуса работы кластера (Рисунок 4).

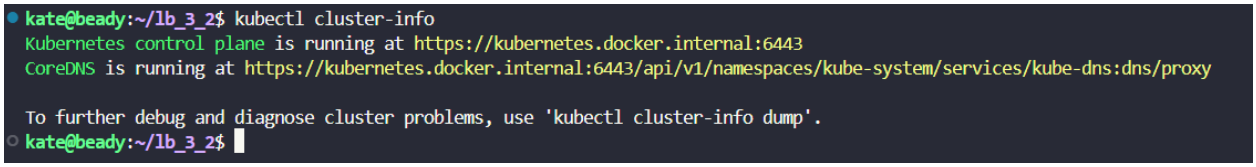


Рисунок 4

Проверка нодов (Рисунок 5).

```
kate@beady:~/lb_3_2$ kubectl get nodes
NAME             STATUS   ROLES           AGE   VERSION
docker-desktop   Ready    control-plane   11m   v1.31.4
kate@beady:~/lb_3_2$
```

Рисунок 5

Создание директории для проекта (Рисунок 6).

```
kate@beady:~$ mkdir todo-app
kate@beady:~$ cd todo-app
kate@beady:~/todo-app$
```

Рисунок 6

2. Подготовка приложения

1) Создание main.go (Рисунок 7).

main.go

```go
package main

import (
    "database/sql"
    "fmt"
    "log"
    "net/http"
    "github.com/gorilla/mux"
    _ "github.com/lib/pq"
)

type Task struct {
    ID    int    `json:"id"`
    Title string `json:"title"`
}

var db *sql.DB

func initDB() {
    var err error
    connStr := "user=postgres dbname=todo sslmode=disable password=yourpassword host=postgres"
    db, err = sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal(err)
    }

    if err = db.Ping(); err != nil {
        log.Fatal(err)
    }

    _, err = db.Exec("CREATE TABLE IF NOT EXISTS tasks (id SERIAL PRIMARY KEY, title TEXT)")
    if err != nil {
        log.Fatal(err)
    }
}

func getTasks(w http.ResponseWriter, r *http.Request) {
    rows, err := db.Query("SELECT id, title FROM tasks")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
```

main.go

```
37  func getTasks(w http.ResponseWriter, r *http.Request) {
46      for rows.Next() {
47          var task Task
48          if err := rows.Scan(&task.ID, &task.Title); err != nil {
49              http.Error(w, err.Error(), http.StatusInternalServerError)
50              return
51          }
52          tasks = append(tasks, task)
53      }
54
55      w.Header().Set("Content-Type", "application/json")
56      fmt.Fprintf(w, "%v", tasks)
57  }
58
59  func addTask(w http.ResponseWriter, r *http.Request) {
60      title := r.FormValue("title")
61      if title == "" {
62          http.Error(w, "Title is required", http.StatusBadRequest)
63          return
64      }
65
66      _, err := db.Exec("INSERT INTO tasks (title) VALUES ($1)", title)
67      if err != nil {
68          http.Error(w, err.Error(), http.StatusInternalServerError)
69          return
70      }
71
72      w.WriteHeader(http.StatusCreated)
73  }
74
75  func main() {
76      initDB()
77      r := mux.NewRouter()
78      r.HandleFunc("/tasks", getTasks).Methods("GET")
79      r.HandleFunc("/tasks", addTask).Methods("POST")
80
81      log.Println("Server started on :8080")
82      log.Fatal(http.ListenAndServe(":8080", r))
83  }
```

Рисунок 7

2) Создание Dockerfile (Рисунок 8).

Dockerfile > ...

```
1   FROM golang:1.22-alpine
2
3   WORKDIR /app
4
5   COPY . .
6
7   RUN go mod download
8
9   RUN go build -o todo-app .
10
11  EXPOSE 8080
12
13  CMD ["./todo-app"]
```

Рисунок 8

3) Создание файла go.mod (Рисунок 9) и определение зависимостей (Рисунок 10).

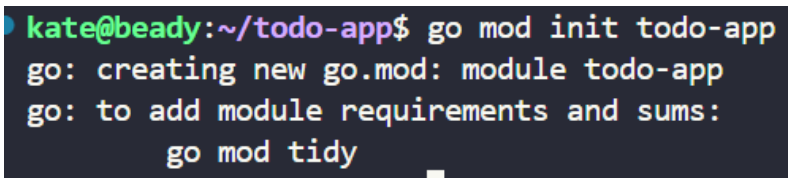


Рисунок 9

Рисунок 10

4) Сборка образа (Рисунок 11).

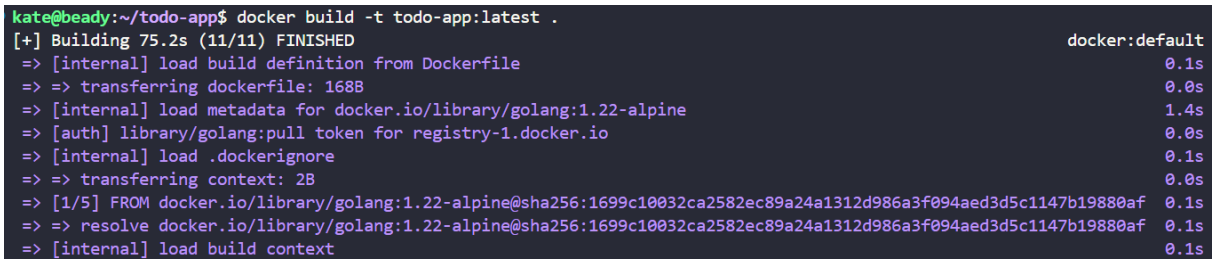


Рисунок 11

3. Развертывание PostgreSQL в Kubernetes

1) Создание файла postgres-deployment.yaml (Рисунок 12).

Рисунок 12

2) Создание postgres-service.yaml (Рисунок 13).

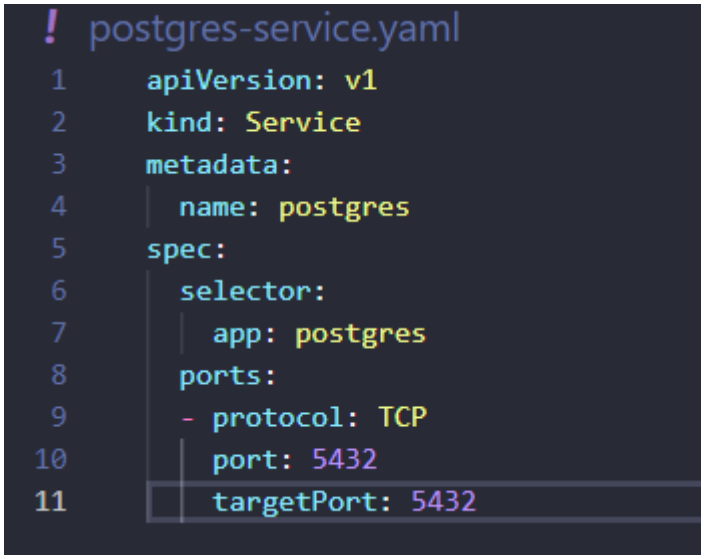


Рисунок 13

3) Применение конфигураций (Рисунок 14).

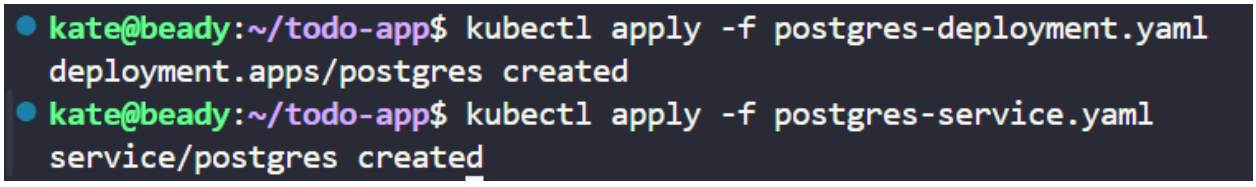


Рисунок 14

4) Проверка запуска PostgreSQL (Рисунок 15).

```
kate@beady:~/todo-app$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
postgres-cc56bd75d-q227p    1/1     Running   0          2m35s
```

Рисунок 15

4. Развертывание Golang приложения в Kubernetes

1) Создание todo-app-deployment.yaml (Рисунок 16).

todo-app-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todo-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: todo-app
  template:
    metadata:
      labels:
        app: todo-app
    spec:
      containers:
      - name: todo-app
        image: todo-app:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
        env:
        - name: POSTGRES_PASSWORD
          value: yourpassword
```

Рисунок 16

2) Создание todo-app-service.yaml (Рисунок 17).

todo-app-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: todo-app
spec:
  selector:
    app: todo-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: NodePort
```

Рисунок 17

3) Применение конфигураций (Рисунок 18).

```
kate@beady:~/todo-app$ kubectl apply -f todo-app-deployment.yaml
deployment.apps/todo-app created
kate@beady:~/todo-app$ kubectl apply -f todo-app-service.yaml
service/todo-app created
```

Рисунок 18

4) Проверка состояния пода (Рисунок 19).

```
kate@beady:~/todo-app$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
postgres-cc56bd75d-q227p    1/1     Running   0          25m
todo-app-5f88ffcfb7-v6mrd   1/1     Running   0          8s
```

Рисунок 19

5) Получение IP-адреса удалённого сервера (Рисунок 20).

```
kate@beady:~/todo-app$ kubectl get svc todo-app
NAME       TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
todo-app   NodePort   10.110.128.224   <none>        80:31062/TCP   31m
```

Рисунок 20

6) Проверка доступности (Рисунок 21).

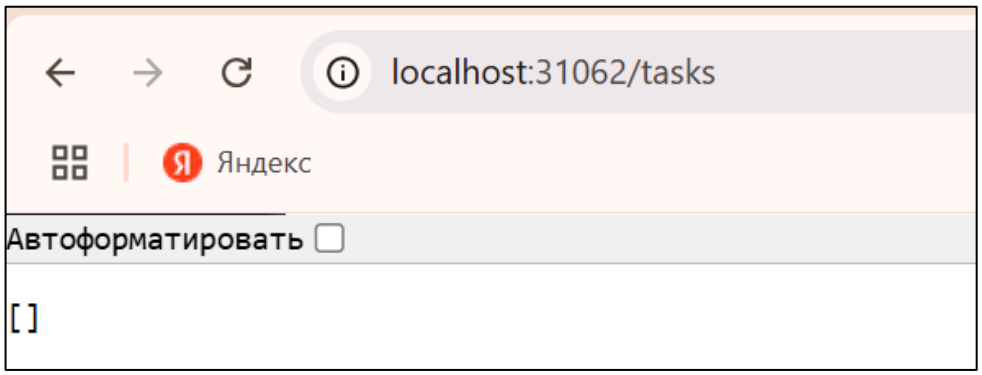


Рисунок 21

7) Запись задачи (Рисунок 22).

```
kate@beady:~/todo-app$ curl -X POST -d "title=expand k8s" http://localhost:31062/tasks
kate@beady:~/todo-app$
```

Рисунок 22

Проверка записи (Рисунок 23).

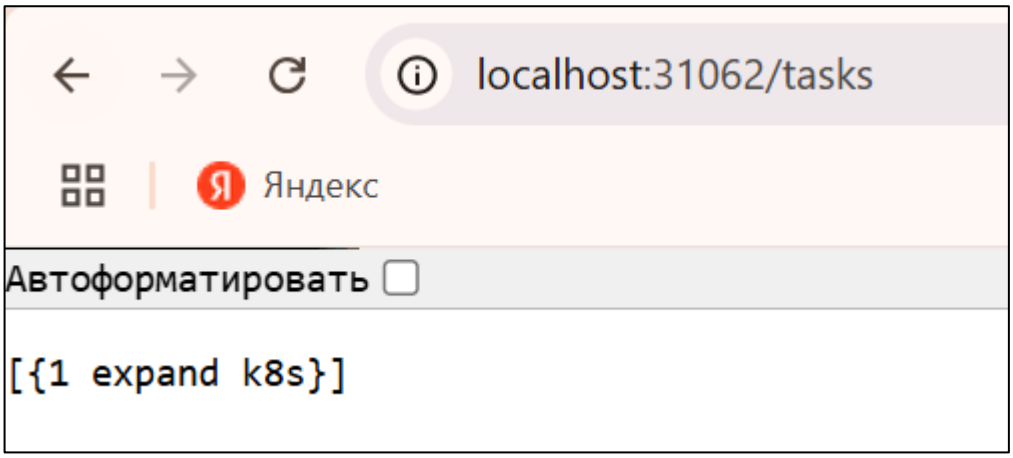


Рисунок 23

8) Подключение к PostgreSQL и проверка записей в таблице (Рисунок 24).

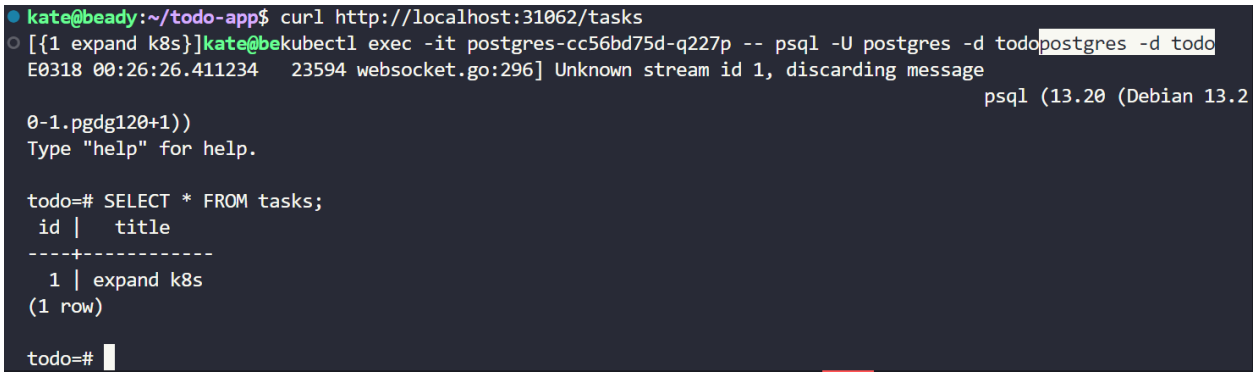


Рисунок 24

9) Добавление функций на удаление и обновление задач (Рисунок 25).

main.go

```
59    func addTask(w http.ResponseWriter, r *http.Request) {
73    }
74
75    func updateTask(w http.ResponseWriter, r *http.Request) {
76        vars := mux.Vars(r)
77        id := vars["id"]
78        title := r.FormValue("title")
79
80        _, err := db.Exec("UPDATE tasks SET title = $1 WHERE id = $2", title, id)
81        if err != nil {
82            http.Error(w, err.Error(), http.StatusInternalServerError)
83            return
84        }
85
86        w.WriteHeader(http.StatusOK)
87    }
88
89    func deleteTask(w http.ResponseWriter, r *http.Request) {
90        vars := mux.Vars(r)
91        id := vars["id"]
92
93        _, err := db.Exec("DELETE FROM tasks WHERE id = $1", id)
94        if err != nil {
95            http.Error(w, err.Error(), http.StatusInternalServerError)
96            return
97        }
98
99        w.WriteHeader(http.StatusOK)
100   }
```

Рисунок 25

10) Обновление маршутизации (Рисунок 26).

```
func main() {
    initDB()
    r := mux.NewRouter()
    r.HandleFunc("/tasks", getTasks).Methods("GET")
    r.HandleFunc("/tasks", addTask).Methods("POST")
    r.HandleFunc("/tasks/{id}", updateTask).Methods("PUT")
    r.HandleFunc("/tasks/{id}", deleteTask).Methods("DELETE")

    log.Println("Server started on :8080")
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Рисунок 26

11) После рестарта и пересборки образа выполним PUT запрос на обновление задачи (Рисунок 27).

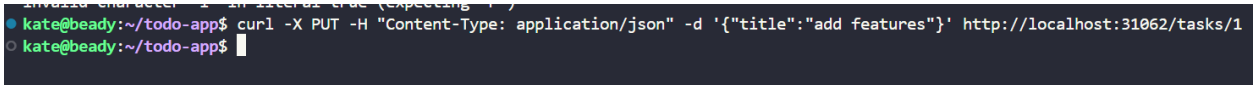


Рисунок 27

Проверка обновления записи (Рисунок 28).

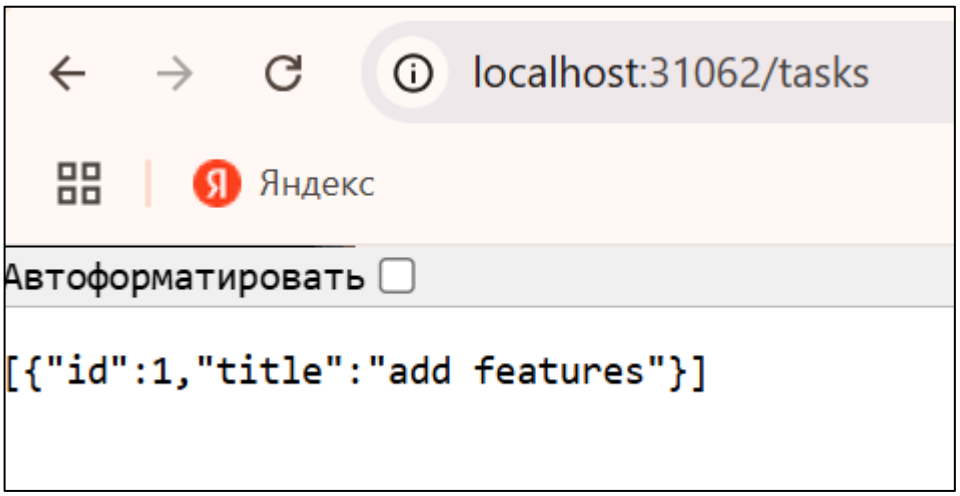


Рисунок 28

12) Удаление задачи с помощью запроса Delete (Рисунок 29).

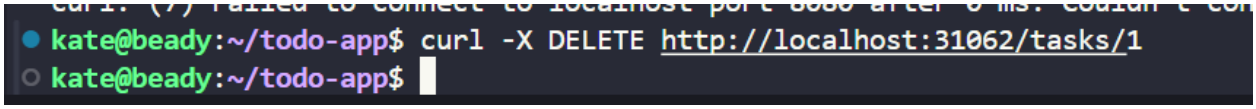


Рисунок 29

Проверка удаления (Рисунок 30).

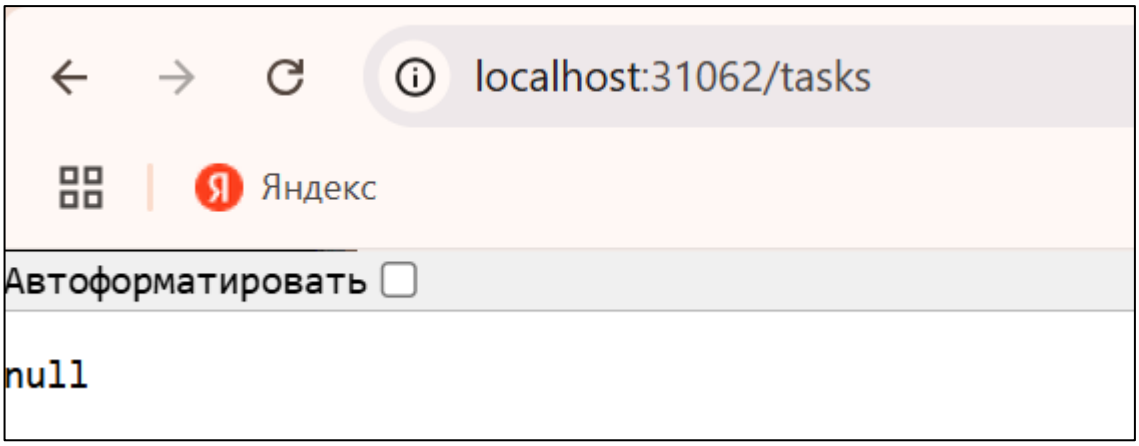


Рисунок 30

5. Очистка ресурсов (Рисунок 31).

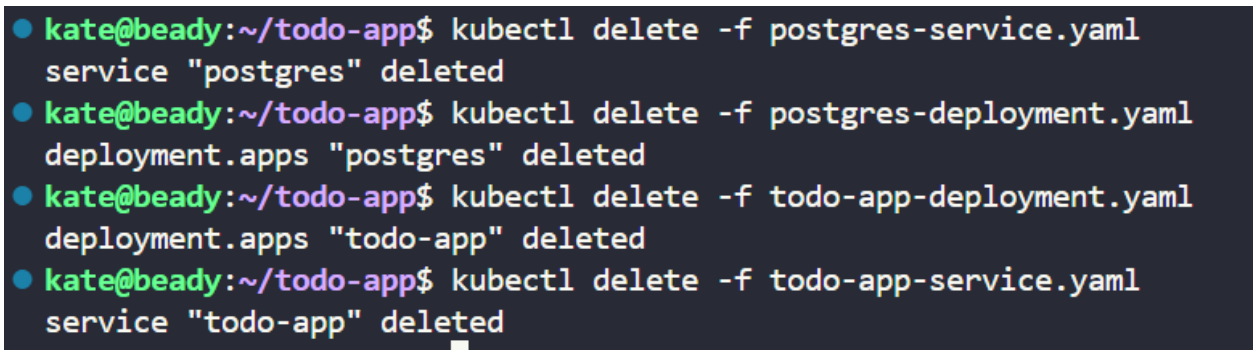

kate@beady:~/todo-app$ kubectl delete -f postgres-service.yaml
service "postgres" deleted
kate@beady:~/todo-app$ kubectl delete -f postgres-deployment.yaml
deployment.apps "postgres" deleted
kate@beady:~/todo-app$ kubectl delete -f todo-app-deployment.yaml
deployment.apps "todo-app" deleted
kate@beady:~/todo-app$ kubectl delete -f todo-app-service.yaml
service "todo-app" deleted

Рисунок 31

Выводы по работе:

В ходе выполнения лабораторной работы были успешно выполнены поставленные задачи, связанные с развертыванием приложения на Golang, использующего базу данных PostgreSQL, в среде Kubernetes. В процессе работы были освоены ключевые концепции и инструменты Kubernetes, такие как Deployment, Service, Pod, а также методы контейнеризации приложения с использованием Docker.

- Было создано приложение на Golang, предоставляющее API для управления списком задач (TODO list). Приложение поддерживает операции добавления, просмотра, обновления и удаления задач.

- Для хранения данных использовалась база данных PostgreSQL, которая также была развернута в Kubernetes.

- Созданы Deployment для Golang-приложения и PostgreSQL, что позволило управлять жизненным циклом контейнеров, обеспечивать отказоустойчивость и масштабируемость.

- Для обеспечения доступа к приложению был создан Service типа NodePort, который предоставил внешний IP-адрес для взаимодействия с API.

- Приложение было успешно протестировано: проверены все API-эндпоинты (GET, POST, PUT, DELETE), а также корректность хранения данных в PostgreSQL.

- Были выполнены операции добавления, обновления и удаления задач, что подтвердило корректность работы приложения и базы данных.

- После завершения работы все созданные ресурсы (Deployment, Service, Pods) были успешно удалены с помощью команды kubectl delete, что подтвердило понимание процесса управления ресурсами в Kubernetes.

Контрольные вопросы:

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

Pod - наименьшая и самая простая единица в Kubernetes, группа из одного или нескольких контейнеров, которые разделяют сеть и хранилище.

Deployment - управляет созданием, обновлением и масштабированием Pods.

Service - обеспечивает сетевой доступ к Pods, а также балансирует нагрузку между Pods.

2. Каково назначение Deployment в Kubernetes?

- Управляет жизненным циклом Pods (создание, обновление, масштабирование).

- Обеспечивает отказоустойчивость и обновление без downtime.

3. Каково назначение Service в Kubernetes?

- Предоставляет стабильный IP и DNS для доступа к Pods.

- Балансирует нагрузку между Pods.

- Обеспечивает сетевую связность внутри и вне кластера.

4. Как создать Deployment в Kubernetes?

Создать YAML-файл (манифест):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
```

```
    containers:
    - name: my-app
      image: my-app:latest
      ports:
      - containerPort: 8080
```

Применить деплой с помощью команды ``*kubectl apply -f deployment.yaml*``

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

Создать YAML-файл:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: LoadBalancer
```

Применить сервис с помощью команды ``*kubectl apply -f service.yaml*``

Типы Services:

- ClusterIP (по умолчанию): Внутренний IP для доступа внутри кластера.
- NodePort: Открывает порт на каждом узле кластера.
- LoadBalancer: Создает внешний балансировщик нагрузки (в облачных провайдерах).
- ExternalName: Сопоставляет Service с внешним DNS-именем.