

# GITA 5-10 კვირის დავალება.

## Task 1

### 1. პროექტის სტრუქტურა შევქმენი

პირველ რიგში, შევქმენი პროექტის დირექტორია და საჭირო ფაილები:

```
mkdir gita-final-n1
cd gita-final-n1
touch Dockerfile index.html server.py requirements.txt
```

### 2. **index.html** ფაილი დავამატე

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Welcome</title>
</head>
<body>
  <h1>Hello, Tato Zhvania</h1>
</body>
</html>
```

### 3. Flask სერვერის კოდი დავწერე **server.py** ფაილში

გამოვიყენე Flask-ის **send\_file()** მეთოდი.

```
from flask import Flask, send_file

app = Flask(__name__)

@app.route('/')
def home():
    return send_file('index.html')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

#### 4. **requirements.txt** ფაილში Flask ჩავწერე

### 5. Dockerfile შევქმენი

```
# გამოვიყენე პატარა ზომის Python image
FROM python:3.10-alpine

# სამუშაო დირექტორიის შექმნა
WORKDIR /app

# ყველა საჭირო ფაილის ერთდროულად კოპირება
COPY . /app/

# Flask-ის ინსტალაცია
RUN pip install --no-cache-dir -r requirements.txt

# პორტის გახსნა
EXPOSE 8000

# სერვერის გაშვების ბრძანება
CMD ["python", "server.py"]
```

აქ ყველა ფაილი ერთიანად დავაკოპირე `COPY ./app/`, რაც **Docker Image-ის** აგებას უფრო სწრაფს ხდის.

## 6. Docker Image ავაგე

**Docker Image** ავაგე ჩემი დოკერჰაბის უსერნიმით:

```
docker build -t tzhvania33/flask-webserver .
```

## 7. Docker Hub-ზე დავფუშე

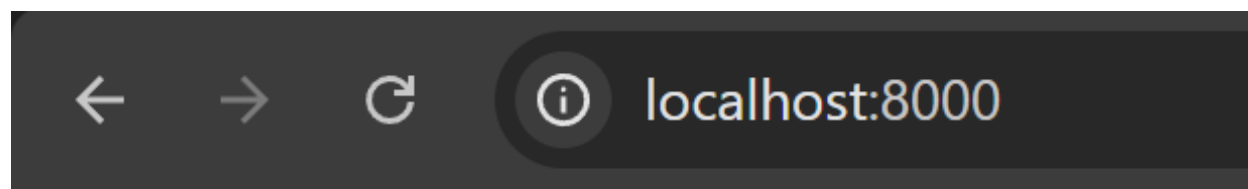
შემდეგ, **Docker Hub-ზე** ავტვირთე ჩემი იმიჯი: დალოგინებული ვიყავი უკვე და docker login ბრძანება არ დამჭირდა

```
docker push tzhvania33/flask-webserver
```

Name	Last Pushed <span>↑</span>	Contains	Visibility	Scout
tzhvania33/python-webserver	in about 4 hours	IMAGE	Public	Inactive

## 8. საბოლოო ტესტი

```
docker run -d -p 8000:8000 tzhvania33/flask-webserver
```



# Hello, Tato Zhvania

## Task 2

### 1. React პროექტის შექმნა

პირველ რიგში, React აპლიკაცია შევქმენი `create-react-app` -ით:

```
npx create-react-app my-app  
cd my-app  
npm install
```

### 2. Multi-Stage `Dockerfile` შექმნა

შემდეგ, React აპლიკაციისთვის **Multi-Stage Dockerfile** დავწერე:

```
# პირველი ეტაპი: React აპლიკაციის ბილდვა Node.js-ის გამოყენებით  
FROM node:18-alpine AS builder  
  
# სამუშაო დირექტორიის დაყენება  
WORKDIR /app  
  
# package.json და package-lock.json ფაილების კოპირება, რათა Docker c  
ache გამოვიყენოთ  
COPY package*.json ./  
  
# აუცილებელი დამოკიდებულებების (dependencies) დაყენება  
RUN npm install  
  
# სრული პროექტის კოპირება კონტეინერში  
COPY . .
```

```
# React აპლიკაციის ბილდვა  
RUN npm run build
```

```
# მეორე ეტაპი: NGINX სერვერის გამოყენება React აპლიკაციის გასაშვებად  
FROM nginx:latest
```

```
# სამუშაო დირექტორიის დაყენება NGINX-ში  
WORKDIR /usr/share/nginx/html
```

```
# გაშვებისთვის მზა React აპლიკაციის ფაილების კოპირება NGINX-ის კონტეინერში  
COPY --from=builder /app/build .
```

```
# NGINX-ის გაშვება foreground რეჟიმში, რათა კონტეინერი არ გაჩერდეს  
CMD ["nginx", "-g", "daemon off;"]
```

### 3. Docker Image-ის აგება

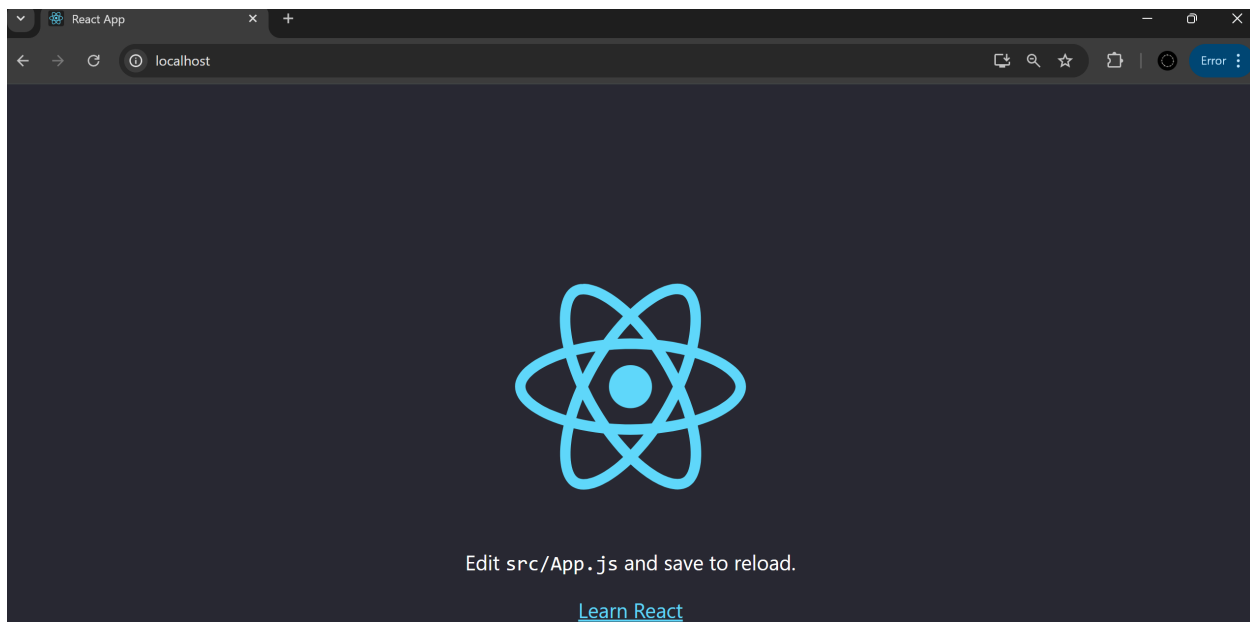
შემდეგ, **Docker Image** ავაგე:

```
docker build -t react-nginx-app .
```

### 4. Docker კონტეინერის გაშვება

React აპლიკაციის **Docker** კონტეინერი გავუშვი:

```
docker run -d -p 80:80 react-nginx-app
```



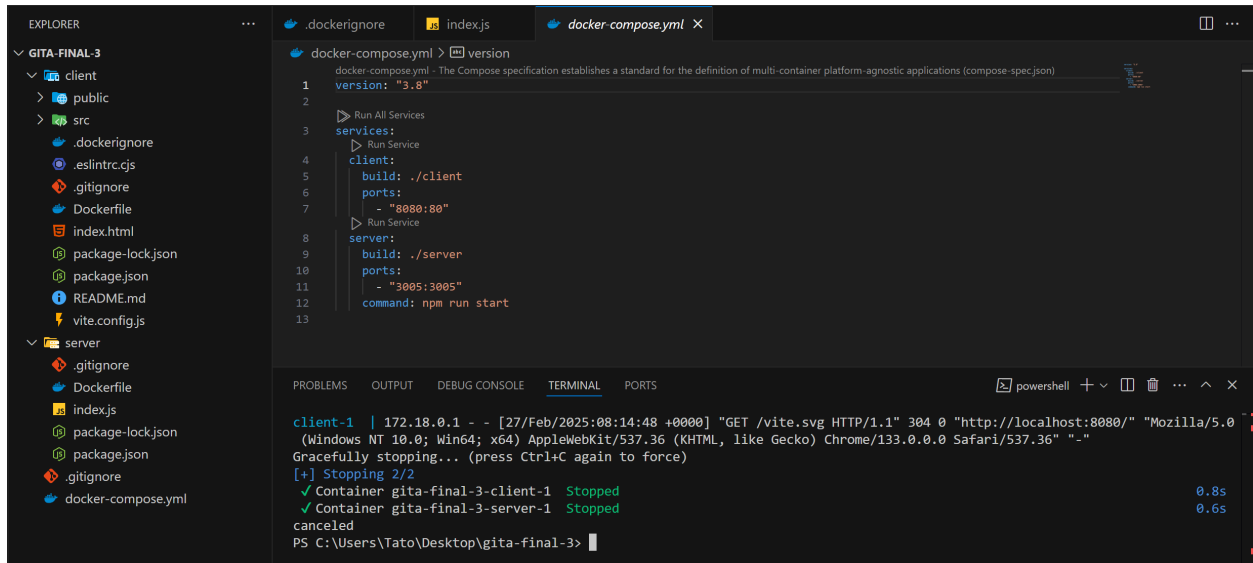
## Task 3

### 1. React პროექტის შექმნა

პირველ რიგში, **React** აპლიკაცია შევქმენი `create-react-app` -ით:

```
npm create vite@latest  
npm install
```

**File Structure:**



## 2. Server **Dockerfile** -ის შექმნა

სერვერ დოკერფაილი ცალკე დავწერე:

```
# ვიყენებ Node.js 18-ის ალპაინ ვერსიას, რათა კონტეინერი იყოს მსუბუქი და სწრაფი
FROM node:18-alpine
```

```
# ვქმნი სამუშაო დირექტორიას '/app', სადაც იმუშავებს ჩემი აპლიკაცია
WORKDIR /app
```

```
# ვაკოპირებ 'package.json' და 'package-lock.json' ფაილებს
COPY package*.json .
```

```
# ვამუშავებ 'npm install', რათა ყველა საჭირო პაკეტი ჩაიტვირთოს
RUN npm install
```

```
# ვაკოპირებ ყველა ფაილს სამუშაო დირექტორიაში, რათა ჩემი აპლიკაცია სრულად ჩაიტვირთოს
COPY . .
```

### 3. Client **Dockerfile** -ის შექმნა

კლიენტ დოკერფაილი ცალკე დავწერე:

```
# პირველ სტეიჯში ვიყენებ Node.js 18-ის ალპაინ ვერსიას, რათა კონტეინერი იყოს მსუბუქი და სწრაფი
FROM node:18-alpine as builder

# ვქმნი სამუშაო დირექტორიას `/app`, სადაც ჩემი აპლიკაცია იმუშავებს
WORKDIR /app

# ვაკოპირებ `package.json` და `package-lock.json` ფაილებს

# ვამუშავებ `npm install`, რათა ყველა საჭირო პაკეტი ჩაიტვირთოს
RUN npm install

# ვაკოპირებ ყველა ფაილს სამუშაო დირექტორიაში, რათა პროექტის კოდი სრულად ჩაიტვირთოს
COPY . .

# ვამუშავებ `npm run build`, რათა შევექმნა კომპილირებული/ოპტიმიზირებული ფაილები
RUN npm run build

# ვიწყებ ახალი სტეიჯიდან, სადაც ვიყენებ Nginx-ს როგორც სერვერს
FROM nginx

# ვაკოპირებ მხოლოდ `dist` დირექტორიაში არსებულ გაშვებისათვის მზად ფაილებს Nginx-ის სტანდარტულ დირექტორიაში
COPY --from=builder /app/dist /usr/share/nginx/html

# ვუშვებ Nginx-ს წინა პლანზე (foreground mode), რათა კონტეინერმა არ შეწყვიტოს მუშაობა
CMD ["nginx", "-g", "daemon off;"]
```



## 4. docker-compose.yml ფაილი

```
version: "3.8"

services:
  client:
    build: ./client
    ports:
      - "8080:80"
  server:
    build: ./server
    ports:
      - "3005:3005"
    command: npm run start
```

ეს `docker-compose.yml` ფაილი აღწერს ორი სერვისის კონტეინერიზაციას:

### 1. `client` (Front-End სერვისი)

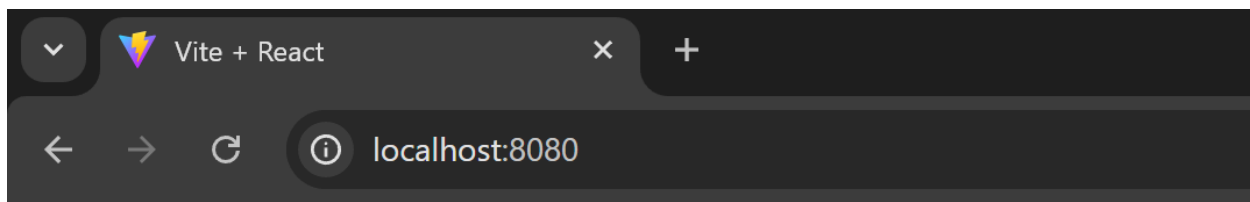
- იკომპილირება `./client` დირექტორიაში არსებული `Dockerfile` იდან.
- იყენებს პორტს **8080**

### 2. `server` (Back-End სერვისი)

- იკომპილირება `./server` დირექტორიაში არსებული `Dockerfile` იდან.
- იყენებს პორტს **3005**, რომელიც რჩება იგივე (3005:3005).
- კონტეინერში გაშვების შემდეგ, `npm run start` ბრძანებით იწყებს სერვერის გაშვებას.

შედეგი:

- **Front-End** ( `client` ) მუშაობს `http://localhost:8080`-ზე.
- **Back-End** ( `server` ) მუშაობს `http://localhost:3005`-ზე.



count is 0!!

Tato  
Zhvania

## 5. Docker Image-ის აგება და გაშვება

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

### Task 4

#### 1. Namespace-ის შექმნა

პირველ რიგში, შევქმენი **prod namespace**, რათა Deployment და Service ამ Namespaced-ში განთავსებულიყო. ამისთვის დავწერე შემდეგი YAML ფაილი:

namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod
```

#### 2. Deployment-ის შექმნა

შემდეგ, შევქმენი **Deployment**, რომელიც:

- იყენებს `modem7/docker-rickroll:latest` Docker იმიჯს.
- აქვს ორი რეპლიკა (**replicas: 2**), რათა აპლიკაცია იყოს ხელმისაწვდომი.
- Pod იმუშავებს **პორტ 80-ზე**.

`web-app-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
  namespace: prod
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
        - name: web-app
          image: modem7/docker-rickroll:latest
          ports:
            - containerPort: 80
```

### 3. Service-ის შექმნა

შემდეგ შევქმენი **Service**, რომელიც:

- მიიღებს ტრაფიკს **პორტ 8080-ზე**.
- გადაამისამართებს **Pod-ის პორტ 80-ზე**.

- იყენებს **NodePort** ტიპის სერვისს, რათა ის ხელმისაწვდომი იყოს კლასტერს გარეთაც.

web-app-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: web-app-service
  namespace: prod
spec:
  type: NodePort
  selector:
    app: web-app
  ports:
    - protocol: TCP
      port: 8080    # კლასტერში შესული ტრაფიკის პორტი
      targetPort: 80 # Pod-ის შიდა პორტი
      nodePort: 30080 # კლასტერის გარეთ მისაწვდომი პორტი (Kind ან Minikube-სთვის)
```

#### 4. yml ფაილების apply

```
kubectl apply -f namespace.yaml web-app-deployment.yaml web-app-service.yaml
```

#### Pods-ის სტატუსი:

```
kubectl get pods --namespace=prod
```

```
PS C:\Users\Tato\Desktop\gita-final-n4> kubectl get pods --namespace=prod
NAME                                READY   STATUS    RESTARTS   AGE
web-app-8f86597d8-778mz            1/1     Running   0           8m25s
web-app-8f86597d8-kglxr            1/1     Running   0           8m25s
PS C:\Users\Tato\Desktop\gita-final-n4>
```

## Task 5

### 1. შექმენი DaemonSet - daemonset.yaml

რომელიც გაშვებულია ყველა ნოდზე და შეიცავს ორ კონტეინერს:

- `init-container`, რომელიც იყენებს `busybox` image-ს. შეასრულებს `date` ბრძანებას და შედეგს ჩაწერს `/tmp/date.log` ფაილში.
- `main-container`, რომელიც ასევე `busybox` ს იყენებს და ამოწმებს, არსებობს თუ არა ეს ფაილი.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: busybox-daemonset
  labels:
    app: busybox
spec:
  selector:
    matchLabels:
      app: busybox
  template:
    metadata:
      labels:
        app: busybox
    spec:
      volumes:
        - name: shared-volume
          emptyDir: {}

      initContainers:
        - name: init-container
```

```
image: busybox
command: ["/bin/sh", "-c"]
args:
  - "date > /tmp/date.log && cat /tmp/date.log"
volumeMounts:
  - name: shared-volume
    mountPath: /tmp
```

```
containers:
  - name: main-container
    image: busybox
    command: ["/bin/sh", "-c"]
    args:
      - "if [ -f /tmp/date.log ]; then echo 'File exists!'; cat /tmp/date.log; else
echo 'File not found!'; fi"
    volumeMounts:
      - name: shared-volume
        mountPath: /tmp
```

- **ვოლუმის გაზიარება დავამატე**, რათა `init-container` -ში ჩანერილი `/tmp/date.log` ფაილი ხელმისაწვდომი ყოფილიყო მთავარ კონტეინერში.
  - ამისთვის გამოვიყენე `emptyDir` **ვოლუმი**, რომელიც დროებით ინახავს მონაცემებს.
  - `init-container` ში ეს ვოლუმი **დამაუნთდა** `/tmp` **დირექტორიაზე**, სადაც შეიქმნა `date.log` ფაილი.
  - იგივე ვოლუმი **მთავარ კონტეინერშიც** დამაუნთდა `/tmp` **დირექტორიაზე**.

## 2. bash სკრიპტი

1. `if [ -f /tmp/date.log ]; then`
  - ამოწმებს, არსებობს თუ არა `/tmp/date.log` ფაილი.

- `f` ნიშნავს, რომ უნდა შეამოწმოს, არის თუ არა ეს ფაილი ნამდვილი (regular file), ანუ არა დირექტორია.

2. `echo 'File exists!';`

- თუ ფაილი არსებობს, ეკრანზე გამოაქვს **"File exists!"**.

3. `cat /tmp/date.log;`

- იმავე შემთხვევაში, ფაილის შიგთავსს ბეჭდავს (`cat` ბრძანებით).

4. `else echo 'File not found!';`

- თუ ფაილი არ არსებობს, ეკრანზე გამოაქვს **"File not found!"**.

5. `fi`

- `if` პირობის დახურვა.

როგორ მუშაობს?

✓ თუ `/tmp/date.log` არსებობს, გამოიტანს: File exists!

✗

თუ `/tmp/date.log` არ არსებობს, გამოიტანს: File not found!

### 3. YAML ფაილი-ის გაშვება და შემოწმება:

```
kubectl apply -f daemonset.yaml
```

```
Tato@z MINGW64 ~/Desktop/gita-final-n5
$ kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
busybox-daemonset-mzpjn            0/1     CrashLoopBackOff    7 (95s ago) 12m
```


```
kubectl logs -l app=busybox -c main-container
```

```
Tato@z MINGW64 ~/Desktop/gita-final-n5
$ kubectl logs -l app=busybox -c main-container
File exists!
Thu Feb 27 14:19:47 UTC 2025
```

## Task 6

### 1. Secret შევქმენი, რომელიც შეიცავს GITA=Secret მნიშვნელობას

YAML ფაილში დავწერე Secret-ის კონფიგურაცია ( `secret.yaml` ):

 შენიშვნა: `Secret` -ის მნიშვნელობა ( `Secret` ) უნდა იყოს **Base64-ით დაშიფრული**. შეგიძლიათ ეს გააკეთოთ Linux ტერმინალში:

```
echo -n "Secret" | base64
```

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  GITA: U2VjcmV0 # Base64-ით დაშიფრული "Secret"
```

შემდეგ გავუშვი:

```
kubectl apply -f secret.yaml
```

ამით `my-secret` Secret შევქმენი Kubernetes-ში.

### 2. Pod-ის შექმნა.

შევქმენი, რომელიც ამ Secret-ს იყენებს როგორც გარემოს ცვლადს



`pod-with-secret.yaml` ფაილში დავწერ

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  containers:
  - name: secret-container
    image: busybox
    command: ["sh", "-c", "echo $GITA && sleep 3600"]
    env:
    - name: GITA
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: GITA
```

შემდეგ გავუშვი:

```
kubectl apply -f pod-with-secret.yaml
```

Secret-ის მნიშვნელობის გამოტანა:

```
kubectl exec secret-pod -- sh -c 'echo $GITA'
```

```
smonitoring@C0257 MINGW64 ~/Desktop/tato
$ kubectl exec secret-pod -- sh -c 'echo $GITA'
Secret
```

## Task 7

### 1 ConfigMap-ის შექმნა

კპირველ რიგში, ონფიგურაცია ჩავწერე Kubernetes-ის `ConfigMap` რესურსში, სახელად `nginx-config`, რომელიც შეიცავს `nginx.conf` ფაილის შინაარსს.

#### nginx-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    server {
      listen 80;
      server_name localhost;

      location / {
        root /usr/share/nginx/html;
        index index.html;
      }
    }
```

კომანდით ConfigMap-ის შექმნა:

```
kubectl apply -f nginx-configmap.yaml
```

```
Tato@Z MINGW64 ~/Desktop/gita-final-n7
$ ls
nginx-configmap.yaml  nginx-pod.yaml

Tato@Z MINGW64 ~/Desktop/gita-final-n7
$ kubectl apply -f nginx-configmap.yaml
configmap/nginx-config created
```

## 2 Pod-ის კონფიგურაცია

შემდეგი ნაბიჯი იყო Nginx პოდის შექმნა, რომელიც გამოიყენებს `nginx-config` ConfigMap-ს.

### nginx-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - name: config-volume
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
  volumes:
    - name: config-volume
      configMap:
        name: nginx-config
```

**პოდის გაშვება Kubernetes-ში:**

```
kubectl apply -f nginx-pod.yaml
```

```
Tato@z MINGW64 ~/Desktop/gita-final-n7
$ kubectl apply -f nginx-pod.yaml
pod/nginx-pod created
```

### 3 შემოწმება

```
Tato@z MINGW64 ~/Desktop/gita-final-n7
$ kubectl get pods
NAME                READY   STATUS              RESTARTS   AGE
busybox-daemonset-p4j2t 0/1     CrashLoopBackOff    6 (3m30s ago) 9m43s
nginx-pod            1/1     Running             0           4m36s

Tato@z MINGW64 ~/Desktop/gita-final-n7
$ kubectl get configmap
NAME          DATA   AGE
kube-root-ca.crt 1       4d
nginx-config    1       5m4s
```

## Task 8

### 1 Deployment კონფიგურაციის შექმნა

პირველ რიგში, შევქმენი Kubernetes-ის **Deployment** რესურსის YAML ფაილი.

შევარჩიე **nginx** image

Deployment-ს დავუწერე **2 რეპლიკა** და მივუთითე რესურსების ლიმიტები:

- **Requests:**
  - CPU: **250m** (0.25 core)

- Memory: 256Mi (256MB RAM)
- **Limits:**
  - CPU: 500m (0.5 core)
  - Memory: 512Mi (512MB RAM)

ამ მონაცემებით შევქმენი `nginx-deployment.yaml` ფაილი:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: nginx
          resources:
            requests:
              cpu: "250m"
              memory: "256Mi"
            limits:
              cpu: "500m"
              memory: "512Mi"
```

## 2 Deployment-ის გაშვება და შემოწმება.

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

```
Tato@z MINGW64 ~/Desktop/gita-final-n8
$ ls

Tato@z MINGW64 ~/Desktop/gita-final-n8
$ vim nginx-deployment.yaml

Tato@z MINGW64 ~/Desktop/gita-final-n8
$ kubectl apply -f nginx-deployment.yaml
deployment.apps/my-deployment created

Tato@z MINGW64 ~/Desktop/gita-final-n8
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
my-deployment-5499565544-65vtr	1/1	Running	0	7s
my-deployment-5499565544-j46mh	1/1	Running	0	7s