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 (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, da Google Cloud Run nur Images aus der Google Container Registry akzeptiert. 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 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
```

und dem verwendeten Google Account durchgeführt. Im `build.gradle` wurde anschliessend 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` (`alpine` besitzt eine geringere Base Grösse als das Default Image) und wird auf die Google Container Registry gepusht (dazu muss die Container Registry zunächst aktiviert werden, vgl. dazu

cloud.google.com/container-registry/docs/enable-service). Die Tags sind die Versionsnummer und `latest`. Der `CredHelper` (also der verwendete Credentials Helper) ist `gcloud`, damit werden die Credentials aus dem Credentials Store der Google Cloud CLI angefordert. Das Image wird mit der Main Class `app.App` gestartet und der Port `7000` wird freigegeben. Mittels des Befehls

```
./gradlew jib
```

kann das Image gebaut und auf die Google Container Registry gepusht werden. 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 erstmaligem 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 führte jedoch 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 testhalber zunächst lokal gestartet um allfällige Fehler im Image zu finden:

```
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 für das Projekt konfigurierten Java Runtime 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 manuell. Anschliessend wurden die manuell durchgeführten Schritte in Jenkins automatisiert, was in den folgenden Abschnitten beschrieben wird.

Jenkins

Jenkins wurde lokal als Docker Container mittels Docker-Compose ausgeführt. Grund für die Verwendung des Docker Containers ist die so sichergestellte Unabhängigkeit von der Arbeitsumgebung, da die Aufgabe an unterschiedlichen Geräten bearbeitet wurde.

Docker Compose wurde verwendet, da der Container dadurch einfacher zu handhaben ist, dieser kann so mittels `docker-compose up -d` mit den notwendigen Konfigurationen (welche in `docker-compose.yml`hinterlegt werden) gestartet werden. Ausserdem erlaubt Docker-Compose die Verwendung von relativen Pfadnamen für die Volumes und das einfache Einbinden weiterer Container. Die `docker-compose.yml` wurde wie folgt erstellt:

```
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

  sonarqube:
      image: sonarqube:latest
      container_name: sonarqube
      ports:
          - "9000:9000"
          - "9092:9092"
      volumes:
          - ./sonarqube-data:/opt/sonarqube/data
          - ./sonarqube-extensions:/opt/sonarqube/extensions
          - ./sonarqube-logs:/opt/sonarqube/logs
          - ./sonarqube-temp:/opt/sonarqube/temp
      restart: always
```

So werden gleichzeitig die beiden Services `jenkins` und `sonarqube` gestartet. Die beiden Container können sich über das Default Docker-Compose Netz erreichen. Die Daten der beiden Container werden in den Ordnern `jenkins-data`, `sonarqube-data`, `sonarqube-extensions`, `sonarqube-logs` und `sonarqube-temp` gespeichert. Die Ports `8080` und `50000` werden für Jenkins und die Ports `9000` und `9092` für SonarQube gegen aussn (Host) freigegeben. Die beiden Container werden automatisch neu gestartet, falls sie beendet werden.

Das Webinterface von Jenkins 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 von Jenkins mit den default Plugins und dem User `matthias` abgeschlossen.

Konfiguration von Jenkins

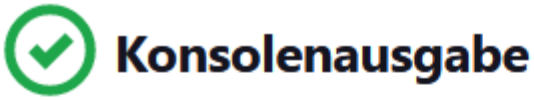
Zunächst wird die Authentifizierung an der Google Cloud Platform (GCP) konfiguriert. Dazu muss die Google Cloud CLI im Jenkins Container gemäss der Anleitung von cloud.google.com/sdk/docs/install#deb installiert werden. Dies muss mittels Bash im Container durchgeführt werden, dazu muss die Shell als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

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

Mit Hilfe der folgenden kleinen Test-Pipeline wird geprüft, ob die Google Cloud CLI korrekt installiert wurde:

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

Die Ausgabe bestätigt die erfolgreiche Installation:




```
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 in der Google Cloud Plattform 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 in Jenkins über `Credentials` -> `System` -> `Global credentials (unrestricted)` -> `Add Credentials` ein neues Secret vom Typ `Secret File` mit Hilfe des JSON Keys erstellt. In der Pipeline muss die GCP Projekt-ID `cellular-syntax-231507` mittels Variable `CLOUDSDK_CORE_PROJECT` und das zu verwendende Secret via `withCredentials()`

gesetzt werden. Die Pipeline zum Testen der Credentials sieht wie folgt aus (`gcloud compute zones list` 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 funktionierte der Test erfolgreich.

Jenkins Pipeline

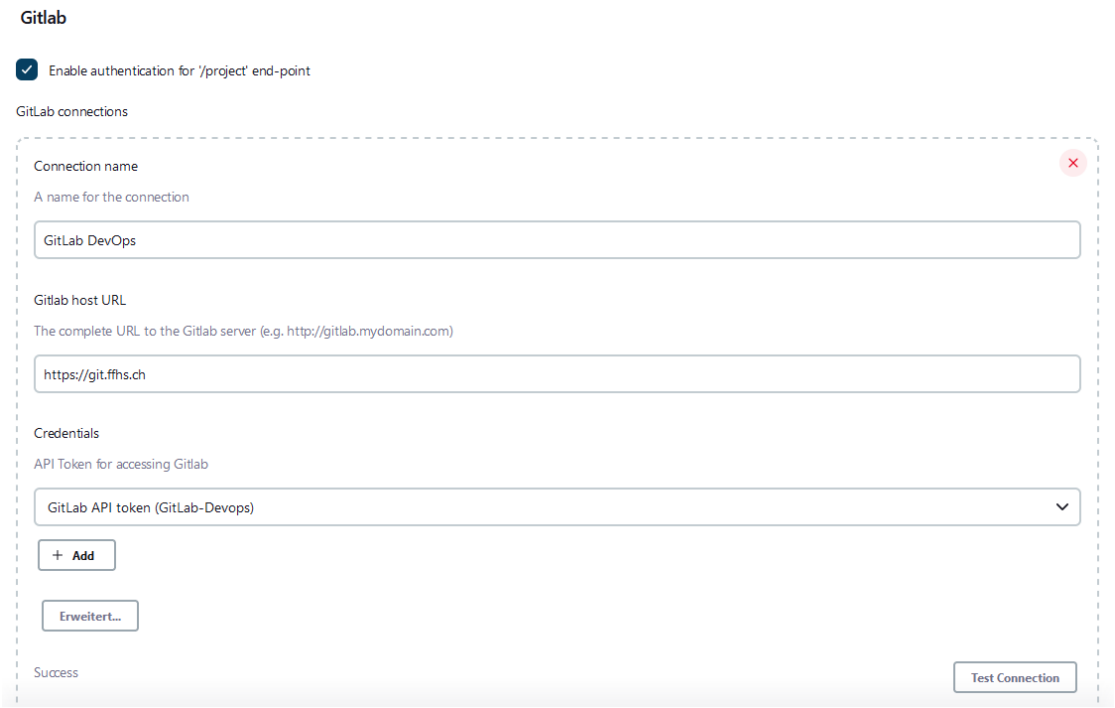
Nach erfolgreicher Konfiguration von Jenkins wird die eigentliche Pipeline aufgesetzt. Die Pipeline soll die folgenden Punkte erfüllen:

- Die Pipeline soll bei jedem Push auf den `develop`-Branch ausgelöst werden
- In der Pipeline soll der Code mittels SonarQube geprüft werden. Falls die Code-Qualität nicht ausreichend ist (Check via definiertem Quality Gate), soll die Pipeline abgebrochen werden
- In der Pipeline soll das Docker Image gebaut und in die Google Container Registry gepusht werden
- Die Pipeline 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 git.ffhs.ch/matthias.heimberg/devops eingebunden (so ist die Versionierung des

Files sichergestellt). Die für die Verbindung zu GitLab erforderlichen Credentials werden in Jenkins hinterlegt (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 (via Jenkins Webhook). Da Jenkins aber nur lokal ausgeführt wird und via git.ffhs.ch nicht erreichbar ist, 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',
                    usernameVariable: 'GITLAB_USER',
                    passwordVariable: 'GITLAB_PASSWORD'
                    )]) {
```

```
                git url: 'https://git.ffhs.ch/matthias.heimberg/devops.git',
                branch: 'develop',
                credentialsId: 'gitlab-access'
            }
        }
    }
```

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: 'GCLOUDCOMPUTE')]) {
            sh '''
                gcloud auth activate-service-account \
                --key-file="$GCLOUDCOMPUTE"
                gcloud run deploy devops \
                  --image gcr.io/cellular-syntax-231507/devops \
                  --platform managed \
                  --region europe-west4 \
                  --allow-unauthenticated \
                  --port 7000 \
                  --service-account \
                  382263290396-compute@developer.gserviceaccount.com
```

```
            '''
        }
    }
```

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 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 (via Gmail) an den Entwickler senden kann, muss das Extended Email Plugin installiert sein. Im Google Account wird ein App Password für den Jenkins User erstellt. Dieses wird in den Jenkins Credentials hinterlegt. Das Mail wird in jedem Fall versendet, unabhängig vom Build Status. Die Post Action sieht wie folgt aus:

```
post {
        success {
            // archive the artifacts
            archiveArtifacts artifacts: '**/build/libs/*.jar',
            fingerprint: true
        }
        always {
            emailext (
                subject: 'Jenkins build: $BUILD_STATUS',
                body: '$BUILD_URL',
                from: 'jenkins',
                to: 'matthias.heimberg@students.ffhs.ch'
            )

        }
    }
```

Bei Erfolg wird das Build Artefakt archiviert. Nach jedem Build wird das Mail an die angegebene Adresse mit dem Build Status versendet.

Gitlab CI/CD Um die Pipeline auch im Gitlab Repository unter CI/CD sichtbar zu machen, wird das Jenkinfile wie folgt angepasst: - der Pipeline wird ein Block `options`hinzugefügt, in welchem die Gitlab CI/CD Konfiguration definiert wird. Die dazu notwenige GitLab Verbindung wird in Jenkins via GitLab Plugin erstellt. - zu jedem Step wird ein `gitlabCommitStatus` hinzugefügt, welcher den Build Status in GitLab updated.

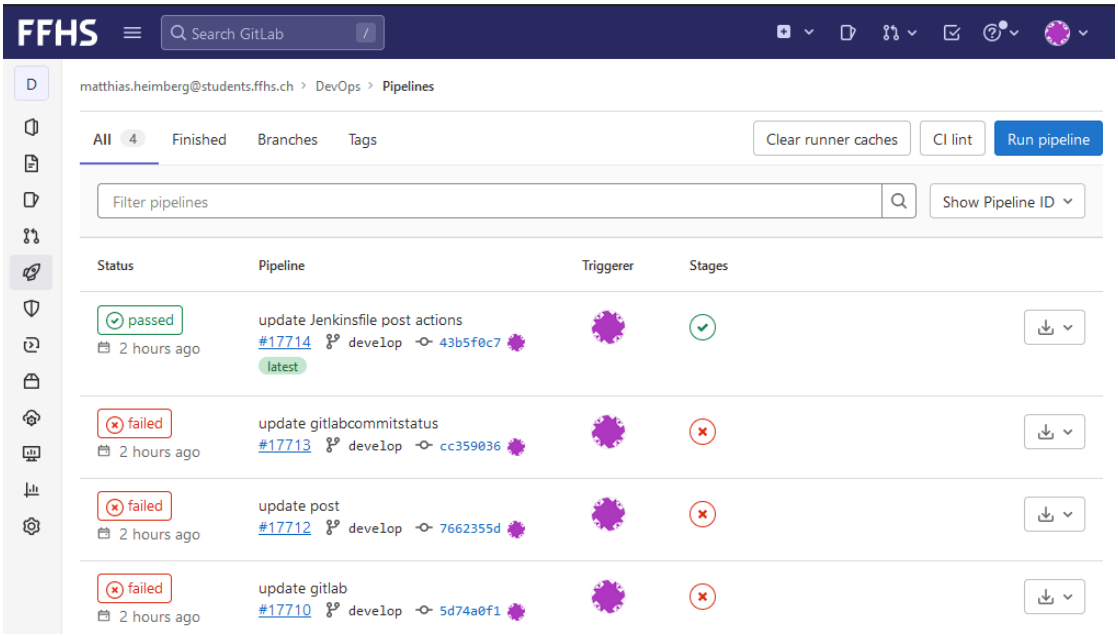


Figure 1: GitLab CI/CD

Fertiges Jenkinsfile

Die Pipeline sieht nach den Anpassungen wie folgt aus:

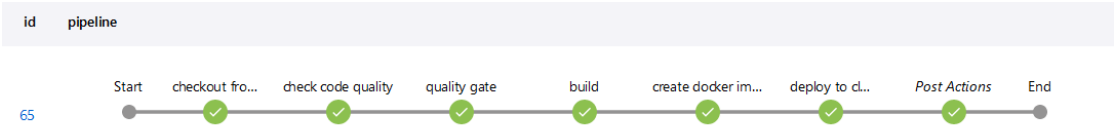
```
pipeline {

    agent any

    environment {
    CLOUDSDK_CORE_PROJECT='cellular-syntax-231507'
    }

    options {
        gitLabConnection('GitLab DevOps')
        gitlabCommitStatus(name: 'Jenkins')
```

```
    gitlabBuilds(builds: [
      'checkout from GitLab',
      'check code quality',
      'quality gate',
      'build',
      'create docker image and push to registry',
      'deploy to cloud run'
      ])
}

stages {
    stage('checkout from GitLab') {
        steps {
            gitlabCommitStatus(name: 'checkout from GitLab') {
                withCredentials([usernamePassword(
                  credentialsId: 'gitlab-access',
                  usernameVariable: 'GITLAB_USER',
                  passwordVariable: 'GITLAB_PASSWORD')]) {
                    git url: 'https://git.ffhs.ch/matthias.heimberg/devops.git',
                    branch: 'develop',
                    credentialsId: 'gitlab-access'
                }
            }
        }
    }
    stage('check code quality') {
        steps {
            gitlabCommitStatus(name: 'check code quality') {
                withSonarQubeEnv('SonarQube') {
                    sh 'chmod +x ./gradlew'
                    sh './gradlew sonarqube -D"sonar.projectKey=DevOps"'
                }
            }
        }
    }
    stage('quality gate') {
        steps {
            gitlabCommitStatus(name: 'quality gate') {
                waitForQualityGate abortPipeline: true
            }
        }
    }
    stage('build') {
        steps {
```

```
            gitlabCommitStatus(name: 'build') {
                sh './gradlew build'
            }
        }
    }
    stage('create docker image and push to registry') {
        steps {
            gitlabCommitStatus(name: 'create docker image and push to registry') {
                withCredentials([file(
                  credentialsId: 'gcloud',
                  variable: 'GCLOUD')]) {
                    sh '''
                        gcloud auth activate-service-account \
                        --key-file="$GCLOUD"
                        ./gradlew jib
                    '''
                }
            }
        }
    }
    // deploy to google cloud run on port 7000
    stage('deploy to cloud run') {
        steps {
            gitlabCommitStatus(name: 'deploy to cloud run') {
                withCredentials([file(
                  credentialsId: 'gcloudcompute',
                  variable: 'GCLOUDCOMPUTE')]) {
                    sh '''
                        gcloud auth activate-service-account \
                          --key-file="$GCLOUDCOMPUTE"
                        gcloud run deploy devops \
                          --image gcr.io/cellular-syntax-231507/devops \
                          --platform managed \
                          --region europe-west4 \
                          --allow-unauthenticated \
                          --port 7000 \
                          --service-account \
                          382263290396-compute@developer.gserviceaccount.com
                    '''
                }
            }
        }
    }
}
```

```
    post {
        success {
            // archive the artifacts
            archiveArtifacts artifacts: '**/build/libs/*.jar', fingerprint: true
        }
        always {
            emailext (
                subject: 'Jenkins build: $BUILD_STATUS',
                body: '$BUILD_URL',
                from: 'jenkins',
                to: 'matthias.heimberg@students.ffhs.ch'
            )

        }
    }
}
```

In Jenkins wird die Pipeline via Pipeline Graph View Plugin wie folgt angezeigt:



Das Jenkinsfile ist unter git.ffhs.ch/matthias.heimberg/devops/-/blob/develop/Jenkinsfile zu finden.

Probleme und deren Lösung

- Docker Image aus Build der Applikation erstellen (klappt wegen Dependencies nicht) -> Lösung: Image mittels `Jib` erstellen. `Jib` erleichtert das Erstellen von Docker Images und ist in der Lage, die Dependencies automatisch zu laden und in das Image zu packen.
- Docker Image aus Docker Hub lässt sich nicht in Google Cloud Run deployen -> Lösung: Image in die Google Cloud Registry pushen.
- Docker Image auf Google Container Registry pushen scheitert an fehlenden Berechtigungen -> 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.
- Mail wird nicht über Google Mail versendet -> Lösung: Google Mail erlaubt keine Anmeldung von Apps, daher muss ein App Passwort erstellt werden. Der Google SMTP Server ist `smtp.gmail.com` und der Port ist '467'.
- Pipeline Status kann nicht an GitLab übermittelt werden -> Lösung: `options` in der Pipeline hinzufügen und `gitlabCommitStatus` in den einzelnen Stages verwenden.

Dokumentation

Die Dokumentation wird mit Hilfe von `pandoc` aus Markdown generiert:

```
pandoc  -V linkcolor:blue \
  --pdf-engine=xelatex \
  .\Documentation\Aufgabe_2.md \
  -o .\Documentation\Aufgabe_2.pdf
```