Sonderdruck aus
JavaSPEKTRUM 4/2019

Leben im Container

Beispiel einer Microservices-Architektur mit Docker

Marcel Jödicke

In IT-Betrieben geht die Entwicklung heutzutage zunehmend weg von monolithischen Systemen über N-Schichtensysteme hin zu Microservices. Zum Verpacken dieser Microservices hat sich die Container-Technologie zunehmend als ideale Möglichkeit in vielen Unternehmen etabliert. Durch Container können Applikationen einfach geteilt und unabhängig vom Host-System überall gleich ausgeführt werden. Die Unterschiede von Host-Systemen in Bezug auf Sicherheit, Netzwerktopologien und andere Abhängigkeiten werden durch Container vollständig abstrahiert. In diesem Artikel soll es nicht erneut um die Grundlagen zur Ausführung von Containern oder um die Funktionsweise von Containern im Allgemeinen gehen. Darüber gibt es bereits genügend Artikel. Es soll in Form eines Tutorials beschrieben werden, wie Container verwendet werden können, um Microservices aufzubauen.

Dafür muss zuerst einmal betrachtet werden, was wichtige Eigenschaften von Microservices sind, und was diese Eigenschaften in Bezug auf Container für Vorteile aufweisen.

Wichtige Eigenschaften von Microservices

Die erste positive Eigenschaft von Microservices steckt schon im Namen: Sie sind klein. Bei Microservices wird idealerweise jeweils eine Funktion oder ein Prozess der Gesamtanwendung ausgeführt. Die Gesamtanwendung ergibt sich aus dem Zusammenschluss und der Kommunikation zwischen den Microservices. In der Docker-Dokumentation findet sich im Abschnitt „Best practices for writing Dockerfiles“:

„Each container should have only one concern. Decoupling applications into multiple containers makes it easier to scale horizontally and reuse containers. For instance, a web application stack might consist of three separate containers, each with its own unique image, to manage the web application, database, and an in-memory cache in a decoupled manner.“ [Dock1]

Das klingt doch schon stark nach Microservices. Das Beispiel wollen wir nun aufgreifen. Der Einfachheit halber bauen wir lediglich einen Microservice bestehend aus einem MySQL- oder Maria DB-Server und einer Node.js-Anwendung.

Erstellen wirklich kleiner Images

In Bezug auf Microservices ist es nicht nur sinnvoll, die Funktionen beziehungsweise Prozesse einzeln in Containern zu hosten, sondern auch die Container-Images möglichst klein zu halten.

Dies bietet gleich mehrere Vorteile. Zum einen starten die Container aus den Container-Images viel schneller, wenn diese sehr klein gehalten werden. Zum anderen bieten Container aufgrund

Marcel Jödicke arbeitet seit 2018 bei EasiRun Europa GmbH und befasst sich seit der Einführung von Docker in Windows mit der Nutzung von Containern in Projekten. Er hält Docker für eine großartige Technologie und teilt gerne seine Erfahrungen zur Nutzung in Projekten mit anderen Lesern. E-Mail: mjoedicke@easirun.de

dessen, dass sie nur die Abhängigkeiten besitzen, die sie wirklich benötigen, um den speziellen Prozess laufen zu lassen, eine sehr geringe Angriffsfläche. Es werden zum Beispiel nur Ports freigegeben, die benötigt werden, um den jeweiligen Prozess laufen zu lassen, was einen Angriff erschwert.

Wenn die Images klein gehalten werden, ergeben sich weitere Vorteile: Das System muss selten gereinigt werden und der Netzwerkverkehr und Building-Prozess können viel schneller durchgeführt werden.

Erstellen von Container-Images von „scratch“

Behalten wir das oben Ausgeführte im Hinterkopf, kommen wir zur kleinstmöglichen Art und Weise, ein Container-Image zu erzeugen: indem man ein Image von „scratch“ baut. Im Folgenden gehe ich auf das offizielle „hola-mundo“-Image [Dock2] ein.

Betrachtet man das Dockerfile des Images, stellt man fest, dass es lediglich aus diesen Befehlen besteht:

```
FROM scratch
COPY hello /
CMD ["/hello"]
```

Mit dem `FROM scratch`-Befehl kommunizieren wir, dass wir Dockers Minimal-Image verwenden wollen. Danach ist zu sehen, dass ein kompiliertes „hello“-Programm in den Container kopiert wird. Dieses ist zum Beispiel ein einfaches C++-Programm mit folgendem Inhalt:

```
#include<iostream>
using namespace std;
int main(){
  cout << "Hello World\n";
  return 0;
}
```

Das kompilierte Programm wird dem Ordner des Dockerfiles mit dem Namen „hello“ hinzugefügt. Über das letzte Kommando des Dockerfiles wird das Programm dann automatisch beim Container-Start ausgeführt.

Betrachten wir die einzelnen Schichten des Images, stellen wir fest, dass das Erstellen des Images von „scratch“ keinen Speicher beansprucht und der Gesamtcontainer durch das „Hello World“-Programm gerade einmal 2,25 MB groß ist (s. Listing 1). Wenn nur die Ausführung eines eigenen Programmes im Container erwünscht ist, ist dies sicherlich die zu empfehlende Variante.

Beziehen des „node:alpine“-Images

Für unsere Webanwendung benötigten wir jedoch Node.js inklusive seiner Abhängigkeiten sowie einen MySQL- oder MariaDB-Server. Dafür empfiehlt sich die nächstgrößere Möglichkeit zum Erstellen von Container-Images, nämlich das Verwenden eines sehr kleinen Container-Basis-Images. Zu empfehlen ist dabei das Image alpine. Dieses ist ein sehr kleines Linux-Image mit einer Größe von gerade einmal 5,53 MB (s. Listing 2).

Ein großer Vorteil bei der Nutzung von Docker ist, dass bereits viele vorgefertigte Images zur Verfügung stehen. Deshalb werden wir das Rad nicht neu erfinden und für Node.js das offizielle Image vom Docker-Hub verwenden [Dock3]. Das Beziehen erfolgt über:

```
docker pull node:alpine
```

Auswahl des Datenbank-Server-Images

Für MySQL existiert leider noch kein Image für alpine. Wenn man für die Datenbank jedoch nicht zwangsläufig auf MySQL angewiesen ist, kann auch das sehr kleine MariaDB-Image für alpine von wangxian verwendet werden [Dock4].

Wenn vorgefertigte Images aus dem Docker-Hub verwendet werden, empfiehlt es sich, im Rahmen des Zieles, Images möglichst klein zu halten, das Dockerfile zu überprüfen und gegebenenfalls anzupassen. Im Folgenden wollen wir das Dockerfile von wangxian für MariaDB betrachten:

```
FROM alpine:latest
MAINTAINER WangXian <xian366@126.com>

WORKDIR /app
VOLUME /app
COPY startup.sh /startup.sh

RUN apk add --update mysql mysql-client && rm -f /var/cache/apk/*
COPY my.cnf /etc/mysql/my.cnf

EXPOSE 3306
CMD ["/startup.sh"]
```

```
root@mj_docker:/data/Artikel# docker history hello
IMAGE            CREATED          CREATED BY                                      SIZE
a6fb62c74c08     16 sec ago       /bin/sh -c #(nop) CMD ["/hello"]                0B
42290fdd01b9     16 sec ago       /bin/sh -c #(nop) ADD file:6f210ca844cef029b...  2.25MB
root@mj_docker;/data/Artikel#
```

Listing 1: Historie des Hello-World-Images von „scratch“ mit Größendarstellung

```
root@mj_docker:/data/Artikel# docker history alpine
IMAGE            CREATED          CREATED BY                                      SIZE
cdf98d1859c1     6 days ago       /bin/sh -c #(nop) CMD ["/bin/sh"]               0B
<missing>        6 days ago       /bin/sh -c #(nop) ADD file:2e3a37883f56a4a27...  5.53MB
root@mj_docker;/data/Artikel#
```

Listing 2: Historie des Alpine-Images mit Größendarstellung

Es verwendet wie gewünscht alpine als Basis-Image. Danach legt es ein Arbeitsverzeichnis und einen geteilten Ordner für Applikationsdaten an. Das Shell-Skript beinhaltet Kommandos, die für die Einrichtung und das Starten des MariaDB-Servers notwendig sind, und reagiert auf die vom Benutzer beim Container-Start übergebenen Parameter.

Wichtig ist der nächste Abschnitt, bei dem zuerst der MySQL-Server installiert und im Anschluss direkt der Cache gelöscht wird. Dadurch bleibt das Image sehr klein. Alternativ kann das „apk add"-Kommando auch über „apk --nocache add" ausgeführt werden. Das Image ist also ideal darauf ausgelegt, platzsparend zu sein. Dies spiegelt sich mit einer Größe von gerade einmal 176 MB wieder.

Wenn ein MySQL-Server verwendet werden muss, kann auch das offizielle MySQL-Server-Image vom Docker Hub heruntergeladen werden. Dieses basiert auf dem „debian:stretch-slim"-Basis-Image, das mit 55,3 MB ebenfalls sehr klein ist. Jedoch ist das vollständige MySQL-Image mit 477 MB größer als das MariaDB-Image für alpine. Deshalb werden wir im Folgenden für die einfache Webanwendung das MariaDB-Image verwenden.

MariaDB-Server über Dockerfile vorbereiten

In diesem Abschnitt bereiten wir den MariaDB-Server für unsere Node.js-Applikation vor. Da die im Docker-Hub beschriebenen Parameter für das Erstellen eines neuen Benutzers mit Passwortzuweisung bei unserer Testinstallation nicht funktionierten, nehmen wir diesen Teil manuell vor. Wir starten den Container über:

```
docker run -it --name mysql -p 3306:3306 -v $(pwd):/app
  -e MYSQL_ROOT_PASSWORD=pass
  -e MYSQL_DATABASE=finale
  -e MYSQL_USER=nodeuser
  -e MYSQL_PASSWORD=pass wangxian/alpine-mysql
```

Dadurch wird bereits eine „finale" Datenbank erstellt. Im Anschluss öffnen wir eine neue Shell über:

```
docker exec -it mysql sh
```

Die Anmeldung am MariaDB-Server erfolgt mit dem Kommando:

```
mysql -uroot
```

Nach der Anmeldung führen wir das folgende Kommando aus:

```
USE mysql;
FLUSH PRIVILEGES;
CREATE USER 'nodeuser'@'localhost' IDENTIFIED BY 'pass';
GRANT ALL PRIVILEGES ON *.* TO 'nodeuser'@'localhost' IDENTIFIED BY 'pass';
```

Dadurch wird ein Nutzer „nodeuser" mit dem Passwort „pass" angelegt, der volle Berechtigungen für die Datenbanken erhält. Im Anschluss gehen wir in unsere „finale" Datenbank, erstellen eine Tabelle mit dem Namen „ende" und fügen die folgenden Strings hinzu:

```
USE finale;
CREATE TABLE ende (endstring TEXT);
INSERT INTO ende (endstring) values ('Tutorial');
INSERT INTO ende (endstring) values ('ist');
INSERT INTO ende (endstring) values ('zu');
INSERT INTO ende (endstring) values ('Ende');
```

Damit ist die Datenbank für unsere Node.js-Anwendung vorbereitet und kann über `exit` verlassen werden.

Erstellen der Node.js-Anwendung

Nachdem die Datenbank für das Tutorial eingerichtet ist, wird als Nächstes die Node.js-Anwendung vorbereitet. Zuerst betrachten wir die „index.js"-Datei (s. Listing 3). Dort wird eine Datenbankverbindung über die Konfigurationen, die sich in der „config.js"-Datei befinden, aufgebaut und der Server gestartet.

```
var server = require('./server/server');
var dbhandler = require('./dbhandler/dbhandler');
var config = require('./config/config');

// Verbinden mit dem Repository.
dbhandler.connect({
  host: config.db.host, database: config.db.database,
  user: config.db.user, password: config.db.password,
  port: config.db.port
}).then((hand) => {
  console.log("Verbunden");
// Starten des Servers
  return server.start({
    port: config.port, dbhandler: hand
  });
}).then((app) => {
  console.log(
    "Der Server ist erfolgreich gestartet und läuft auf Port "
    + config.port + ".");
  app.on('close', () => {
    dbhandler.disonnect();
  });
});
```

Listing 3: „index.js"-Datei

```
module.export = {
  port: process.env.PORT || 8123,
  db: {
    host: process.env.DATABASE_HOST || '127.0.0.1',
    database: 'finale', user: 'nodeuser',
    password: 'pass', port: 3306
  }
};
```

Listing 4: „config.js"-Datei

Die Konfiguration der „config.js"-Datei betrachten wir als Nächstes (s. Listing 4). Zuerst wird der Port für die Node.js-Anwendung festgelegt. Im Anschluss werden die Daten für die Verbindung zur Datenbank eingetragen, die wir oben spezifiziert haben.

```
var express = require('express');
var morgan = require('morgan');
var path = require('path');

module.exports.start = (options) => {
  return new Promise((resolve, reject) => {
    // Erstellen der Express-Anwendung
    var app = express();
    app.use(morgan('dev'));

    // Zuweisen von pug als view engine
    app.set('views', path.join(__dirname, '../views'));
    app.set('view engine', 'pug');

    // Hinzufügen des API zur Expressapplikation
    require('../api/endfield')(app, options);

    // Starten der Applikation
    var server = app.listen(options.port, () => {
      resolve(server);
    });
  });
};
```

Listing 5: „Server.js"-Datei

Nun betrachten wir die „Server.js“-Datei, der wir in der „index.js“-Datei den Port und „dbhandler“ übergeben haben. Wie in Listing 5 zu sehen ist, wird dabei eine reguläre Express-Applikation erstellt. Die übergebenen Parameter werden an das „endfield“-API übergeben und die Applikation wird gestartet. Ebenfalls wichtig hierbei ist, dass wir Pug als view-Engine verwenden, sowie morgan.

```
{
  "name": "microservice", "version": "0.0.0",
  "private": true, "main": "index.js",
  "scripts": {
    "start": "node index.js"
},
  "dependencies": {
    "express": "~4.16.0", "morgan": "~1.9.0",
    "pug": "~^2.0.3", "mysql": "^2.10.2",
  }
}
```

Listing 6: „package.json“-Datei

Alle Abhängigkeiten müssen der „package.json“-Datei hinzugefügt werden (s. Listing 6). In der „package.json“-Datei werden ein Name und eine Version vergeben. Der Start der Applikation erfolgt über die „index.js“-Datei. Notwendige Abhängigkeiten in diesem einfachen Beispiel sind express, morgan, pug und mysql.

```
'use strict';
module.exports = (app, options) => {
  app.get('/end', (req, res, next) => {
    options.dbhandler.getEnd().then((ende) => {
      res.render('index', {ende: ende});
    })
    .catch(next);
    });
};
```

Listing 7: Die Programmierschnittstelle

In Listing 7 betrachten wir das API. Die Programmierschnittstelle besitzt lediglich den Pfad „/end“. Mit dem „dbhandler“ werden die vorher eingetragenen Datenbankdaten bezogen und diese mit einer Variablen, die hier als „ende“ deklariert ist, der „index.pug“-Datei zum Rendern übergeben.

```
'use strict';
var mysql = require('mysql');

class dbhandler {
  constructor(connection) {
    this.connection = connection;
  }
  getEnd() {
    return new Promise((resolve, reject) => {
      this.connection.query('SELECT endstring FROM ende',
          (err, results) => {
        if(err) {
          return reject(new Error(
  "Ein Fehler ist aufgetreten beim Betreten des Strings" + err));
        }
        resolve((resluts || []).map((end) => {
          return {
            ende: end.endstring
          };
        }));
      });
    });
  }
  disconnect() {
    this. connection.end();
  }
}
```

Fortsetzung siehe Folgespalte oben

```
module.exports.connect = (connectionSettings) => {
  return new Promise((resolve, reject) => {
    if(!connectionSetting.host throw new Error(
      "Ein Host muss angegeben werden");
    if(!connectionSetting.user throw new Error(
      "Ein Benutzer muss angegeben werden");
    if(!connectionSetting.password throw new Error(
      "Ein Passwort muss angegeben werden");
    if(!connectionSetting.port throw new Error(
      "Ein Port muss angegeben werden");
    resolve(new dbhandler(mysql.createConnection(
      connectionSettings)));
  });
};
```

Listing 8: Datenbank-Handler

```
extends layout

block content
  h1="Mikroservice Anwendung"
  p Willkommen zum kleinen Mikroservice Tutorial

  button(Name="tutbutton" type= "button" onclick="Tutorial()") Klick
  button(Name="istbutton" type= "button" onclick="ist()") Klick
  button(Name="zubutton" type= "button" onclick="zu()") Klick
  button(Name="endbutton" type= "button" onclick="ende()") Klick

  <div class="altoghether">
    <div id="tuthid" style="visibility:hidden; display:inline-block">
      p !{ende[0].ende}
    </div>
    ...
    <div id="endehid" style="visibility:hidden; display:inline-block">
      p !{ende[3].ende}
    </div>
  </div>
  script.
    function Tutorial() {
      document.getElementByID('tuthid').style.visibility = "";
    }
    ...
    function ende() {
      document.getElementByID('endehid').style.visibility = "";
    }
```

Listing 9: Pug-Datei

Listing 8 zeigt den Datenbank-Handler. Dort ist zu erkennen, dass hier einerseits beim Verbinden die Fehlerabfrage durchgeführt wird und andererseits wird hier die `getEnd()`-Funktion aufgerufen, die die vorher eingetragenen Strings aus der „ende“-Tabelle bezieht. Außerdem wird hier die `disconnect()`-Funktion definiert.

Zum Schluss betrachten wir noch die Pug-Dateien. Wie im API zu sehen ist, wird beim Rendern die „index.pug“-Datei aufgerufen. Diese sehen wir in Listing 9. Zunächst wird die „layout.pug“-Datei für die Grundeinstellungen verwendet. Diese betrachten wir zum Schluss. Danach gibt es eine Einleitung für die dargestellte Website. Im Anschluss werden vier Knöpfe erstellt, die beim Klick jeweils eine eigene Funktion aufrufen. Durch diese Funktionen wird jeweils ein Wort darunter sichtbar, das in der div-Klasse definiert ist. Die Wörter werden aus der Datenbank bezogen.

```
doctype html
html
  head
    title = title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

Listing 10: layout.pug-Datei

Die letzte Datei für die Beispielapplikation ist die „layout.pug“-Datei (s. Listing 10). Das Layout beschreibt lediglich, dass es sich um ein HTML-Dokument handelt. Es besitzt simple Grundeinstellungen und bindet eine css-Datei ein.

```
# Benutzung von node:alpine als Basisimage
FROM node:alpine

# Hinzufügen aller Verzeichnisse und Komponenten für die
# Node.js-Anwendung in das /app-Verzeichnis
ADD . /app

# Installieren alle Abhängigkeiten definiert in der Datei package.json
RUN cd /app; \
  npm install --production

# Veröffentlichen der Ports, unter denen die Applikation
# von außen erreichbar ist
EXPOSE 8123

# Starten der Anwendung mit Containerstart
CMD ["node", "/app/index.js"]
```

Listing 11: Dockerfile zum Bauen als Container-Image

Das Dockerfile zum Bauen der Anwendung als Container-Image zeigt Listing 11. Die Ordnerstruktur für die Applikation sollte wie in Listing 12 aussehen.

```
root@mj_docker:/home/easirun/project/users-service# ls
api config dhbhandler dockerfile index.js package.json server views
```

Listing 12: Ordnerstruktur für die Applikation

Angenommen, das Image wurde mit dem Namen „app“ gebaut, erfolgt das Erstellen und Starten des Containers über das Kommando:

```
docker run -it -p 8123:8123 --link mysql:mysql
  -e DATABASE_HOST=mysql --name nodejs app
```

Durch `-p` wird der Port 8123 des Containers nach außen hin verfügbar gemacht. Über `--link mysql:mysql` wird dem Applikations-Container die IP-Adresse des Containers mit dem Namen „mysql“ bekannt gemacht. Sonst haben die Container untereinander keinen Zugriff. Über `-e DATABASE_HOST=mysql` wird die IP-Adresse des MySQL-Containers der Umgebungsvariable `DATABASE_HOST` zugeordnet. Zum Schluss wird dem Container mit `--name nodejs` der Name „nodejs“ zugewiesen.

Nach dem Start ist das Endergebnis unter `http://<IP>:8123/end` aufrufbar.

Vorbereitung der Konfiguration für Docker Compose

Als Nächstes bereiten wir die Einstellungen vor, um das Starten der Microservices unter Docker Compose vorzunehmen. Dies hat einige Vorteile:

- Alle zusammengehörigen Komponenten können mit ihren Abhängigkeiten sowohl zusammengebaut als auch gestartet werden.
- Es können mehrere Instanzen gemeinsam gestartet werden, um Ausfallsicherheit sowie Skalierbarkeit zu gewährleisten.

Um die Nutzung von Docker Compose zu vereinfachen, schreiben wir zunächst einmal ein weiteres simples Dockerfile, das das Image des MySQL-Servers erweitert. Dafür fügen wir zuerst einige Einträge in die „startup.sh“-Datei des „wangxian/alpine-mysql“-Images hinzu (s. Listing 13). Dadurch werden die vorher manuell hinzugefügten Einträge mit dem Container-Start direkt hinzugefügt.

```
cat >> EOF > $tfile
USE mysql;
FLUSH PRIVILEGES;
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY
  "$MYSQL_ROOT_PASSWORD" WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION;
UPDATE user SET password=PASSWORT("")
  WHERE user='root' AND host='localhost';
FLUSH PRIVILEGES;
CREATE USER 'nodeuser'@'localhost' INDENTIFIED BY 'pass';
GRANT ALL PRIVILGES ON *.* TO 'nodeuser'@'loccalhost'
  IDENTIFIED BY "pass";
CREATE DATABASE finale;
USE finale;
CREATE TABLE ende (endstring TEXT);
INSERT INTO ende (endstring) values ('Tutorial');
INSERT INTO ende (endstring) values ('ist');
INSERT INTO ende (endstring) values ('zu');
INSERT INTO ende (endstring) values ('Ende');
EOF
```

Listing 13: Erweiterung des MySQL-Servers

```
FROM wangxian/alpine-mysql
COPY startup.sh /startup.sh

CMD ["/startup.sh"]
```

Listing 14: Dockerfile

Das passende neue Dockerfile dazu zeigt Listing 14. Es basiert auf den „wangxian/alpine-mysql“-Image, kopiert die neue „startup.sh“-Datei und fügt sie dem Autostart hinzu.

Im Anschluss ist es für Docker Compose sinnvoll, einen Projektordner zu erstellen. Dieser enthält dann zwei Unterordner, einen für den MySQL-Server mit Dockerfile und allen Abhängigkeiten und einen für die Applikation mit Dockerfile und allen zugehörigen Abhängigkeiten. Der Projektordner selbst enthält die „docker-compose.yml“-Datei, die die Konfigurationen für Docker Compose beinhaltet. Die vollständige Projektstruktur zeigt Listing 15. Die „docker-compose.yml“-Datei beinhaltet die in Listing 16 gezeigten Einträge.

```
root@mj_docker:/home/easirun/project# ls
app docker-compose.yml sql
```

Listing 15: Projektstruktur

```
version: '2'
services:
  app;
    build: ./app
    ports:
    - "8123:8123"
  depends_on
  -db
  environment:
    DATABASE_HOST: db
  links:
```

Fortsetzung siehe Folgespalte oben

```
  - db:db
db:
  build: ./sql
  environment:
    MYSQL_DATABASE: "finale"
    MYSQL_USER: "nodeuser"
    MYSQL_PASSWORD: "pass"
```

Listing 16: docker-compose.yml

Die für die Applikation zu bauenden Dateien befinden sich also unter „./app“. Der Port für die Applikation ist 8123 sowohl im Container als auch über den Container-Host. Es ist ebenfalls zu sehen, dass erst der Datenbank-Container für die Applikation gestartet sein muss. Als Umgebungsvariable wird die IP von „db“ übergeben. Für die Datenbank befinden sich die Daten im „./sql“-Verzeichnis. Die zu setzenden Umgebungsvariablen sind MySQL_DATABASE, MYSQL_USER und MYSQL_PASSWORD.

Nutzung von Docker Compose

Nach der Installation von Docker Compose kann das vollständige Projekt gebaut werden über:

```
docker-compose build:
```

Listing 17 zeigt den entsprechenden Bau der Container-Images. Um zusätzlich die Ausfallsicherheit mit Docker Compose zu demonstrieren, starten wir das Projekt über:

```
docker-compose up --scale db=5
```

```
root@mj_docker:/home/easirun/project# docker-compose build
Building db
Set 1/3 : FROM wangxian/alpine-mysql
  ---> 8c8f45aeedf2
Step 2/3 : COPY startup.sh /startup.sh
  ---> Using cache
```

Listing 17: Zusammenbau aller Einzelkomponenten mit docker-compose

Dies ist in Listing 18 dargestellt. Hiermit werden gleich mehrere Instanzen von Datenbank-Containern gestartet. Dadurch kann bis zur Hälfte der Datenbank-Container ausfallen oder gestoppt werden, aber die Applikation ist immer noch voll funktionsfähig (s. Listing 19).

```
root@mj_docker:/home/easirun/project# docker-compose up --scale db=5
Creating network "project_default" with the default driver
Creating project_db_1 ... done
Creating project_db_2 ... done
Creating project_db_3 ... done
Creating project_db_4 ... done
Creating project_db_5 ... done
Creating project_app_1 ... done
```

Listing 18: Ausführung von docker-compose mit Start von fünf DB-Containern

```
root@mj_docker:/home/easirun/project# docker stop project_db_1
project_db_1
root@mj_docker:/home/easirun/project# docker stop project_db_5
project_db_5
root@mj_docker:/home/easirun/project# docker-compose ps
Name          Command            State   Ports
------------------------------------------------------------------
project_app_1 node /app/index.js Up      0.0.0.0.8123->8123/tcp
project_db_1  /startup.sh        Exit 0
project_db_2  /startup.sh        Up      3306/tcp
project_db_3  /startup.sh        Up      3306/tcp
project_db_4  /startup.sh        Up      3306/tcp
project_db_5  /startup.sh        Exit 0
root@mj_docker:/home/easirun/project#
```

Listing 19: Stoppen einzelner DB-Container zur Demonstration der Ausfallsicherheit

Abschließende Worte

Das komplette Projekt hat eine Gesamtgröße von gerade einmal 271,9 MB (95,9 MB Applikation und 176 MB Datenbank). Über Docker Compose können auch noch andere Spezifikationen konfiguriert werden, wie der maximale zugewiesene Arbeitsspeicher der einzelnen Container, die maximal zu nutzenden CPU-Zyklen oder die maximal erlaubte Anzahl der CPU-Kerne.

Das Tutorial zeigt, wie wirklich kleine Container-Images erstellt werden können. Diese eignen sich als Microservices mit all ihren Vorteilen. Sie können schnell gestartet, einfach deployt und ausgetauscht werden. Außerdem wurde betrachtet, wie die Microservices über Docker Compose orchestriert und ausfallsicher konfiguriert werden können.

Wir hoffen, der Artikel hilft einigen Lesern bei der Arbeit und erweckt das Interesse daran, Docker einmal für Microservices selbst auszuprobieren.

Literatur und Links

[Dock1] Best practices for writing Dockerfiles, Docker Inc., https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

[Dock2] https://hub.docker.com/_/hola-mundo/

[Dock3] Node.js is a JavaScript-based platform for server-side and networking application, https://hub.docker.com/_/node?tab=description

[Dock4] A docker image base on alpine with mysql, https://hub.docker.com/r/wangxian/alpine-mysql