

Hochschule München  
University of Applied Sciences  
Fachbereich Informatik und Mathematik

# Docker Compose: Beispiel DB & UI mit IntelliJ IDEA und Maven

Seminararbeit

von

**Simon Symhoven**

Seminar DevOps und Cloud Computing

Professor:

Prof. Dr. Thorsten Zimmer

München, den 6. Juni 2020

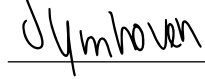
# Eidesstattliche Erklärung

Hiermit erkläre ich, Simon Symhoven, die vorliegende Seminararbeit selbstständig und nur unter Verwendung der von mir angegebenen Literatur verfasst zu haben.

Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Diese Arbeit hat in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

München, den 6. Juni 2020



---

SIMON SYMHOVEN

# Inhaltsverzeichnis

<b>Eidesstattliche Erklärung</b>	<b>ii</b>
<b>1 Motivation</b>	<b>1</b>
<b>2 Grundlagen</b>	<b>3</b>
2.1 Maven . . . . .	3
2.1.1 Erste Schritte mit Maven . . . . .	3
2.1.2 Der Maven Build Prozess . . . . .	5
2.2 Docker . . . . .	6
2.2.1 Erste Schritte mit Docker . . . . .	6
2.2.2 Das Docker-Plugin für IntelliJ IDEA . . . . .	8
<b>3 Ausgangssituation</b>	<b>10</b>
<b>4 Docker Compose: Container orchestrieren</b>	<b>12</b>
4.1 Erste Schritte mit Docker Compose . . . . .	12
4.2 Docker Container definieren . . . . .	12
4.3 Das Compose File . . . . .	12
4.4 Die wichtigsten Befehle . . . . .	16
<b>5 Fazit</b>	<b>17</b>
5.1 Zusammenfassung . . . . .	17
5.2 Ausblick . . . . .	17
<b>Literaturverzeichnis</b>	<b>21</b>

# 1 Motivation

Moderne Anwendungen bestehen üblicherweise aus mehreren Komponenten, die jeweils unterschiedliche Abhängigkeiten besitzen. Dabei bestehen nicht nur Abhängigkeiten innerhalb der Komponente selbst, sondern auch untereinander.

In folgender Seminararbeit wird eine Anwendung entwickelt und vorgestellt, welche aus drei Komponenten besteht wie in Abbildung 1.1 dargestellt. Es wird aufgezeigt, wie die Abhängigkeiten der Komponenten verwaltet werden können. Die Lösung basiert auf der Entwicklungsumgebung *IntelliJ IDEA*<sup>1</sup>, dem Framework *Maven*<sup>2</sup> und dem Container Management Tool *Docker*<sup>3</sup>, speziell *Docker Compose*.

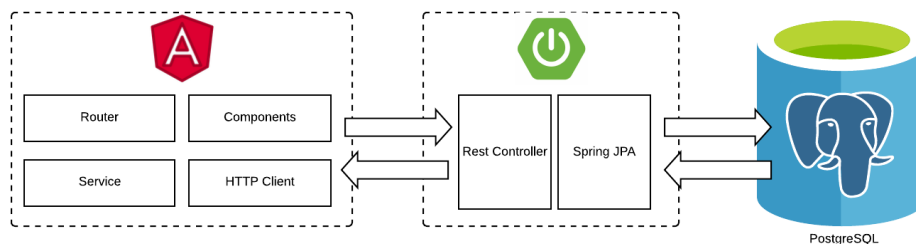


Abbildung 1.1: Eine moderne Architektur, bestehend aus drei Komponenten. *PostgreSQL* als Datenbank, *Spring Boot* im Backend und eine *Angular* Web-App im Frontend.

Die Datenbank stellt dabei die erste Komponente dar. Diese ist für die Persistenz der Daten zuständig und kommuniziert mit dem Server. Die zweite

<sup>1</sup><https://www.jetbrains.com/idea/>

<sup>2</sup><https://maven.apache.org>

<sup>3</sup><https://www.docker.com>

Komponente, die *Spring Boot* Anwendung (Server) nutzt auf der einen Seite die Java Persistence API um Objekte in die Datenbank zu schreiben und auszulesen und auf der anderen Seite stellt diese für die dritte Komponente eine Schnittstelle zur Verfügung. Ein Client, in diesem Fall eine *Angular* Web-App bildet dabei die dritte Komponente und nutzt die Schnittstelle in Form einer *REST-API* um mit dem Server zu kommunizieren und sich dort zu authentifizieren. Jede Komponente besitzt interne Abhängigkeiten. Vor allem die *Spring Boot* Applikation benötigt bestimmte Abhängigkeiten, wie z.B. Datenbanktreiber oder den Json Web Token für die Authentifizierung, die mit *Maven* verwaltet werden sollen. Externe Abhängigkeiten zwischen den Komponenten sollen in *Docker Containern* mit *Docker Compose* orchestriert werden.

Ziel ist es dabei, alle Komponenten so einfach und übersichtlich wie möglich verwalten und nutzen zu können. Zusätzlich wird das *Docker Plugin*<sup>4</sup> für *IntelliJ IDEA* das Absetzen der Befehle über die Kommandozeile ablösen.

**Hinweis:** Bei den eingebunden Code Dateien handelt es sich größtenteils um Ausschnitte. Für eine vollumfängliche Definition des Projekts wird auf das zu Grunde liegende Github Repository<sup>5</sup> verwiesen.

---

<sup>4</sup><https://plugins.jetbrains.com/plugin/7724-docker>

<sup>5</sup><https://github.com/simonsymhoven/skillbatz>

## 2 Grundlagen

### 2.1 Maven

#### 2.1.1 Erste Schritte mit Maven

*Maven* dient dem Projektmanagement von Softwareentwicklungsprojekten. Der Fokus liegt dabei auf Builds, Dokumentation, Reporting, Abhängigkeiten, Software Configuration Management (SCM) Releases und Distribution. Dabei sollen Produktivität und Wiederverwendbarkeit gesteigert werden. Als zentrale Steuerungsdatei dient die *pom.xml* (Project Object Model) wie in Abbildung 2.1 dargestellt, die in fünf Bereiche unterteilt ist: Koordinaten, Projektbeziehungen, Projektinformationen, Projekteinstellungen und Projektumgebung. Für diesen Anwendungsfall sind vor allem die Koordinaten und die Abhängigkeiten von zentraler Bedeutung (Horn, 2015b).

Listing 2.1: Ausschnitt aus dem Project Object Model der *Spring Boot* Komponente.

```
1
2 <groupId>de.simonsymhoven</groupId>
3 <artifactId>skillbatz-backend</artifactId>
4 <version>0.1.2-SNAPSHOT</version>
5 <name>skillbatz-backend</name>
6 <description>Backend project for Skillbatz</description>
7 <dependencies>
8   <dependency>
9     <groupId>org.springframework.boot</groupId>
10    <artifactId>spring-boot-starter</artifactId>
11  </dependency>
12 </dependencies>
```

```
13     <groupId>org.postgresql</groupId>
14     <artifactId>postgresql</artifactId>
15     <scope>runtime</scope>
16 </dependency>
17 <dependency>
18     <groupId>io.jsonwebtoken</groupId>
19     <artifactId>jjwt</artifactId>
20     <version>0.9.1</version>
21 </dependency>
22 </dependencies>
23
24 <build>
25     <plugins>
26         <plugin>
27             <groupId>org.springframework.boot</groupId>
28             <artifactId>spring-boot-maven-plugin</artifactId>
29         </plugin>
30     </plugins>
31 </build>
32
33 </project>
```

Als Koordinaten werden die fünf Informationsbestandteile bezeichnet, die ein Artefakt eindeutig identifizierenden. Oft sind vor allem die drei Wichtigsten gemeint: *groupId*, *artifactId* und *version*. Abhängigkeiten werden als *dependency* eingetragen, transitive Abhängigkeiten benötigen dabei nicht eingetragen zu werden, diese werden von *Maven* erkannt und verwaltet.

Über Lifecycles wie *package* kann mit dem entsprechenden Kommando *mvn package* der Source Code kompiliert werden und anschließend in eine ausführbare Datei verwandelt werden. Diese Lifecycles können unter dem Abschnitt *plugins* durch die dort definierten Plugins erweitern werden. Das *spring-boot-maven-plugin*, welches in diesem Projekt inkludiert ist, bietet dabei einen weiteren Lifecycle *mvn spring-boot:run* an, mit dem die *Spring Boot* Anwendung gestartet werden kann.

*Maven* verwaltet dabei die externen Abhängigkeiten in einem lokalen Reposi-

tory, welches unter  $\${user.home}/.m2/repository$  zu finden ist. Werden neue Abhängigkeiten hinzugefügt, wird zunächst das lokale Repository nach dieser Abhängigkeit durchsucht, bevor im Remote-Repository gesucht wird (Horn, 2015b).

### 2.1.2 Der Maven Build Prozess

Der *Maven* Build Prozess besteht aus zwei wesentlichen Teilen:

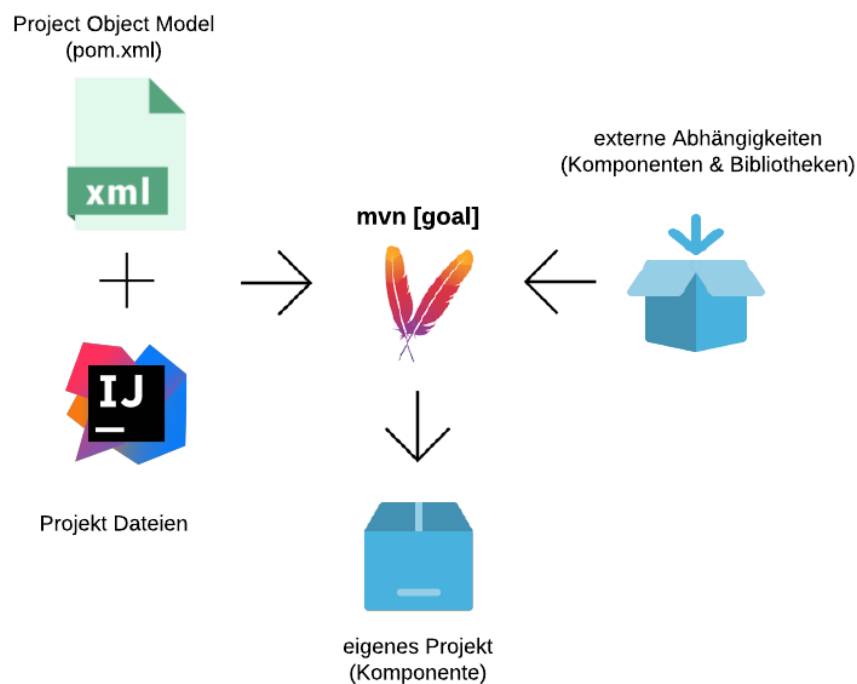


Abbildung 2.1: Der *Maven* Build Prozess.

Den Binärdateien, also den *IntelliJ IDEA* Projekt Dateien und der *pom.xml* und den externen Abhängigkeiten, die in dem Project Object Model definiert sind und von *Maven* in dem lokalen Repository verwaltet werden. Über bestimmte Kommandos, wie z.B. *mvn package* können diese beiden Bestandteile dann



zu einer eigenen Komponente zusammengefügt werden. Als Ergebnis erhält man eine ausführbare Datei, üblicherweise ein *jar*-Datei.

## 2.2 Docker

### 2.2.1 Erste Schritte mit Docker

*Docker* dient der Isolierung von Anwendungen, der Containervirtualisierung und der Vereinfachung der Anwendungsbereitstellung. Das *Docker Image* enthält dabei alle benötigten Komponenten wie Betriebssystem, Laufzeitumgebung oder Libraries. Das ausgeführte *Docker Image* wird als *Docker Container* bezeichnet. Das *Dockerfile* beinhaltet alle Befehle für die Erstellung des *Docker Images* (Horn, 2015a).

*Docker Container* können basierend auf einem selbst erstellten *Docker Image* aus einem *Dockerfile* oder basierend auf einem im Docker-Hub verfügbaren *Docker Image* erstellt und anschließend gestartet werden. Die PostgreSQL Datenbank kann dadurch, basierend auf dem offiziellen *Docker Image* von *postgres*<sup>1</sup> über den *docker run* Befehl ausgeführt werden:

Listing 2.2: *docker run* Befehl für den PostgreSQL Docker Container.

```
1 docker run --name dbpostgresql -e POSTGRES_PASSWORD=123 -p  
5432:5432 postgres
```

Die PostgreSQL Datenbank ist dann unter <https://localhost:5432> erreichbar. Der Parameter *--name dbpostgresql* dient dabei als Bezeichner für den *Docker Container*. Der Parameter *-e* oder *-env* spezifiziert Umgebungsvariablen. Der Zusatz *-p 5432:5432* sorgt dafür, dass der Port des Host Systems auf den Port des *Docker Containers* gemappt wird, wodurch die Anwendung von außen erreichbar wird. Zum Schluss wird der Name des *Docker Images* angegeben.

Die *Docker Container* der *Spring Boot* und *Angular* Anwendung basieren auf dem selbst erstellten *Dockerfile* der jeweiligen Anwendung.

<sup>1</sup>[https://hub.docker.com/\\_/postgres](https://hub.docker.com/_/postgres)

Listing 2.3: *Dockerfile* für die *Spring Boot* Komponente.

```
1 FROM maven AS MAVEN_BUILD
2 COPY pom.xml /build/
3 COPY src /build/src/
4 WORKDIR /build/
5 RUN mvn package -DskipTests
6
7 FROM openjdk:12
8 WORKDIR /spring-boot/
9 EXPOSE 8080
10 COPY --from=MAVEN_BUILD /build/target/skillbatz-backend-0.1.2-SNAPSHOT.jar /spring-boot/
11 ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom",
              "-jar", "skillbatz-backend-0.1.2-SNAPSHOT.jar"]
```

Listing 2.4: *docker build* Befehl für die *Spring Boot* Komponente.

```
1 docker build -t spring-boot-api .
```

Mit Listing 2.8 kann nun aus dem in Listing 2.3 dargestellten *Dockerfile* ein *Docker Image* erzeugt werden, das mit dem Parameter *-t spring-boot-api* getaggt wird. Zum Schluss wird das Verzeichnis des *Dockerfiles* angegeben, in dem Fall das aktuelle Verzeichnis mit einem „.“, da der Befehl im selben Verzeichnis ausgeführt wird.

Anschließend kann das *Docker Image* gestartet werden:

Listing 2.5: *docker run* Befehl für die *Spring Boot* Komponente.

```
1 docker run --name spring-boot-api -p 8080:8080 --link
  dbpostgresql:postgres spring-boot-api
```

Mit *--link dbpostgresql:postgres* kann dem *Docker Container* mitgeteilt werden, auf welche anderen *Docker Container* dieser zugreifen kann. Hier erhält der *Docker Container* der *Spring Boot* Anwendung Zugriff auf den Port der *PostgreSQL* Datenbank.

Analog ist mit dem *Dockerfile* der *Angular* Komponente zu verfahren:

Listing 2.6: *Dockerfile* für die *Angular* Komponente.

```
1 FROM node AS BUILD_STAGE
2 WORKDIR /app
3 COPY package*.json /app/
4 RUN npm install
5 COPY ./ /app/
6 RUN $(npm bin)/ng build --prod
7
8 FROM nginx
9 EXPOSE 4200
10 COPY ./nginx.conf /etc/nginx/nginx.conf
11 COPY --from=BUILD_STAGE /app/dist/skillbatz-frontend/ /usr/
    share/nginx/html
```

Aus dem *Dockerfile* wird zunächst mit

Listing 2.7: *docker build* Befehl für die *Angular* Komponente.

```
1 docker build -t angular-app-frontend .
```

ein *Docker Image* erstellt, welches dann gestartet werden kann:

Listing 2.8: *docker run* Befehl für die *Angular* Komponente.

```
1 docker run --name angular-app-frontend -p 4200:80 --link
    spring-boot-api:spring-boot-api angular-app-frontend
```

## 2.2.2 Das Docker-Plugin für IntelliJ IDEA

Jede Komponente wurde nun mit einem *Dockerfile* versehen oder kann basierend auf einem *Docker Image* gestartet werden. Das manuelle Verwalten der Befehle ist mühsam und fehleranfällig. Das *Docker Plugin* bietet die Möglichkeit diese Befehle in den *Run Configurations* innerhalb der *IntelliJ IDEA* Entwicklungsumgebung zu verwalten. Außerdem können *Docker Images* und auch *Docker Container* damit sehr leicht zusammengestellt und konfiguriert

werden, ohne den Befehl tatsächlich über die Kommandozeile eingeben zu müssen. Die drei Komponenten sind dann auf Knopfdruck verfügbar. Eine ausführliche Erläuterung des Funktionsumfangs führt an dieser Stelle zu weit und ist für den weiteren Verlauf der Arbeit nicht sachdienlich. Sofern *IntelliJ IDEA* nur in der Community Edition vorliegt bedarf es der manuellen Installation des *Docker Plugins*.

### 3 Ausgangssituation

Mit *Maven* werden die internen Abhängigkeiten, zumindest von der *Spring Boot* Komponente über die zentrale Steuerungsdatei, dem *Project Object Model* verwaltet. Jede Komponente ist in einem *Docker Container* virtualisiert. Nachdem somit alle Abhängigkeiten definiert sind, muss das System gestartet werden, d.h. jeder *Docker Container* ausgeführt werden.

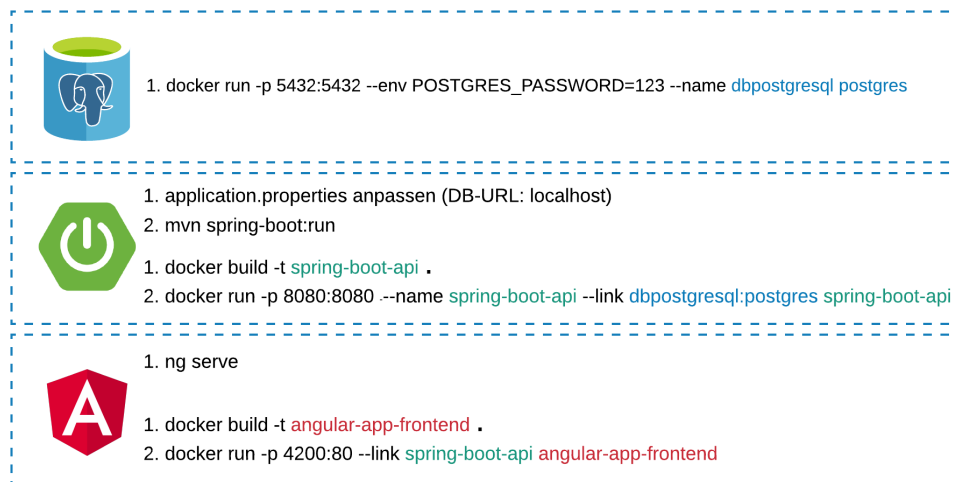


Abbildung 3.1: Überblick: Komponenten der Anwendung einzeln starten.

Dabei ist auf die richtige Reihenfolge zu achten. Zuerst ist die *PostgreSQL* Datenbank zu starten, dann die *Spring Boot* Anwendung und zuletzt der Client, die *Angular* Web-App, wie Abbildung 3.1 zeigt. Die manuelle Verwaltung und das Ausführen bzw. Stoppen der einzelnen *Docker Container* ist mühselig und unübersichtlich. Es wurde bereits gezeigt, wie das *Docker Plugin* dazu genutzt werden kann, diese manuelle Verwaltung über *Run Configurations* zu verein-

fachen, wodurch die Services auf Knopfdruck gestartet werden können. Dennoch muss die Konfiguration an mehreren Stellen vorgenommen werden, da die *Dockerfiles* an unterschiedlichen Stellen liegen. Die dem System zu Grunde liegenden Steuerungsdateien sind zerstreut, was die Verwaltung der gesamten Anwendung erschwert. *Docker Compose* bietet eine Möglichkeit die gesamte Anwendung, basierend auf den drei *Docker Containern* in einer einzigen Steuerungsdatei, dem *Compose File* zu definieren. Im folgenden Kapitel wird dieses Konzept grundlegend erläutert und auf die einzelnen Bestandteile des *Compose Files* eingegangen.

## 4 Docker Compose: Container orchestrieren

### 4.1 Erste Schritte mit Docker Compose

*Docker Compose* ist ein Tool um mehrere *Docker Container* zu konfigurieren und auszuführen. Das *Compose File* dient dabei als zentrale Steuerungsdatei, in der sämtliche Dienste einer Anwendung definiert werden. Das ausgeführte System, speziell die *Docker Container* werden dabei lediglich auf einem Host ausgeführt (Inc., 2020).

### 4.2 Docker Container definieren

Basierend auf den in Kapitel 2.2.1 definierten *Dockerfiles* wird jede Komponente in einem *Docker Container* virtualisiert.

Anschließend werden alle *Docker Container* in dem *Compose File* definiert. Jede Komponente wird als ein Service definiert, welcher auf Basis des zugehörigen *Dockerfiles* als *Docker Container* ausgeführt werden kann. Auf Basis dieses *Compose Files*, kann dann die gesamte Anwendung mit einem Knopfdruck gestartet werden.

### 4.3 Das Compose File

Jeder Service innerhalb des *Compose Files* hat diverse Eigenschaften, die *Docker* benötigt um das Image zu finden oder zu bauen. Dazu zählen vor allem

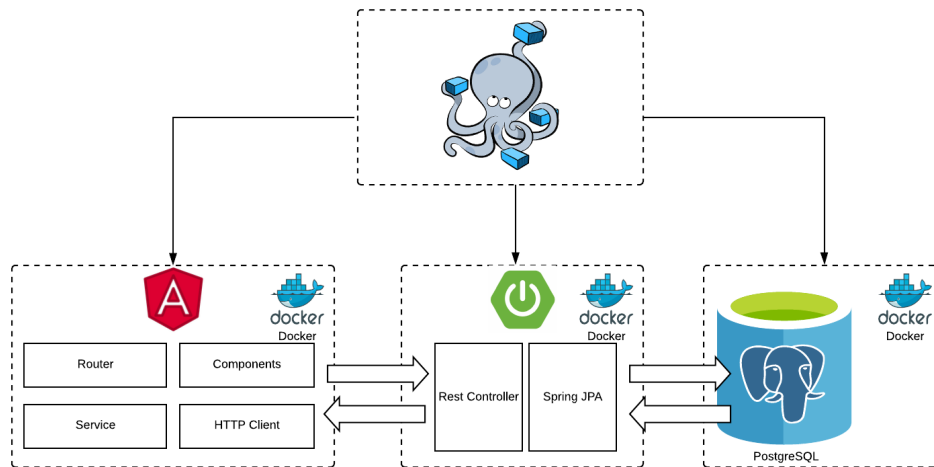


Abbildung 4.1: Compose File: Prozess der Erstellung.

Umgebungsvariablen, Ports, Verlinkungen in Form von Netzwerken und Volumes. Durch Volumes werden Verzeichnisse dem Host System geteilt.

Listing 4.1: Compose File für den Service dbpostgresql.

```

1 version: "3.7"
2
3 services:
4   dbpostgresql:
5     image: postgres
6     container_name: dbpostgresql
7     ports:
8       - "5432:5432"
9     environment:
10      - POSTGRES_PASSWORD=123
11     volumes:
12      - tmpvolume:/var/lib/postgresql
13     networks:
14      - backendNetwork

```

Das *Compose File* ist in der YAML-Syntax geschrieben. Auf der ersten Ebene werden dabei die vorhandenen Services des Systems aufgelistet. Der Ser-



vice *dbpostgresql* beschreibt dabei unsere Datenbank Komponente. Dieser Service basiert auf einem *Docker Image* aus dem Docker-Hub, welches mit dem Schlüsselwort *image* angegeben werden kann. Hier baut die Datenbank Anwendung auf dem offiziellen *Docker Image* von *postgres* auf. Mit *container\_name* kann dem *Docker Container* ein Name, z.B. *dbpostgresql* vergeben werden. Das Schlüsselwort *ports* öffnet die angegebenen Ports und verbindet diese mit dem Host System. Der Port 5432 des *Docker Containers* (rechts) wird auf dem Port 5432 (links) des Host Systems verfügbar gemacht. Mit *environment* werden Umgebungsvariablen festgelegt. Wie der Abschnitt *ports* wird hier eine Liste an Argumenten erwartet, welche eine Ebene eingerückt ist. Für die *PostgreSQL* Datenbank wird lediglich ein Passwort benötigt. Dieses wird mit *POSTGRES\_PASSWORD* definiert und mit dem Wert „123“ initialisiert. Die Konfiguration dieser Umgebungsvariablen hängen natürlich vom inkludierten *Docker Image* ab. Da beim Stoppen bzw. Löschen der *Docker Container* grundsätzlich alle Daten verloren gehen, besteht die Möglichkeit mit *volumes* ein Verzeichnis vom Host auf dem *Docker Container* verfügbar zu machen. Änderungen auf der einen Seite führen zu Änderung auf der anderen Seite. Durch das Einbinden eines Volumes bleibt die Datenkonsistenz erhalten und somit auch die Daten beim Neustart des *Docker Containers* erhalten. Die Kommunikation der *Docker Container* untereinander kann über den Abschnitt *networks* gewährleistet werden. Die Datenbank Komponente befindet sich im *backendnetwork*. Volumes sowie Netzwerke werden am Ende des *Compose Files* definiert.

Listing 4.2: Compose File für den Service *springapi*.

```
1 springapi:
2   build: ./skillbatz-backend/
3   container_name: springapi
4   ports:
5     - "8080:8080"
6   depends_on:
7     - dbpostgresql
8   networks:
9     - backendNetwork
10    - frontendNetwork
```

Den nächsten Service bildet die *Spring Boot* Komponente. Anders als die Datenbank Anwendung basiert dieser *Docker Container* auf dem eigens erstellten *Dockerfile*. Deshalb kann mit dem Schlüsselwort *build* der Pfad zu dem Ordner, in dem das entsprechende *Dockerfile* liegt, angegeben werden. Die Schlüsselwörter *image* und *build* sind dabei exklusiv zu verwenden! Mit *depends\_on* kann eine Abhängigkeit beim Starten der Services festgelegt werden. Somit wird der Service *springapi* erst gestartet wenn der Service *dbpostgresql* erfolgreich ausgeführt und gestartet wurde.

Listing 4.3: *Compose File* für den Service *angular*.

```
1  angular:
2    build: ./skillbatz-frontend/
3    container_name: angular-webapp
4    ports:
5      - "4200:80"
6    depends_on:
7      - springapi
8    networks:
9      - frontendNetwork
10
11 volumes:
12   tmpvolume:
13
14 networks:
15   backendNetwork:
16   frontendNetwork:
```

Im letzten Abschnitt können nun die in den Services benutzten Volumes und Netzwerke definiert werden. Die Komponenten der Anwendung kommunizieren in zwei Netzwerken, dem *frontendNetwork* und *backendNetwork* mit einander und können somit gegenseitig auf die Ports zugreifen. Der Name des Services auf der ersten Ebene dient dabei als Referenz. In der Docker Machine wäre der Service der Datenbank also über *http://dbpostgresql:5432* erreichbar. Eine Verlinkung der Container untereinander wie in Kapitel 2.2.1 gezeigt, ist mit *networks* nicht mehr nötig. (Öggl und Kofler, 2020)

## 4.4 Die wichtigsten Befehle

Unabhängig von dem *Docker Plugin* bietet *Docker Compose* Kommandos an, die über die Kommandozeile ausgeführt werden können:

Kommando	Funktion
<code>docker-compose create</code>	Services erstellen
<code>docker-compose start</code>	Services starten
<code>docker-compose stop</code>	Services stoppen
<code>docker-compose pause</code>	Services pausieren
<code>docker-compose unpause</code>	Services aufwecken
<code>docker-compose ps</code>	Container auflisten
<code>docker-compose images</code>	Images auflisten
<code>docker-compose up</code>	Services erstellen und starten
<code>docker-compose down</code>	Services stoppen und löschen (inkl. Networks, Images, Volumes)

Tabelle 4.1: Die wichtigsten *Docker Compose* Befehle und deren Funktion (Inc., 2020).

Mit *docker compose up* werden also die *Docker Images* erzeugt, die Netzwerke und Volumes konfiguriert und die *Docker Container* gestartet. Das komplette System ist innerhalb kürzester Zeit über ein Kommando verfügbar.

## 5 Fazit

### 5.1 Zusammenfassung

Das manuelle Verwalten eines größeren Systems aus mehreren Services wird schnell unübersichtlich und kompliziert. Es wurde gezeigt, wie komplexe Anwendungen bestehend aus mehreren Komponenten schnell und einfach orchestriert und verwaltet werden können. *Docker* bietet dabei die Möglichkeit jede Komponente in einen *Docker Container* auszulagern. Allerdings wird auch die Verwaltung der *Docker Container* mit zunehmender Komplexität der Anwendung umso schwieriger und übersichtlicher. Mit *Docker Compose* ist es möglich das gesamte System, also innerhalb einer Datei, dem *Compose File* zu definieren und zu verwalten. Die Applikation und die Abhängigkeiten der einzelnen Komponenten lassen sich somit strukturiert abbilden. Das *Docker Plugin* bietet weiterhin die Möglichkeit die gesamte Anwendung mit einem Knopfdruck zu starten und zu verwalten.

### 5.2 Ausblick

Als weiteren Service kann mit einem *Load Balancer*, z.B. mit *NGINX*<sup>1</sup> die Anwendung skaliert werden, d.h. mehrere Container für ein Image erzeugt werden, die die Anfragen an den Service abwechselnd verarbeiten. Im nächsten Schritt kann mit *Docker Swarm* die Ausfallsicherheit gewährleistet werden, *Docker Container* auf mehreren Hosts ausgeführt werden und die Anwendung in Produktion überführt werden. An dieser Stelle tritt Docker, speziell *Docker*

---

<sup>1</sup><https://www.nginx.com>

*Swarm* in große Konkurrenz mit *Kubernetes*<sup>2</sup> oder *Mesos*<sup>3</sup>. Die Zeit wird zeigen, welches Tool sich durchsetzen wird.

---

<sup>2</sup><https://kubernetes.io/de/>

<sup>3</sup><https://mesos.apache.org>

# Tabellenverzeichnis

4.1	Die wichtigsten <i>Docker Compose</i> Befehle und deren Funktion (Inc., 2020). . . . .	16
-----	---	----

# Abbildungsverzeichnis

1.1	Eine moderne Architektur, bestehend aus drei Komponenten. <i>PostgreSQL</i> als Datenbank, <i>Spring Boot</i> im Backend und eine <i>Angular</i> Web-App im Frontend. . . . .	1
2.1	Der <i>Maven</i> Build Prozess. . . . .	5
3.1	Überblick: Komponenten der Anwendung einzeln starten. . . .	10
4.1	Compose File: <i>Prozess der Erstellung</i> . . . . .	13

# Listings

2.1	Ausschnitt aus dem Project Object Model der <i>Spring Boot</i> Komponente. . . . .	3
2.2	<i>docker run</i> Befehl für den <i>PostgreSQL Docker Container</i> . . . . .	6
2.3	<i>Dockerfile</i> für die <i>Spring Boot</i> Komponente. . . . .	7
2.4	<i>docker build</i> Befehl für die <i>Spring Boot</i> Komponente. . . . .	7
2.5	<i>docker run</i> Befehl für die <i>Spring Boot</i> Komponente. . . . .	7
2.6	<i>Dockerfile</i> für die <i>Angular</i> Komponente. . . . .	8
2.7	<i>docker build</i> Befehl für die <i>Angular</i> Komponente. . . . .	8
2.8	<i>docker run</i> Befehl für die <i>Angular</i> Komponente. . . . .	8
4.1	<i>Compose File</i> für den Service <i>dbpostgresql</i> . . . . .	13
4.2	<i>Compose File</i> für den Service <i>springapi</i> . . . . .	14
4.3	<i>Compose File</i> für den Service <i>angular</i> . . . . .	15



# Literaturverzeichnis

- [Horn 2015a] HORN, T.: *Docker*. <https://www.torsten-horn.de/techdocs/Docker.html>. 2015
- [Horn 2015b] HORN, T.: *Maven 3.5*. <https://www.torsten-horn.de/techdocs/maven.htm>. 2015
- [Inc. 2020] INC., Docker: *Overview of Docker Compose*. <https://docs.docker.com/compose/>. 2020
- [Öggl und Kofler 2020] ÖGGL, B. ; KOFLER, M.: *Docker - Das Praxisbuch für Entwickler und DevOps-Teams*. Rheinwerk Computing, Januar 2020