Semesterarbeit Aufgabe 2

Matthias Heimberg

Aufgabe 2: Arbeiten mit Jenkins

Im Folgenden werden die Schritte zu Aufgabe 2 dokumentiert:

Manueller Build der App als Docker Image

Die App wird mittels Gradle gebaut. Dazu wird im Terminal folgender Befehl ausgeführt:

```
./gradlew build
```

Zunächst war geplant den Build innerhalb eines Docker Containers durchzuführen und dann das Docker Image mittels eines Dockerfiles zu bauen. Probleme mit Dependencies und dem Gradle Wrapper haben dies jedoch verhindert. Als Alternative wurde deshalb Google Jib (https://cloud. google.com/java/getting-started/jib) verwendet. Dieses Tool erlaubt es Docker Images direkt aus Gradle zu bauen. Dazu wurde das Plugin `com.google.cloud.tools.jib` in der `build.gradle` hinzugefügt. Anschliessend wurde das Docker Image mittels

```
./gradlew jib
```

gebaut und in das Docker Hub Repository `heimberg/devops` gepusht. Damit der Push funktioniert, muss der Docker Hub Token in den Jenkins Credentials hinterlegt werden. Später stellte sich heraus, dass das Docker Repository nicht für das geplante Deployment auf Google Cloud Platform geeignet ist. Deshalb wurde das Docker Image schlussendlich auf die Google Container Registry gepusht. Dazu musste das `build. gradle` angepasst werden. Die Änderungen sind im folgenden Code Snippet zu sehen:

```
jib {
    to {
        image = "gcr.io/devops/devops"
        tags = [version, 'latest']
    }
}
```

Damit das Image auf die Google Container Registry gepusht werden kann, muss gemäss https://cloud.google.com/java/getting-started/jib mittels Google Cloud CLI ein Login durchgeführt werden:

```
gcloud auth configure-docker
```

Dies reicht aber offenbar nicht aus. Schlussendlich funktionierte der Push mit Hilfe von `docker-credential-gcr`, welcher mit

```
gcloud components install docker-credential-gcr
```

lokal installiert wurde. Anschliessend wurde ein Login mit

```
docker-credential-gcr gcr-login
```

durchgeführt. Im `build.gradle` wurde dann noch folgender Code für die Konfiguration von `jib`hinzugefügt:

```
jib {
    from {
        image = 'openjdk:17-alpine'
    }
    to { image = 'gcr.io/cellular-syntax-231507/devops'
        tags = [version, 'latest']
        credHelper = 'gcloud'

    }
    container {
        mainClass = 'app.App'
        ports = ['7000']
    }

}
```

Das Image basiert nun auf `openjdk:14-alpine` und wird auf die Google Container Registry gepusht (dazu muss die Container Registry zunächst aktiviert werden, vgl. dazu https://cloud.google.com/container-registry/docs/enable-service?hl=de). Die Tags

sind die Versionsnummer und `latest`. Der CredHelper ist `gcloud`, damit werden die Credentials aus der Google Cloud CLI angefordert. Das Image wird mit der Main Class `app.App` gestartet und der Port `7000` wird freigegeben. Der Befehl

```
./gradlew jib
```

baut schliesslich das Image und pusht es auf die Google Container Registry. Dort steht das Image für das anschliessende Deployment zur Verfügung.

Manuelles Deployment der App

Das Deployment der App wird auf Google Cloud Run durchgeführt. Dies wurde nach dem erfolgreichen Push des Docker Images zunächst manuell mittels der Google Cloud CLI ausgeführt:

```
gcloud run deploy devops \
  --image gcr.io/cellular-syntax-231507/devops \
  --platform managed \
  --region europe-west4 \
  --allow-unauthenticated \
  --port 7000
```

Dies führt aber zu einem Fehler auf Google Cloud Run (`The user-provided container failed to start and listen on the port defined provided by the PORT=7000 environment variable`). Daher wurde der Container zunächst lokal gestartet:

```
docker run -p 7000:7000 gcr.io/cellular-syntax-231507/devops
```

Dabei stellte sich heraus, dass die ursprünglich im Container verwendete Java Runtime (`openjdk:14-alpine`) nicht mit der Java Runtime auf dem Host übereinstimmt. Deshalb wurde das Docker Image mit der Java Runtime `openjdk:17-alpine` neu gebaut und in das Registry gepusht. Das Deployment auf Google Cloud Run funktioniert nun. Nun können die manuell durchgeführten Schritte in Jenkins automatisiert werden, was in den folgenden Abschnitten beschrieben wird.

Jenkins

Jenkins wurde lokal als Docker Container mittels Docker-Compose ausgeführt. Grund dazu ist die einfachere Handhabung des Containers, dieser kann so mittels `docker-compose up -d` mit den notwendigen Konfigurationen gestartet werden. Ausserdem erlaubt Docker-Compose die Verwendung von relativen Pfadnamen für die Volumes. Dazu wurde die `docker-compose.yml` wie folgt erstellt:

```yaml
version: "3.8"
services:
  jenkins:
    image: jenkins/jenkins:lts
    container_name: jenkins
    ports:
      - "8080:8080"
      - "50000:50000"
    volumes:
      - ./jenkins-data:/var/jenkins_home
    restart: always
```

Das Webinterface ist danach unter `localhost:8080/` erreichbar. Sämtliche Konfigurationen werden persistiert, da der Ordner `jenkins_home` auf dem Host gemountet wird. Das Passwort für den Admin User kann bei laufendem Container mittels

```
docker exec jenkins-local cat /var/jenkins_home/secrets/initialAdminPassword
```

ausgelesen werden. Danach wird die Installation mit den default Plugins und dem User `matthias` abgeschlossen.

Konfiguration von Jenkins

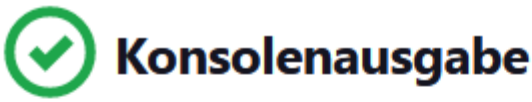
Zunächst wird die Authentifizierung an Google Cloud Platform konfiguriert. Dazu muss zunächst die Google Cloud CLI im Jenkins Container gemäss der Anleitung von https://cloud.google.com/sdk/docs/install#deb installiert werden. Dies muss mittels Bash im Container durchgeführt werden, dazu muss die Shell als root gestartet werden:

```
docker exec -it jenkins /bin/bash
```

Mittels der folgenden kleinen Pipeline wird geprüft, ob die Google Cloud CLI korrekt installiert wurde:

```
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                sh 'gcloud version'
            }
        }
    }
}
```

Die Ausgabe:




```
Started by user Matthias Heimberg
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/devops-test
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] sh
+ gcloud version
Google Cloud SDK 405.0.0
alpha 2022.09.30
beta 2022.09.30
bq 2.0.78
bundled-python3-unix 3.9.12
core 2022.09.30
gcloud-crc32c 1.0.0
gsutil 5.14
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Als nächstes muss ein Service Account für Google Cloud Run erstellt werden. Dazu wird über IAM & Admin -> Service Accounts -> Create Service Account ein neuer Service Account erstellt. Der Service Account wird `jenkins-gcloud` genannt. Der Service Account Key wird anschliessend als JSON Key heruntergeladen und im Jenkins Container als Secret hinterlegt. Dazu wird über Credentials -> System -> Global credentials (unrestricted) -> Add Credentials ein neues Secret vom Typ `Secret File` mit Hilfe des JSON Key erstellt. In der Pipeline muss die GCP Projekt-ID `cellular-syntax-231507` mittels

Variable `CLOUDSDK_CORE_PROJECT` und das zu verwendende Secret gesetzt werden. Die Pipeline zum Testen der Credentials sieht wie folgt aus (listet die verfügbaren Zonen auf, dazu muss eine entsprechende Berechtigung vorhanden sein):

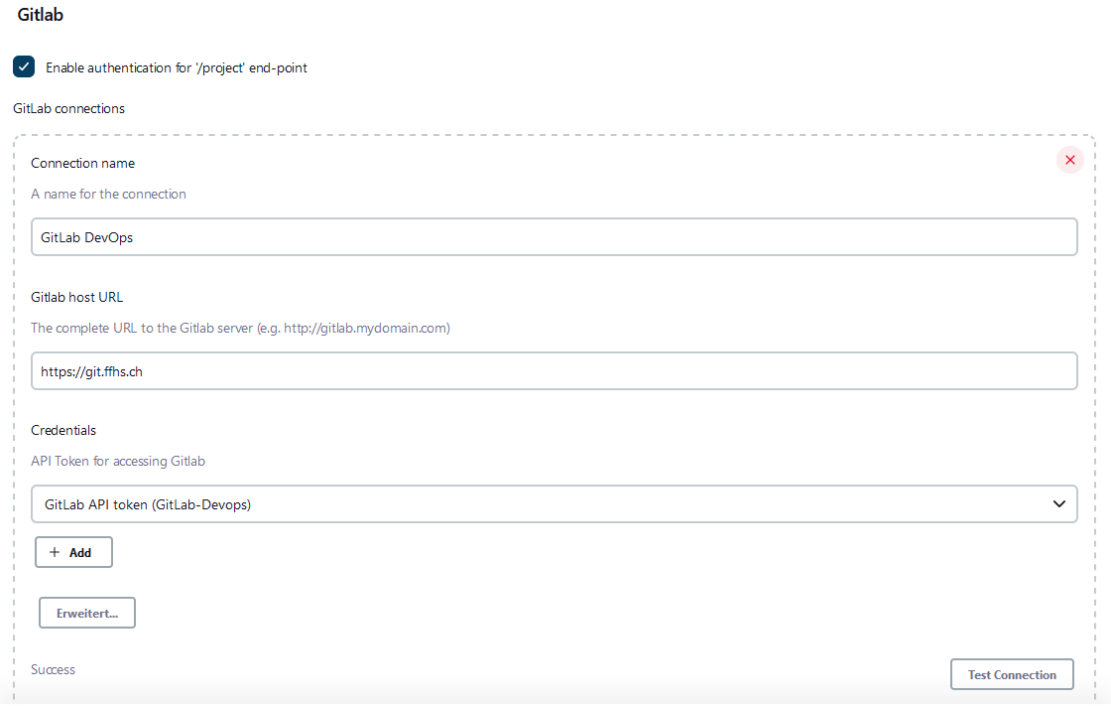
```
pipeline {
  agent any
  environment {
    CLOUDSDK_CORE_PROJECT='cellular-syntax-231507'
  }
  stages {
    stage('test') {
      steps {
        withCredentials([file(credentialsId: 'gcloud', variable: 'GCLOUD')]) {
          sh '''
            gcloud version
            gcloud auth activate-service-account --key-file="$GCLOUD"
            gcloud compute zones list
          '''
        }
      }
    }
  }
}
```

Der Test zeigte, dass die Google Compute Engine API nicht aktiviert ist. Nach Aktivierung der API funktioniert der Test erfolgreich.

Jenkins Pipeline

Nach erfolgreicher Konfiguration von Jenkins wird die Pipeline aufgesetzt. Die Pipeline soll die folgenden Punkte erfüllen: - Der Build soll bei jedem Push auf den Develop-Branch ausgelöst werden - Der Build soll das Docker Image bauen und in die Google Container Registry pushen - Der Build soll das Docker Image auf Google Cloud Run deployen Das dazu erforderliche Jenkinsfile wird im Root-Verzeichnis des Repositories (Branch `develop`) erstellt und in Jenkins (Multibranch pipeline) vom Git Repository https://git.ffhs.ch/matthias.heimberg/devops eingebunden. Die für die Verbindung zu GitLab erforderlichen Credentials werden in Jenkins hinterlegt (https://docs.gitlab.com/ee/integration/jenkins.html).

Jeder Push auf den Develop-Branch löst nun einen Build aus. Dazu wird das GitLab Plugin installiert, welches die GitLab API verwendet. Das dafür erforderliche Access Token wird in Jenkins hinterlegt:



GitLab könnte nun Jenkins per Webhook über die API benachrichtigen, wenn ein neuer Build gestartet werden soll, da Jenkins aber nur lokal ausgeführt wird, kann diese Funktionalität nicht verwendet werden.

Checkout des Repositories Die folgende Stage sorgt dafür, dass das Repository von GitLab (im Moment nur der `develop` Branch) ausgecheckt wird:

```
stage('checkout from GitLab') {
            steps {
                withCredentials([usernamePassword(credentialsId: 'gitlab-access', usernar
                    git url: 'https://git.ffhs.ch/matthias.heimberg/devops.git', branch:
                }
            }
        }
```

Die dazu notwendigen Credentials werden in Jenkins hinterlegt.

Code Quality Check / Quality Gate Als nächstes wurde die Stage `check code quality`konfiguriert. Voraussetzung ist die Installation des SonaQube Scanner Plugins in Jenkins. In der SonarQube WebUI wird ein Webhook für Jenkins erstellt. Die URL

lautet `http://jenkins:8080/sonarqube-webhook/`. Damit kann Jenkins die Ergebnisse von SonarQube abrufen, was in der Stage `quality gate` realisiert wird. Diese bricht die Pipeline ab, wenn die Qualität des Codes nicht den Anforderungen gemäss Quality Gate in Sonarqube entspricht.

Die beiden Stages sehen wie folgt aus:

```
stage('check code quality') {
            steps {
                withSonarQubeEnv('SonarQube') {
                    sh 'chmod +x ./gradlew'
                    sh './gradlew sonarqube -D"sonar.projectKey=DevOps"'
                }
            }
        }
        stage('quality gate') {
            steps {
                waitForQualityGate abortPipeline: true
            }
        }
```

Build der Applikation Die Stage `build` baut die Applikation mittels Gradle. Die Stage sieht wie folgt aus:

```
stage('build') {
            steps {
                sh './gradlew build'
            }
        }
```

Build des Docker Images / Push in die Google Container Registry Die Stage `create docker image and push to registry` baut das Docker Image und pusht es in die Google Container Registry. Die Stage sieht wie folgt aus:

```
stage('create docker image and push to registry') {
            steps {
                withCredentials([file(credentialsId: 'gcloud', variable: 'GCLOUD')]) {
                    sh '''
                        gcloud auth activate-service-account --key-file="$GCLOUD"
                        ./gradlew jib
                    '''
```

```
            }
        }
    }
```

Vor dem Ausführen des Gradle Tasks `jib` wird der Service Account aktiviert. Der Service Account ist in der Google Cloud Platform erstellt worden und hat die Berechtigung, Docker Images in die Google Container Registry zu pushen. Die Credentials für den Service Account werden in Jenkins hinterlegt.

Deployment auf Google Cloud Run Die Stage `deploy to cloud run` deployt das Docker Image auf Google Cloud Run. Die Stage sieht wie folgt aus:

```
steps {
            withCredentials([file(credentialsId: 'gcloudcompute', variable: 'GCLOUDC
                sh '''
                    gcloud auth activate-service-account --key-file="$GCLOUDCOMPUTE"
                    gcloud run deploy devops --image gcr.io/cellular-syntax-231507/de

                '''
            }
        }
```

Der dazu notwendige Service Account besitzt die Berechtigung, Docker Images auf Google Cloud Run zu deployen. Die Verwendung des Service Accounts muss im Befehl `gcloud run deploy` explizit angegeben werden, ansonsten wird der Default Service Account verwendet, welcher nicht die nötigen Berechtigungen hat. Die App ist nach erfolgreichem Deployment unter https://devops-d4bqj7s2iq-ez.a.run.app erreichbar.

Post Actions Nach den Stages werden die Post Actions ausgeführt.

Damit Jenkins das Ergebnis der Pipeline per Mail an den Entwickler senden kann, muss

Im Google Account wird ein App Password für den Jenkins User erstellt. Dieses wird in den Jenkins Credentials hinterlegt.

Probleme und deren Lösung

- Docker Image aus Build der Applikation erstellen (Dependencies) -> Lösung: Image mittels `Jib` erstellen.
- Docker Image auf Google Container Registry pushen -> Lösung: `docker-credential-gcr` installieren und Login mittels `docker-credential-gcr gcr-login` durchführen.
- Google Cloud Run führt das Image wegen Port 7000 nicht aus -> Lösung: Port 7000 freigeben.
- Docker lässt kein Mounten von relativen Pfaden zu -> Lösung: Docker-Compose verwenden, diese Lösung vereinfacht gleichzeitig das Starten von Jenkins.
- Permission denied beim Ausführen von `gradlew` -> Lösung: `gradlew` mittels `chmod +x gradlew` ausführbar machen.
- Sonarqube kann nicht erreicht werden -> Lösung: Hostname wird in `docker-compose.yml` auf `sonarqube` gesetzt, damit die Container sich gegenseitig finden.
- Deployment auf Google Cloud Run failed immer mit einem Permission Denied obwohl der Service Account die Berechtigung hat -> Lösung: Flag `--service-account` beim Deployment hinzufügen, ansonsten wird der Default Service Account verwendet.