

# FEEDBACKSYSTEM - DOKUMENTATION

Softwaretechnik Projekt von Philipp Bykow, Vladyslav  
Sokyrskyy und Benjamin Manns

## Einleitung

Für verschiedene Kurse werden Abgaben als Klausurvorleistung verlangt. Für die automatische Überprüfung solcher Hausaufgaben wurden, zumindest an der THM, verschiedene Abgabesysteme von den Dozenten entwickelt. Zum Teil klappen diese besser oder schlechter, können aber vor allem nur von dem Dozenten selbst gewartet werden und leider dann auch nicht so einfach für andere Dozenten wiederverwendbar.

Aus diesem Problem heraus entsteht die Fragestellung nach einem generischen Abgabesystem, welches erlaubt eigene Testsysteme für die verschiedensten Anwendungsgebiete zu entwickeln, sodass diese einfacher von anderen Dozenten genutzt werden können.

Man wünscht sich also eine Weboberfläche, wo ein Student seine Kurse sieht, zu den einzelnen Kursaufgaben entsprechend Abgaben einreichen kann und ein Feedback der Auswertung erhält.

Dozenten und Tutoren können die Kurse verwalten, Aufgaben hinzufügen usw. Das entscheidende ist, dass die Tests generisch laufen und von der Weboberfläche entkoppelt sind. Die Webseite des „Feedback Systems“ weiß nur, dass Aufgabe A zu dem Testsystem T getestet werden muss und schickt dies dahin. Wie der Test läuft ist in dem jeweiligen Testsystem hinterlegt und wird dort bearbeitet.

Es muss dann allerdings eine allgemeine Schnittstelle existieren, sodass auch später hinzugefügte Testsysteme einwandfrei mit dem Webservice interagieren können

Die Umsetzung soll mit aktuellen Web Technologien umgesetzt werden. Das heißt es soll möglich sein das ganze System zu skalieren, falls verschiedene Test sehr lange brauchen oder einfach eine große Menge an Studenten auf die Webseite zugreifen möchten. Lässt man die einzelnen Komponenten des „Feedback System“ beispielsweise in Docker Containern laufen, so wird das möglich gemacht.

Zunächst wollen wir uns aber anschauen, aus welchen Komponenten das „Feedback System“ besteht und wie diese zusammenarbeiten. Außerdem schauen wir uns an, was Docker ist

und wie man diese Komponenten in Docker zum Laufen bringen kann und eine Skalierung einrichtet.

Der Wunsch ist, dass dieses System einfach erweiterbar bleibt, dafür wollen wir das Konzept verteilter Systeme anwenden und die Software in Teilbereiche untergliedern.

Um die verteilten Systeme (Microservices) miteinander kommunizieren zu lassen, benutzen wir Apache Kafka.

## Apache Kafka

Kafka ist ein Message Broker, das meint, eine zentrale Stelle, an die man nach Themen geordnet Nachrichten schicken kann und auch empfängt. Ein Thema nennt man einfach „Topic“, eine Nachricht an Kafka zu schicken übernimmt der Producer und der Consumer registriert sich für gewisse Topics und holt sich neue Nachrichten ab.

Kafka gibt an extrem Ausfallsicher zu sein und die Daten redundant zu speichern. Es behält auch seine hohe Verfügbarkeit, wenn auch die Auslastung sehr hoch ist.

So ein Topic besteht aus einer Menge von Partitionen, für jedes Topic werden die Partitionen auf die verfügbaren Broker verteilt. Kafka ist ausgelegt, dass es in einem Cluster mit mehreren Brokern (mindestens einem) betrieben wird, sodass so eine Aufteilung möglich ist. Ein Broker kann mehrere Partitionen zu einem Thema verwalten, da die Anzahl an Partitionen in der Regel die Anzahl der verfügbaren Broker übersteigt.

Eine einzelne Partition ist im Prinzip ein fortlaufender Commit Log, also einmal geschriebene Daten bleiben unveränderlich. Jede Nachricht, die angehängt wird, bekommt eine innerhalb des Logs eindeutige und fortlaufende Offset, beginnend bei 0. Somit ist eine gesendete Nachricht durch das Tripel: (Topic, Partition, Offset) eindeutig bestimmbar.

Eine Nachricht, die an Kafka geschickt wird, ist ein „Schlüssel-Werte-Paar“. Diese Nachricht weist Kafka einer Partition zu. Dies wird aus dem Schlüssel berechnet (Hash-Verfahren) oder

wenn kein Schlüssel angegeben wurde, per Round-Rubin. Wie man seine Nachrichten codiert ist hingegen einem selbst überlassen.

Registriert sich ein Consumer, wird er einer „Consumer Group“ zugeordnet, dies geschieht, wenn man nichts einstellt automatisch. Eine „Consumer Group“ empfängt alle Nachrichten von allen Partitionen eines Topics. Gibt es nur einen Consumer innerhalb dieser Gruppe empfängt dieser alle Nachrichten, gibt es mehrere werden die Partitionen aufgeteilt.

Um dieses System Ausfallsicher zu machen, gibt für jede Partition eines Topics einen Leader, der Broker, der Nachrichten empfangen und senden darf und eine gewisse Zahl an „Followern“, die den aktuellen Log mitlesen. Fällt ein Leader aus, findet eine Neuwahl statt. Probleme können auftreten, wenn Leader auf einem zu alten Stand sind.

Genutzt wird Kafka in diesem Projekt, zum Beispiel, dass eine Benutzerabgabe zu dem entsprechenden Testsystem weitergeleitet wird. Das Testsystem wiederum arbeitete alle Nachrichten ab, die an ihn adressiert sind und meldet seinerseits das Testergebnis an Kafka, welches dann in die Datenbank des Webservices zu der Entsprechenden Abgabe gespeichert wird.

Folgende Services bilden das Feedback-System:

- Webservice, geschrieben in SpringBoot+Scala (Webserver) und Angular (Frontend).
- Kafka Message Broker
- Testsysteme, konkret: Skript-Checker (Bash, PHP), SQL-Checker

Jeder Service, wie auch der Webservice, haben ihre eigene Datenbank zur Zwischenspeicherung, bzw der Webservice stellt die Daten dann über die Webseite zur Verfügung.

Kafka-Topics Übersicht

Jedes Testsystem hat einen eindeutigen Namen, dieser wird bei der Registrierung des Systems bei dem Webservice angegeben. Diese ID muss bei allen Interaktionen von dem Webservice mit dem Testsystem konsistent verwendet werden.

Die Beschreibung verwendet das Testsystem mit der ID „sqlchecker“ als Beispiel.

Beschreibung für Kafka Nachrichten von dem Webservice zu den Testsystemen.

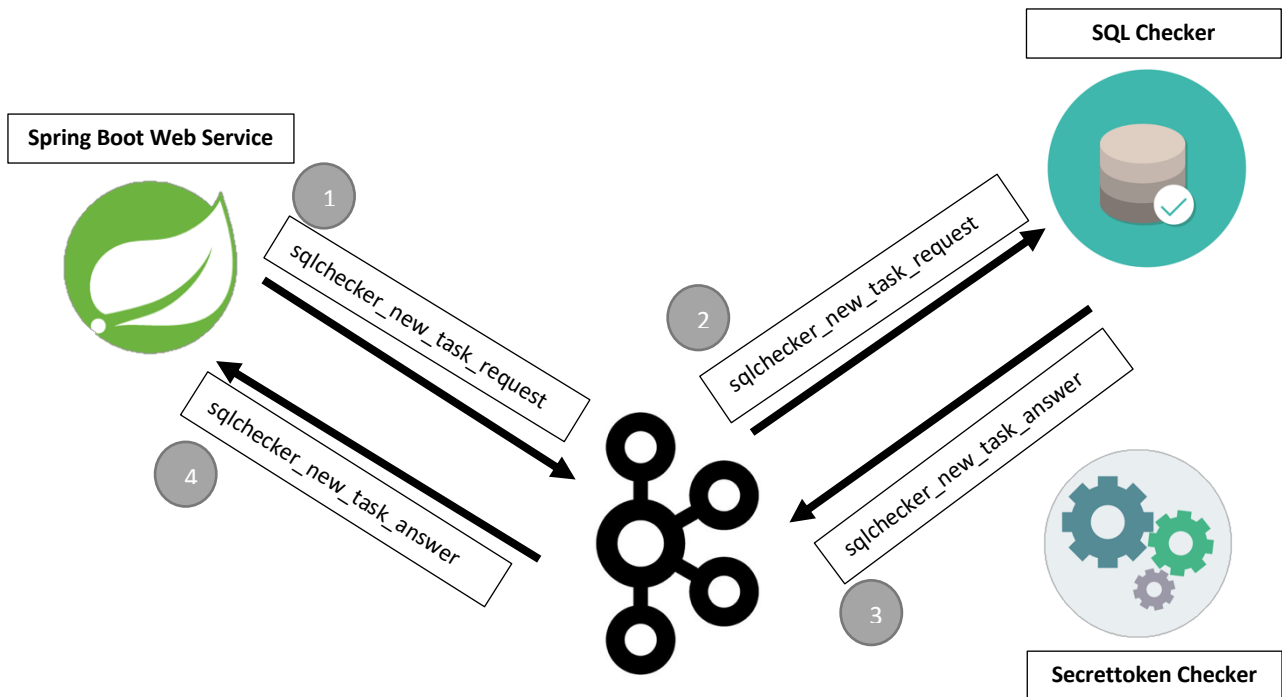
Spring Boot Webservice	Testsystem
SEND	LISTEN
<p>Topic: *_new_task_request</p> <p>Content (JSON):</p> <ul style="list-style-type: none"> <li>- testfile_url</li> <li>- taskid</li> <li>- jwt_token</li> </ul> <p>Wird benutzt beim Anlegen und Anpassen eines Tests und schickt die Testdatei an das Testsystem.</p> <p><i>Beispiel:</i></p> <p>Topic: sqlchecker_new_task_request</p> <p>Content: {</p> <pre>"jwt_token":"eyJhb...", "taskid":"12", "testfile_url":"https://localhost:/.../tasks/12/..."</pre> <p>}</p>	<p>Topic: *_new_task_request</p> <p>Der <i>jwt_token</i> wird zum Herunterladen der Datei unter <i>testfile_url</i> benötigt</p> <p><i>Beispiel:</i></p> <p>Topic: sqlchecker_new_task_request</p>
<p>Topic: *_check_request</p> <p>Content (JSON):</p> <p>Entweder:</p> <ul style="list-style-type: none"> <li>- fileurl</li> </ul>	<p>Topic: *_check_request</p> <p>Der <i>jwt_token</i> wird zum Herunterladen der Datei unter <i>fileurl</i></p>

<ul style="list-style-type: none"> <li>- jwt_token</li> <li>- submissionid</li> <li>- submit_typ = file</li> <li>- taskid</li> <li>- userid</li> </ul> <p>Oder:</p> <ul style="list-style-type: none"> <li>- data</li> <li>- submissionid</li> <li>- submit_typ = data</li> <li>- taskid</li> <li>- userid</li> </ul> <p>Je nach dem welcher <i>submit_typ</i> gewählt wurde, werden andere Parameter erwartet</p> <p><i>Beispiel:</i></p> <p>Topic: sqlchecker_check_request</p> <p>Content: {</p> <pre>"fileurl":"https://.../tasks/90/files/submissions...",   "jwt_token":"eyJhb...",   "submissionid":"213",   "submit_typ":"file",   "taskid":"90",   "userid":"xxx"</pre> <p>}</p>	<p>benötigt. Wird nur ein String (<i>data</i>) gesendet, ist ein Herunterladen natürlich nicht notwendig.</p> <p>Diese Kafka Nachricht triggert den Test-Prozess, der am Ende eine Kafka Nachricht mit dem Ergebnis des Tests auslöst.</p> <p><i>Beispiel:</i></p> <p>Topic: sqlchecker_check_request</p>
---	---

Beschreibung für Kafka Nachrichten von den Testsystemen zu dem Webservice.

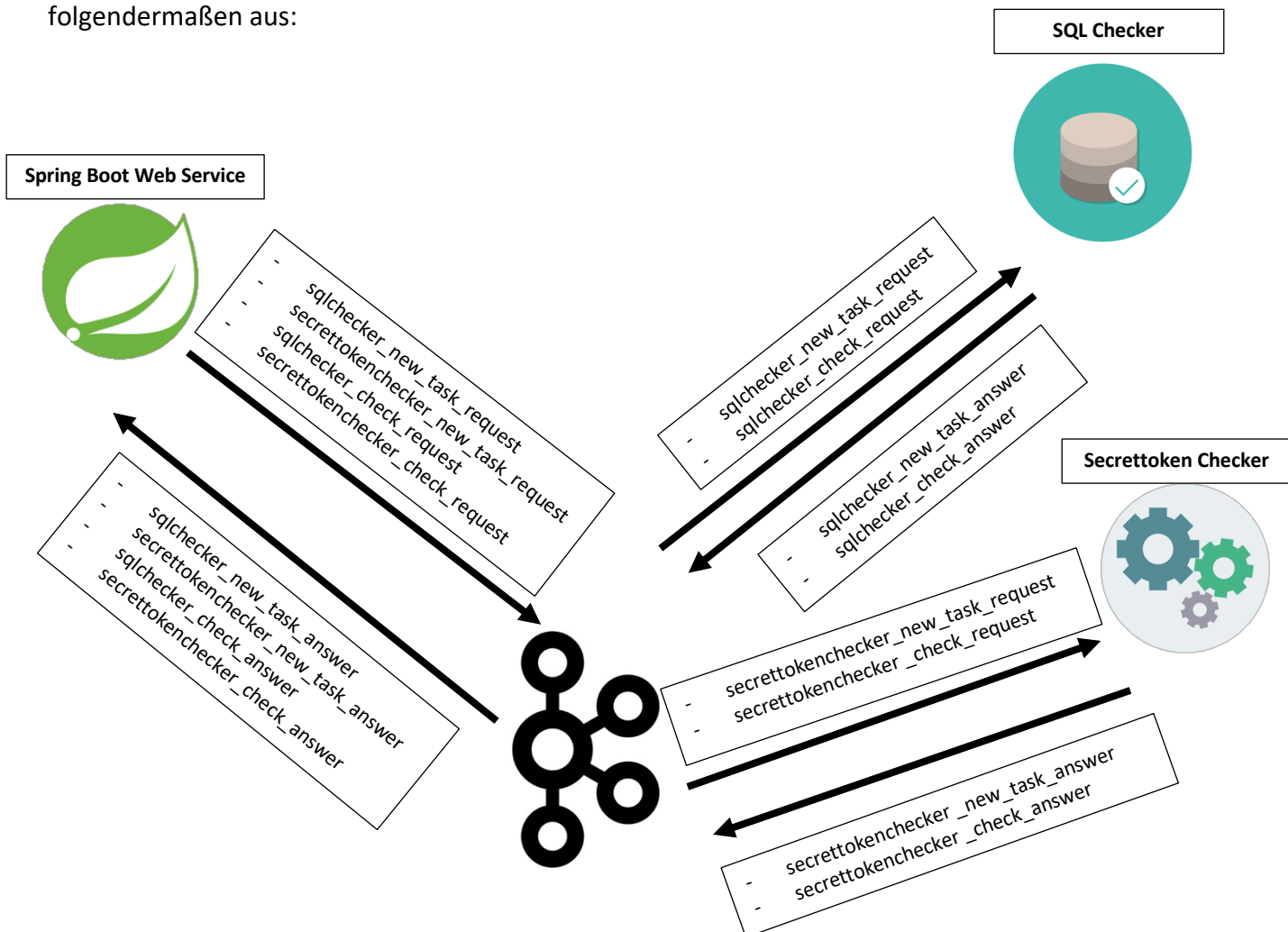
Testsystem	Spring Boot Webservice
SEND	LISTEN
<p>Topic: *_new_task_answer</p> <p>Content (JSON):</p> <ul style="list-style-type: none"> <li>- accept (true, false)</li> </ul>	<p>Topic: *_new_task_answer</p>

<ul style="list-style-type: none"> <li>- error (etwaige Fehlermeldung)</li> <li>- taskid</li> </ul> <p><i>Beispiel:</i></p> <p>Topic: sqlchecker_check_answer</p> <p>Content: {</p> <pre>"accept":false, "error": "Please provide a file called db.sql", "taskid":90 }</pre>	<p>Der Webservice speichert die Info zu der entsprechenden Aufgabe in die Datenbank.</p>
<p>Topic: *_check_answer</p> <p>Content (JSON):</p> <ul style="list-style-type: none"> <li>- passed (0=fail,1=passed)</li> <li>- userid</li> <li>- taskid</li> <li>- submissionid</li> <li>- data (Fehlermeldungen, Infos)</li> <li>- exitcode</li> <li>-</li> </ul> <p>Nachdem eine Abgabe ausgewertet wurde, wird das Ergebnis an den Webservice zurückgeschickt.</p> <p><i>Beispiel:</i></p> <p>Topic: sqlchecker_check_answer</p> <p>Content: {</p> <pre>"exitcode":0, "userid":55, "taskid":90, "submissionid":1034 }</pre>	<p>Topic: *_check_answer</p> <p>Die Auswertung des Testsystems wird in der verbundenen Datenbank hinterlegt und dem Anwender per Webseite sichtbar gemacht.</p>



Beispielhafte Kommunikation, wie der Web Service eine neue Aufgabe an ein Testsystem über Apache Kafka sendet, in dem Fall dem SQL Checker, und eine Antwort erhält, ob die gesendete Datei valide ist. Die Kommunikation läuft asynchron.

Die komplette Übersicht aller „Topics“, auf die gesendet und auch gehört wird sehe folgendermaßen aus:





## Spring Boot

Das Spring Boot Framework (Vereinfachung / Nachfolger von Spring) ist ein JVM Framework und erlaubt es Java Web Anwendungen sehr einfach zu entwickeln. Ein paar Zeilen Code reichen für eine *Hello World* Anwendung aus. Spring Boot nutzt viele Java Annotations was den Code deutlich entschlackt und unterstützt auch Groovy.

```
@RestController
@RequestMapping(path = Array("/api/health"))
class HealthService {
    /**
     * @return A static message: Alive of everything is okay.
     */
    @RequestMapping(value = Array("/beat"))
    def getBeat(): String = "Alive!"
}
```

*(Beispiel für eine REST Endpoint, geschrieben in Scala.)*

Wir haben das Spring Framework genutzt und die Anwendung in Scala geschrieben.

Nutzt man Spring Boot und Java / Scala wird es zusammen mit einem Paketmanager, in unserem Fall *Gradle* betrieben. Da zum einen viele Abhängigkeiten vorhanden sind, zum anderen auch das Deployen erst richtig funktioniert. Man entwickelt und debuggt an seiner Spring Boot Application und kann am Ende eine lauffähiges JAR Programm bereitstellen, was Tomcat Server und weitere Tool inklusive. Dieses Konzept macht Spring Boot auch beliebt um Microservices zu schreiben.

Konfiguriert wird Spring Boot über *.properties* – Dateien, wie man das auch von Apache Tomcat kennt.

In unserem Fall musste noch etwas zusätzliche Konfigurationsarbeit geleistet werden, da Spring von Hause aus nicht eingerichtet ist um vollständig kompatibel mit Scala zu sein-

Wir haben das Spring / Spring Boot Framework genutzt um für das „Feedback System“ eine REST Schnittstelle zu entwickeln, auf die bisher nur von einer Angular Webapp zugegriffen wird, aber auch von Mobilien Apps oder anderen Diensten verwendet werden könnte.

## Travis-CI und Unittests

Travis (<https://travis-ci.com/>) dient der kontinuierlichen Bereitstellung von Software aber auch des Testens. Man kann es leicht in sein GIT-Repository integrieren. Wir nutzen Travis um alle Scala Anwendungen zu bauen und den Codestyle zu checken, die Anwendung in Docker Images zu packen und den Webservice zu testen.

Travis nutzt Linux und MacOSX Systeme um Software für die unterschiedlichsten Plattformen zu bauen und Tests durchzuführen. Für open Source Projekte ist die Nutzung kostenlos.

**.travis.yml** Datei baut die Anwendungen und startet das Python Testskript.

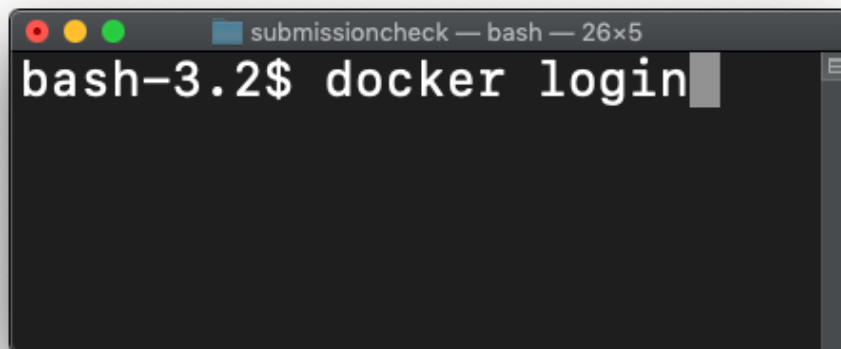
```
1  language: java
2
3  sudo: required
4
5  services:
6  - docker
7
8  python:
9  - "3.4"
10
11 before_install:
12 - sudo apt-get update
13 - sudo apt-get -y install python3-pip python-dev
14 - python3 -V
15 - pip3 -V
16 - sudo bash py-test/travis-fix.sh
17
18 install:
19 - sudo pip3 install requests_toolbelt
20
21 jdk:
22 - oraclejdk9
23
24 script:
25 - docker-compose down
26 - ./gradlew check dist
27 - ./gradlew dist
28 # Hack to debug docker and run python
29 - (docker-compose up --build) & (sleep 120 && python3 py-test/request.py)
30 #- ./gradlew ws:run --stacktrace &
31
32 before_cache:
33 - rm -f $HOME/.gradle/caches/modules-2/modules-2.lock
34 - rm -rf $HOME/.gradle/caches/*/plugin-resolution/
35
36 cache:
37 directories:
38 - $HOME/.gradle/caches/
39 - $HOME/.gradle/wrapper/
```

## Bereitstellen der Anwendung in Docker Images

Das Deployen auf Dockerhub kann mit Travis ebenso durchgeführt werden, sodass jede neue Änderung oder auch nur jedes neue Tag auf Dockerhub verfügbar gemacht wird.

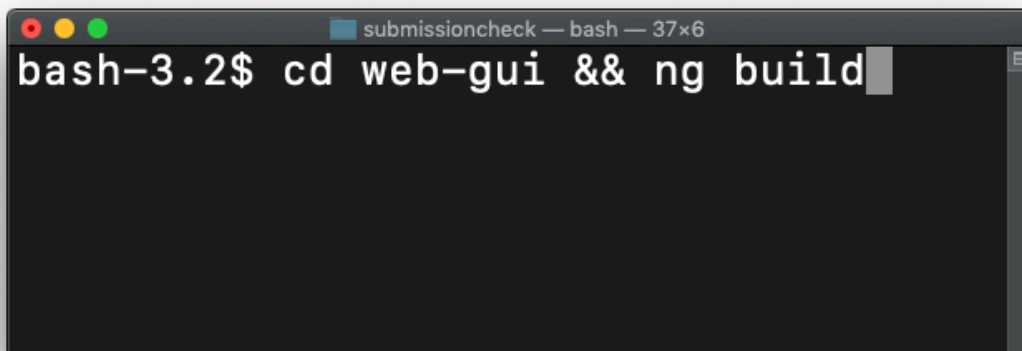
Wie das im Einzelnen geht soll hier erläutert werden. Im Prinzip sind 4 Schritte nötig, zuvor muss man aber einen Docker Hub Account haben. Außerdem sollte man in den Branch *master\_docker\_images* wechseln, da dort einige Konfigurationen vorgenommen worden sind, die die Containerisierung ermöglichen.

Man meldet sich über die Konsole mittels *docker login* an:

A terminal window with a dark background. The title bar shows 'submissioncheck — bash — 26x5'. The prompt is 'bash-3.2\$' and the command 'docker login' is being entered, followed by a cursor. There is a small icon in the top right corner of the terminal window.

```
bash-3.2$ docker login
```

Nun wechselt man in den web-gui Ordner und packt die Angular Anwendung mit diesem Befehl: *cd web-gui && ng build*.

A terminal window with a dark background. The title bar shows 'submissioncheck — bash — 37x6'. The prompt is 'bash-3.2\$' and the command 'cd web-gui && ng build' is being entered, followed by a cursor. There is a small icon in the top right corner of the terminal window.

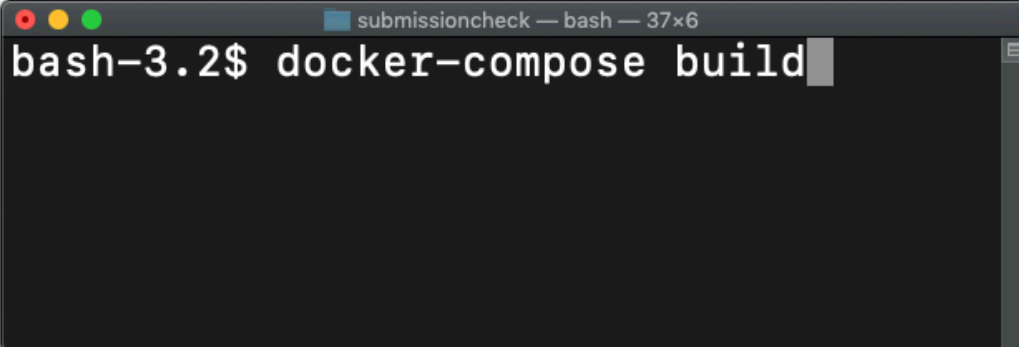
```
bash-3.2$ cd web-gui && ng build
```

Ist die Anwendung gebaut (evt müssen node Pakete installiert werden), baut man alle Scala Anwendungen, das meint die Testsysteme wie auch Spring Boot: `cd .. && ./gradlew dist`.



```
submissioncheck — bash — 37x6
bash-3.2$ cd .. && ./gradlew dist
```

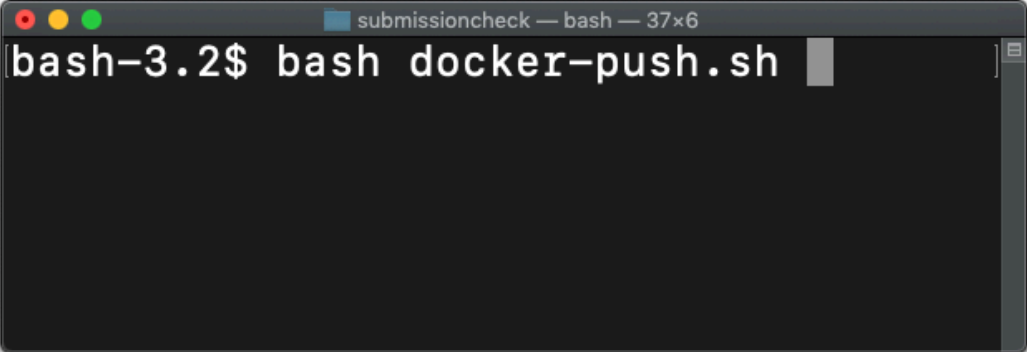
Das Erstellen der Docker Images ist nun ein einfacher Befehl: `docker-compose build`



```
submissioncheck — bash — 37x6
bash-3.2$ docker-compose build
```

Ist dies abgeschlossen, verwende man das kleine Skript `docker-push.sh` und führe es aus:

`bash docker-push.sh`



```
submissioncheck — bash — 37x6
bash-3.2$ bash docker-push.sh
```

Nun kann man die Images unter seinem Docker Hub Account aufrufen und weiterverwenden, wo immer man möchte.

## Installation / Inbetriebnahme der Software

Möchte man das Feedbacksystem in Betrieb nehmen, dann muss man sich die nötigen Docker Images benutzen und vernetzen. Dazu verwende man die vorgegebene *docker-compose.yml* Datei. (Verfügbar im GIT Repo).

Diese erwartet die Konfigurationsdateien der Scala Anwendungen und ein initiales SQL, welches das Datenbank Schema und initiale Benutzer / Kurse enthält.

Aktuell enthält die *init.sql* Datei Beispiel Kurse und Benutzer, diese werden für die definierten Travis Python Tests benötigt. Startet man die MySQL Datenbank zum ersten Mal, werden diese Dummy-Daten importiert. Gerne können die Benutzer und Kurse dann entsprechend gelöscht und aufgeräumt werden.

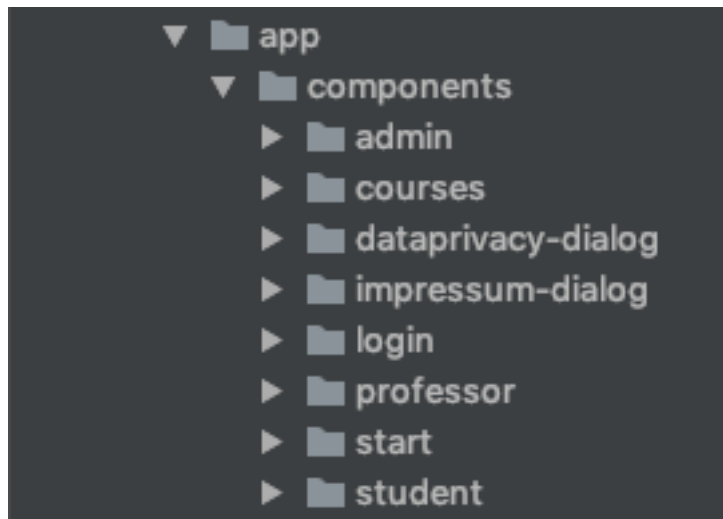
Für den Secrettoken Checker muss die Variable *HOST\_UPLOAD\_DIR* gesetzt werden. Das ist der Root-Pfad des Upload Ordners auf dem Host System.

Die aktuelle Version der Docker Images lautet 0.0.8 und wird bei Minor Changes immer hochgezählt.

## Webseite / Frontend / Angular

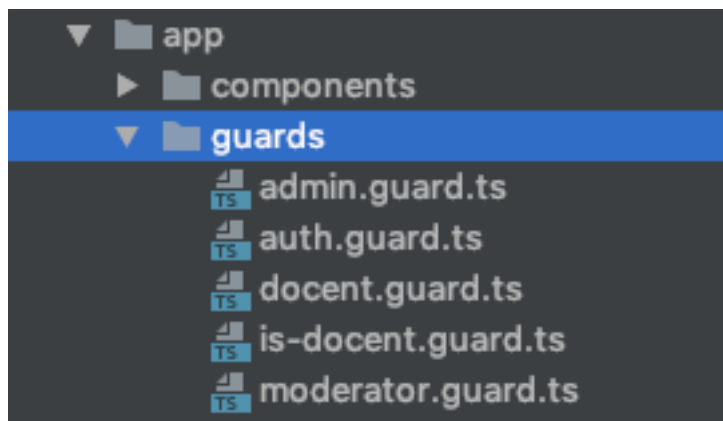
Angular ist ein Webframework entwickelt von Google. Es ist komplett in Typescript geschrieben und wird auch in dieser Programmiersprache genutzt. Angular bietet einen Modularen Aufbau, was eine Wiederverwendung von einzelnen Teilen des Programmes ermöglicht. Eine Angular App besteht aus mehreren Einzelteilen die zusammen eine ganze Website ergeben. Darunter fallen unter anderem Komponenten, Guards, Services und Modules. Komponenten bestehe dabei immer aus einer HTML, CSS und Typescript Datei.

Das Feedbacksystem besteht aus folgenden Komponenten:

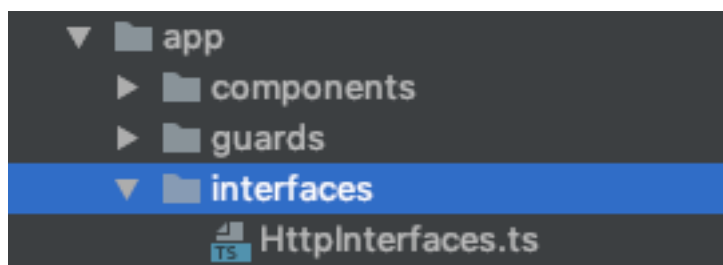


Diese Komponenten besitzen noch weitere Unterkomponenten, die zur Struktur der einzelnen Komponente beitragen. Eine Komponente kann eine komplette Seite, oder ein Teil einer Seite repräsentieren.

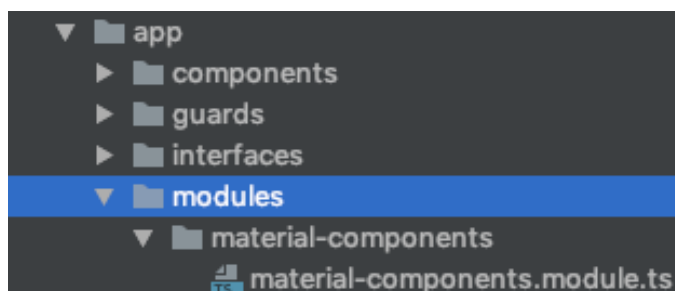
Guards dienen zur Sicherheit des internen Angular Routings. Im Feedbacksystem wird zum Beispiel überprüft, ob ein User angemeldet ist. Weiterhin gibt es Guards die überprüfen, welche Rolle der User besitzt und ob er die Erlaubnis hat eine gewisse Route aufzurufen.



Im Ordner Interfaces gibt es genau eine Datei, welche alle Interfaces für die einzelnen API aufrufe definiert. Da in der kompletten Web App Typescript benutzen wird, ist es sinnvoll die Möglichkeit der Typisierung zu nutzen. Somit wird es gestattet, direkt einzelne Werte zu überprüfen und auf sie zuzugreifen.

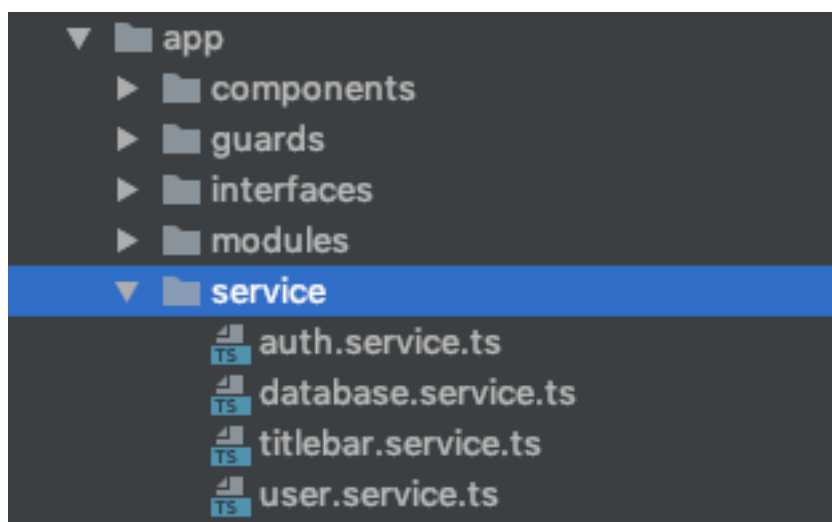


Unter dem Ordner Modules fällt eine Datei, die für den Import aller benutzten Material-Design Komponenten zuständig ist. Möchte man nun weitere Material-Design Komponenten benutzen, fügt man diese in der material-components Datei ein. Somit sind alle eingetragenen Material-Design Komponenten in jeder anderen Komponente verfügbar.



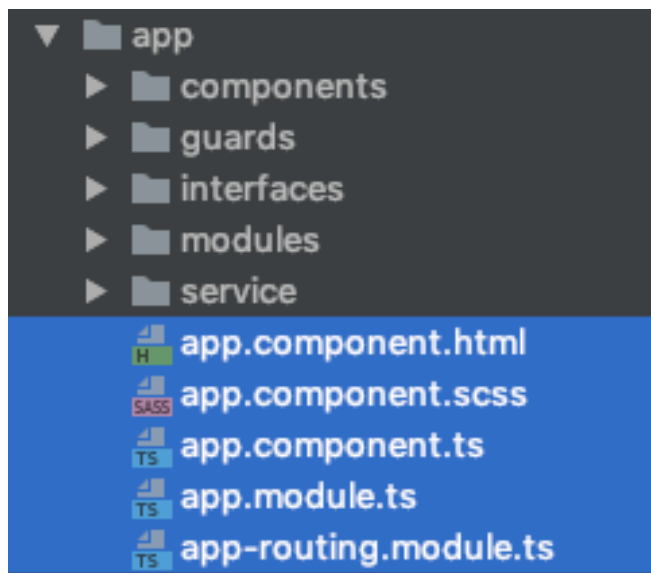
Komponenten sollten selber keine Daten von einer API erhalten. Diese Aufgabe übernehmen sogenannte Services. Services werden meistens dazu benutzt, um Daten aus einer API zu erhalten oder um einen Austausch von Daten zwischen Komponenten zu realisieren. Eine Komponente kann nun diesen Service nutzen, um Daten zu bekommen und muss sich somit nicht selber darum kümmern. Dadurch kann sich eine Komponente vielmehr um die Gestaltung der Daten kümmern.

Im Feedbacksystem werden vier Services benutzt. Der Auth Service dient zur Realisierung des User Login/Log-out. Database Service ist für den Austausch zwischen Datenbank und Web-App zuständig. Es wird ein Titelbar Service benutzt, um den richtigen Titel einer Seite anzuzeigen. Der Letzte Service ist der User Service, er dient zur Bereitstellung von User Informationen in einzelnen Komponenten.





Bei einer neu erstellten Angular App, wird automatisch ein Modul und eine Komponente erstellt. Zusätzlich kann ein Modul zum Routing erzeugt werden. Jede Angular App besteht mindestens aus einem Modul und einer Komponente.



In dem Routing Modul werden die einzelnen Routen erstellt und wenn notwendig mit Guards versehen. Es wird festgelegt, welche Komponente für eine Route zuständig ist. Diese werden dann bei Aufruf der Route angezeigt.

```
const routes: Routes = [
  {path: 'login', component: LoginComponent},
  {
    path: '', component: StartComponent, canActivate: [AuthGuard], children: [
      {path: 'courses/user', component: CoursesComponent},
      {path: 'courses/docent', component: GrantDocentComponent, canActivate: [ModeratorGuard]},
      {path: 'courses/tutor', component: GrantTutorComponent, canActivate: [DocentGuard]},
      {path: 'courses/new', component: NewCourseComponent, canActivate: [ModeratorGuard]},
      {path: 'courses/search', component: SearchCourseComponent},
      {path: 'courses/:id', component: DetailCourseComponent},
      // Admin
      {path: 'admin/dashboard', component: AdminDashboardComponent, canActivate: [AdminGuard]},
      {path: 'admin/user-management', component: AdminUserManagementComponent, canActivate: [AdminGuard]},
      {path: 'admin/checker', component: AdminCheckerComponent, canActivate: [AdminGuard]},
      // Student
      {path: 'student/dashboard', component: StudentDashboardComponent},
      // Prof
      {path: 'docent/dashboard', component: ProfDashboardComponent, canActivate: [IsDocentGuard]}
    ]
  },
];
```

## JWT Autorisierung

Zur Autorisierung wird ein JWT benutzt (<https://jwt.io/introduction/>). Nachdem sich ein User angemeldet hat, wird ihm ein JWT gegeben. Dieser wird bei jedem API Aufruf mitgeschickt und vom Server überprüft, ob dieser valide ist. Das mitschicken des Token muss manuell erfolgen. Dabei wird im Header ein Eintrag mit Authorization: Bearer „Token“ angegeben. In der Angular App wird ein Packet benutzt, welches über einen http interceptor, diesen Token automatisch mitschickt. In einigen Request wird der Token noch manuell eingefügt, da diese Requests nach dem ersten Aufruf einen zweiten API Aufruf tätigen und nicht vom interceptor abgefangen werden. Ein JWT besteht aus drei Teilen, dem Header, dem Payload und die Signatur. Jeder Teil wird verschlüsselt und mit einem Punkt voneinander getrennt.

### Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJjbGllbnRfYXV0aGVudGljYXRpb24iLCJ1c2VyX2lkIjo0MDgsInVzZXJuYXV1IjoicGJ5azEyIiwicHJlbnRtZSI6IiBoaWxpcHAiLCJzdXJuYXV1IjoicQnlrb3ciLCJyb2x1X2lkIjoxNiwicm9sZV9uYXV1Ijoic3R1ZGVudCIsImVtYWlsIjoicGhpbG1wcC5ieWtvd0BtbmkudGhtLmRlIiwidG9rZW5fdHlwZSI6InVzZXIiLCJpYXQiOiJlNTAxNzI1OTQsImV4cCI6MTU1MTM4ODM0NX0.HsTwOym0ItcGn6nuGz42FSVsQhEnmGrBTcJ7cF7gJcA
```

### Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256"
}
```

PAYLOAD: DATA

```
{
  "sub": "client_authentication",
  "user_id": 408,
  "username": "pbyk12",
  "prename": "Philipp",
  "surname": "Bykow",
  "role_id": 16,
  "role_name": "student",
  "email": "philipp.bykow@mni.thm.de",
  "token_type": "user",
  "iat": 1550172594,
  "exp": 1551388345
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Im Header ist angegeben welcher Algorithmus zur Verschlüsselung benutzt wird. Der Payload speichert User Informationen. Die Signatur ist für die Verschlüsselung zuständig. Im Feedbacksystem wird der User Service benutzt, um die Payload Daten aus dem JWT zu bekommen. Dieser Service kann dann in jeder Komponente benutzt werden, die auf User Informationen zugreifen muss. Ein JWT wird entweder nach dem Login über CAS oder LDAP erstellt und dem Benutzer mitgeschickt. Der JWT wird nach dem Erstellen in einem Cookie gespeichert. Die Anwendung liest diesen Cookie aus, speichert den JWT im Lokal Storage und Loggt den User ein.

## Docker

Wir benutzen in unserem Projekt Docker, wodurch die einzelnen Komponenten (Webservice, MySQL Datenbankserver, Kafka, Testsysteme) jeweils in unabhängigen Containern laufen.

Die Virtualisierung in Containerumgebungen erlaubt uns zum einen eine Skalierbarkeit, wodurch höhere Lasten bewältigt werden können. Zudem haben Docker Container den Vorteil, dass sie als Docker Images abgespeichert werden können und somit schnell auf anderen Systemen installierbar sind.

Docker Compose ermöglicht es verschiedene Docker Images in einem virtuellen Netzwerk in mehreren Containern zu starten und zu stoppen um größere, verteilte Systeme zu realisieren.

Viele Programme wie Kafka oder MySQL-Server können für Docker einfach über Dockerhub als Images in der Docker-Compose-YAML eingebunden und heruntergeladen werden. Um die Entwicklung auf Docker noch portabler zu machen wurde auch unsere eigene Software auf Dockerhub hochgeladen.

## Docker Images

Docker Images wurden von den verschiedenen System-Komponenten erstellt um sie in Containern laufen zu lassen. Dies geschah mit Hilfe von sogenannten Dockerfiles, welche aus Anweisungen zur Image-Erstellung und zum Container-Start bestehen.

### Dockerfile – Anweisungen (Auswahl)

*FROM* – Das Basis-Image wird festgelegt, auf dem man das eigene Image baut

*ADD <SRC> <DEST>* - Kopiert aus der Quelladresse in die Zieladresse im Container. (Die Quelle kann dabei auch eine URL sein)

*COPY <SRC> <DEST>* – Kopiert aus der Quelladresse im lokalen Hostsystem in die Zieladresse im Container.

*VOLUME* – Setzt ein Volumen in dem Container für persistenten Speicher

*ENV* – Setzt eine Umgebungsvariable im Container

*EXPOSE* - Öffnet einen Port auf den der Container hören wird

*WORKDIR* – Setzt das interne Verzeichnis aus welchem der Container startet

*ENTRYPOINT* – Die Anweisung mit der der Container gestartet wird

### Benutzte Dockerfiles

Für den Webservice und die einzelnen Testsysteme wurden Dockerfiles erstellt.

#### *Webservice*

```
FROM openjdk:10-jre
MAINTAINER Andrej Sajenko <Andrej.Sajenko@mni.thm.de>

ADD build/install/ws /usr/local/ws
ADD src/main/resources/init.sql /usr/local/ws
VOLUME /usr/local/appconfig/application.properties
VOLUME /upload-dir
VOLUME /zip-dir
EXPOSE 8080
WORKDIR /usr/local/ws/bin
ENV JAVA_OPTS=""
ENTRYPOINT ["/wsd"]
```

### *Secrettoken-Checker*

```
FROM docker:dind
MAINTAINER Vlad Sokyrskyy <vladyslav.sokyrskyy@mni.thm.de>
RUN apk add --no-cache openjdk8
ADD build/install/secrettoken-checker /usr/local/secrettoken-checker
VOLUME /usr/local/secrettoken-checker-upload
VOLUME /usr/local/appconfig/application.config
WORKDIR /usr/local/secrettoken-checker/bin
ENTRYPOINT ["/secrettoken-checker"]
```

### *SQL-Checker*

```
FROM openjdk:10-jre
MAINTAINER Vlad Sokyrskyy <vladyslav.sokyrskyy@mni.thm.de>
ADD build/install/sql-checker /usr/local/sql-checker
VOLUME /upload-dir
VOLUME /usr/local/appconfig/application.config
WORKDIR /usr/local/sql-checker/bin
ENTRYPOINT ["/sql-checker"]
```

### [Docker Compose](#)

Auch Docker Compose wurde für das Projekt benutzt um die verschiedenen Komponenten in ihren Containern in ein gemeinsames virtuelles Netzwerk zu bringen und sie gleichzeitig starten zu können. Dafür wurde eine Docker Compose YAML-Datei angelegt.

Die folgende YAML-Datei ist abhängig von einer lokalen Installation des Feedbacksystems.

## Eine docker-compose.yml Konfiguration im Feedbacksystem

version: '2'

services:

zoo1:

image: 'bitnami/zookeeper:latest'

ports:

- '2181:2181'

environment:

- ALLOW\_ANONYMOUS\_LOGIN=yes

kafka1:

image: 'bitnami/kafka:latest'

ports:

- '9092:9092'

environment:

- KAFKA\_ZOOKEEPER\_CONNECT=zoo1:2181

- ALLOW\_PLAINTEXT\_LISTENER=yes

depends\_on:

- zoo1

mysql1:

image: mysql:8.0

command: --default-authentication-plugin=mysql\_native\_password

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: example

MYSQL\_DATABASE: submissionchecker

ports:

- "3308:3306"

bash1:

image: bash:4.4

mysql2:

image: mysql:8.0

command: --default-authentication-plugin=mysql\_native\_password

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: secretpw

ports:

- "3309:3306"

ws:

build: ws

```
  depends_on:
  - mysql1
  - kafka1
  - zoo1
  ports:
  - "443:8080"
  volumes:
  - ./ws/upload-dir:/upload-dir
  - ./ws/zip-dir:/zip-dir
  - ./docker-config/ws/application.properties:/usr/local/appconfig/application.properties
secrettokenchecker:
  build: secrettoken-checker
  depends_on:
  - mysql1
  - kafka1
  - zoo1
  - ws
  environment:
    HOST_UPLOAD_DIR: ./secrettoken-checker/upload-dir
  volumes:
  - /var/run:/var/run/
  - /var/run/docker.sock:/var/run/docker.sock
  - ./secrettoken-checker/upload-dir:/upload-dir
  - ./docker-
config/secrettoken/application.conf:/usr/local/appconfig/application.config
sqlchecker:
  build: sql-checker
  depends_on:
  - mysql2
  - kafka1
  - zoo1
  - ws
  volumes:
  - ./sql-checker/upload-dir:/upload-dir
  - ./docker-
config/sqlchecker/application.conf:/usr/local/appconfig/application.config
```



## Mehr zu Docker-Compose

Mit dem Kommando `docker-compose up -d --build` startet man die Komposition im Hintergrund. Um im Terminal alle Outputs zu sehen kann man das `-d` weglassen. Wenn die neusten Docker Images des Webservices und der Testsysteme bereits gebaut sind muss man `--build` nicht hinzunehmen.

Mit dem Kommando `docker-compose scale` kann man Container skalieren. Wollte man z.B. den SQL-Checker auf drei Instanzen erweitern kann man das mit dem Befehl `docker-compose scale sqlchecker=3` erreichen.

## Rancher

Rancher (<https://rancher.com/>) kann eingesetzt werden um Container durch eine Web-GUI zu verwalten. Damit lassen sich viele Docker Funktionalitäten einfacher bedienen, wie zum Beispiel das Öffnen einer Shell in einem Container oder das Skalieren von Containern. Es gibt dazu nützliche Anzeigen wie z.B. Speicherverbrauch, Netzwerkadressen, Logs oder Graphen.

## Quellen

[1] Der Softwerker / Das Magazin Vol 10 / Seite 14 bis 22

[2] <https://jaxenter.de/spring-boot-2279>

[3] <https://www.heise.de/developer/artikel/Spring-Boot-Vom-Hype-zur-etablierten-Basistechnologie-4247771.html>

[4] <https://spring.io/projects/spring-boot>

[5] <http://spring.io/projects/spring-framework>