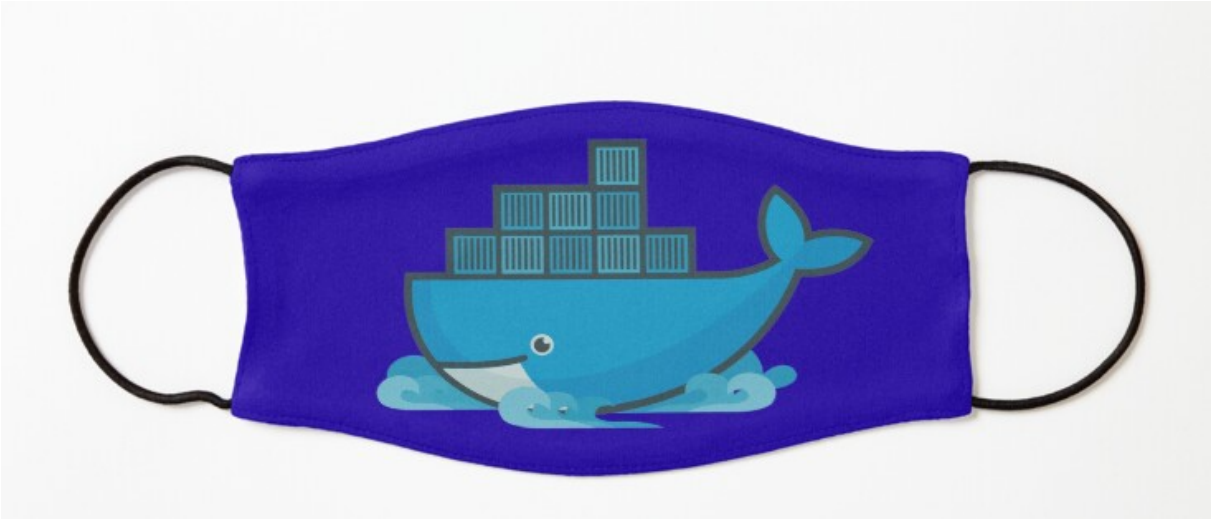
Corona-Rules

Dokumentation der Applikation zur Anzeige und Überprüfung der bundeslandspezifischen Regularien.



by
Hendrik Ohrdes and Robert Rudolph

powered by

Fakultät 73

Stand: 02.09.2020

Inhaltsverzeichnis

1. Einleitung

1.1. Problembeschreibung

In der aktuellen Corona bedingten Situation gibt es neue Regularien, an die sich die Einwohner halten müssen. Diese Regularien ändern sich häufig, sodass es schwierig ist einen Überblick zu behalten. Zudem sind die Regularien für verschiedene Regionen unterschiedlich, sodass die Informationen von verschiedenen Stellen gesammelt werden müssen. Daraus ergibt sich ein erhöhter Aufwand, besonders in der Planung von Veranstaltungen.

1.2. Lösungsvorschlag

Um die Situation zu verbessern wird eine Möglichkeit benötigt, einfach auf die Informationen zuzugreifen. Zusätzlich ist es wünschenswert eine einfache Überprüfung eines Vorhaben bereitzustellen. Zur Lösung dieses Problems wird eine Applikation vorgeschlagen, in der jeder übersichtlich alle Regularien für ein ausgewähltes Bundesland abrufen kann. Zusätzlich wird eine Eingabemaske in der Applikation bereitgestellt, die nach Eingabe einiger Informationen zu einer Veranstaltung ermittelt, ob diese Veranstaltung den Regularien entspricht. Um zu gewährleisten, dass die Daten immer aktuell sind, werden die Informationen regelmäßig von den öffentlichen Schnittstellen abgegriffen und in die Applikation eingepflegt. Außerdem wird noch ein Dashboard bereitgestellt, in dem Autorisierte Nutzer den Import der Daten manuell anstoßen können. Dieser Import kann sowohl für ein einzelnes Bundesland, als auch für alle Bundesländer zusammen angestoßen werden. Weiterhin gibt das Dashboard das Datum des letzten Updates an, sodass die Aktualität der Daten geprüft werden kann.

2. Arbeitsauftrag

Im Rahmen des Projektes sollen mehrere Microservices erstellt werden. Diese sollen Verordnungen im Rahmen der Corona-Pandemie zugänglich zu machen und bei deren Einhaltung unterstützen. in den folgenden Abschnitten werden die Services beschrieben.

2.1. Verordnung-Service

Der Verordnung-Service soll die Verordnungen über eine Rest-API zur Verfügung stellen. Die Verordnungen sollen für jedes Bundesland einzeln abgerufen werden können. Sie enthalten Informationen wie die maximale Anzahl an Personen innerhalb geschlossener Räume oder ob eine Maskenpflicht besteht und welche Gebäudetypen im Rahmen der Pandemie geschlossen sind.

2.2. Import-Service mit Dashboard

Der Import-Service hat die Aufgabe, die Daten aus dem Verordnung-Service in die Datenbank zu überführen. Dies soll über eine per Login geschützte Website vom Nutzer für jedes Bundesland angestoßen werden können. Zudem soll der Service in regelmäßigen Abständen die Regularien für alle Bundesländer vom Verordnung-Service importieren.

2.3. Datenbank

Es soll eine Datenbank erstellt werden, in der die Regularien in einem geeignetem Schema abgelegt werden können.

2.4. Öffentliche Website

Die öffentliche Website soll die Regularien präsentieren. Es soll ein Assistent verfügbar sein, der anhand von vom Nutzer eingegeben Informationen bestimmte, ob das Vorhaben unter den gültigen Regularien stattfinden kann.

3. Projektplanung

3.1. Vorgeschlagene Architektur

In der vorgeschlagenen Architektur gibt es vier Services. Eine öffentliche REST-API in Spring Boot, um die Regelungen zur Verfügung zu stellen. Für den Endnutzer wird ein Web Frontend bereitgestellt, welches durch ein Spring Boot Backend ausgeliefert wird. Das Backend hat direkt Zugriff auf eine Datenbank. Für die Imports der Verordnungen gibt es einen weiteren Service, welcher in Spring Boot umgesetzt wird. Dieser hat zwei Aufgaben: Zum einen wird ein Frontend ausgeliefert, das den manuellen Import ermöglicht. Zum anderen gibt es einen Scheduler, der regelmäßig Daten vom öffentlichen Service abruft. Auch dieser Service hat direkt Zugriff auf die Datenbank. Diese Architektur hat einige Design-Schwächen. Zum einen sind die Aufgaben nicht klar im Sinne des Domain Driven Design getrennt. Der Import-Service zum Beispiel ist zwar auf den ersten Blick nur für den Import zuständig, bei genauer Betrachtung wird aber deutlich, dass er auch für die Datenverarbeitung und Speicherung zuständig. Ähnlich sieht es bei der Öffentlichen Webseite aus. Ein weiterer Nachteil ist, dass beide Backends (Import-Service und Öffentliche Webseite) die gleichen Entitäten vorhalten müssen, da beide mit der selben Datenbank kommunizieren. Dies führt zu erhöhtem Wartungsaufwand, wenn es zu Ändeurngen im Datenmodell kommt. Außerdem muss bei dieser Architektur auf die Synchronisierung geachtet werden, da beide Services gleichzeitig auf die selben Daten zugreifen könnten.

3.2. Optimierte Architektur

Die vorgeschlagene Architektur hat, wie dargestellt, einige Nachteile. Um diese Nachteile zu Umgehen wird eine neue Architektur verschlagen, siehe Abbildung 3.1. Das Problem, dass Entitäten in verschiedenen Containern gehalten und gepflegt werden müssen, wird umgangen, indem die Backends der Services entzerrt werden. Dafür wird vor die Datenbank eine REST-API gesetzt, deren einzige Aufgabe es ist, Daten in die Datenbank zu speichern und Daten daraus auszulesen. Diese REST-API wird sowohl von dem Import-Dashbord, als auch vom Public-Dashboard konsumiert. Durch diesen Schritt werden die Entitäten ausschließlich im Service zur Datenverarbeitung vorgehalten. Das Public- und Import-Dashboard werden dadurch zu reinen Frontend Anwendungen, sodass auch hier die Aufgaben klar abgegrenzt sind. Die öffentliche REST-API bleibt, wie bei der vorgeschlagenen Architektur, ein eigener Service. Zusätzlich zu den genannten Services wird ein Import-Scheduler erstellt. Dieser ist dafür zuständig regelmäßig Daten von der öffentli-

chen API zu sammeln und diese über die REST-API der Datenbank im eigenen System abzuspeichern. Damit besteht auch hier eine klare Trennung der Verantwortlichkeiten nach dem Gedanken des Domain Driven Design.

Diese Architektur hat zudem den Vorteil, dass die Bounded Contexts etwas besser getrennt sind. Der Kontext ist in Abbildung 3.1 durch die verwendeten Netzwerke zu erkennen. Es gibt einen Kontext für den Import, in dem die öffentliche REST-API, das Import Dashboard, der Import Scheduler, sowie die VerordnungserviceAPI enthalten sind. Dieser Kontext hat die Aufgabe Daten von der Öffentlichen API in die eigene VerordnungsserviceAPI zu importieren. Das Netzwerk publicnet spiegelt den Kontext der Interaktion mit dem öffentliche Benutzer wieder. In diesem Kontext ist das Public Dashboard und die VerordnugsserviceAPI enthalten, sodass alle Informationen angezeigt werden können.

3.3. Notwendigkeit der Architektur

Eine weitere mögliche Architektur wäre der direkte Zugriff vom Public Dashboard auf auf die öffentliche REST-API. Durch eine solche Architektur würde alle weiteren Services eingespart werden, sodass Entwicklungs- und Betriebskosten massiv gesenkt werden würden. Es geht aber mit dem Nachteil der Abhängigkeit vom öffentlichen Service einher. Ist dieser nicht erreichbar, ist auch die Funktion der Applikation nicht mehr gegeben. Zudem hängt die Performance des Public-Frontend dann von dem öffentlichen Service ab, sodass auch eine Skalierung nicht möglich wäre. Sollte die öffentliche Schnittstelle ihr Datenmodell, oder ihre Endpunkte ändern, wäre das Public-Dashboard wieder nicht funktionsfähig, bis es an die neue Struktur angepasst wird. Durch die eigenen Services bleibt das Public-Dashboard immer funktionsfähig, es kann skaliert werden, und auf Änderungen der öffentlichen API kann reagiert werden, ohne dass die Applikation ihre Funktion verliert. Schlussendlich könnte die öffentliche API Kosten pro Anfrage verursachen. Diese Kosten werden durch die gewählte Architektur massiv gesenkt.

3.4. Verwendete Technologien

Aufgrund der unterschiedlichen Anforderungen an die Services ergeben sich verschiedene Grundvoraussetzungen und Lösungsansätze für die einzelnen Services. In diesem Abschnitt wird erklärt welche Technologien für welche Services verwendet werden.

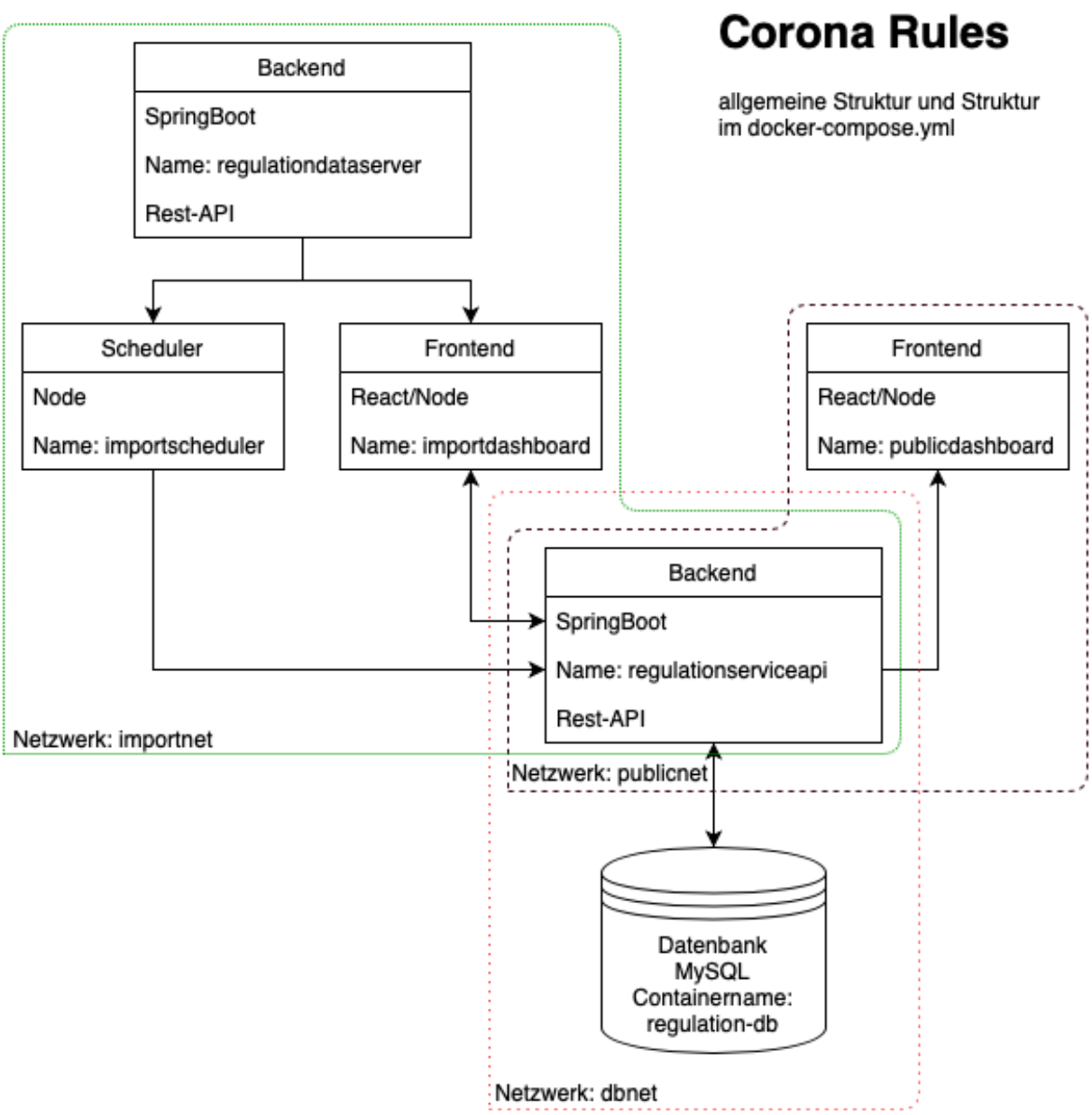


Abbildung 3.1.: Optimierte Architektur

3.4.1. Öffentliche REST-API

Die öffentliche REST-API bietet eine Schnittstelle, um die offiziellen Verordnungen auszulesen. Dieser Service wird als REST-API in Spring Boot umgesetzt. Die Daten werden bei Anfrage zufällig generiert, sodass Änderungen im Datensatz eines echten öffentlichen Services simuliert werden.

3.4.2. VerordnungsserviceAPI

Für die Datenbank REST-API wird wie vorgeschlagen eine Spring Boot Applikation aufgesetzt. Angebunden an die Applikation ist eine MySQL-Datenbank. Die MySQL Datenbank eignet sich für den Anwendungsfall, da die Daten in einem typischen relationalem Datenbankschema vorliegen, sodass sie effizient in einer MySQL Datenbank gespeichert werden können.

3.4.3. Public- und Import-Frontend

Für beide Frontends wird React verwendet. Durch die Verwendung von React stehen alle Möglichkeiten eine moderne Web Applikation zu erstellen zur Verfügung. Außerdem kann die Applikation abschließend kompiliert werden, sodass die direkt durch einen leichtgewichtigen NGINX Server bereitgestellt werden kann.

3.4.4. Import-Scheduler

Der Import-Scheduler hat lediglich die Aufgabe die Daten von der öffentlichen API an die API des eigenen Verordnungsservices weiterzugeben. Das heißt es müssen lediglich zwei Requests regelmäßig durchgeführt werden. Eine schwergewichtige Spring Boot Anwendung ist für diesen Fall ungeeignet, sodass eine einfache Node-Applikation für diesen Zweck verwendet wird. Der Scheduler importiert alle 30 Minuten die Daten von der öffentlichen API und speichert sie durch die Verordnungsservice API in der Datenbank.

3.5. Aufgabenteilung

Es gibt einige Punkte, die gemeinsam im Team diskutiert und ausgearbeitet werden, bevor die einzelnen Services entwickelt werden. Im wesentlichen sind dies die Schnittstellen und das Datenmodell. Es gibt also folgende zwei Schritte, welche als erstes gemeinsam bearbeitet werden, bevor die einzelnen Services bearbeitet werden:

- Erstellung des Datenmodells
- Schnittstellen definieren

Nachdem die Basisvorraussetzungen geschaffen sind, werden die einzelnen Services bearbeitet. Die vorgestellte Architektur bietet klar voneinander abgegrenzte Services. Die zu implementierenden Services sind:

- Öffentlicher Verordnungsservice
- Import Frontend
- Public Frontend
- Verordnungsservice mit Datenbank
- Import Scheduler

Diese Services werden einzeln bearbeitet. Zu der Bearbeitung der einzelnen Services gehört jeweils noch das Erstellen eines Dockerfiles, welches den entsprechenden Service in einen Container verpackt. Abschließend wird in Zusammenarbeit ein docker-compose File erstellt, welches alle notwendigen Images baut und geeignet startet.

Um den Entwicklungsfluss zu maximieren wurden zuerst die Backend Services (Öffentlicher Verordnungsservice, Verordnungsservice mit Datenbank) parallel entwicklet. Anschließend wurden die Frontends (Import Frontend, Public Frontend) entwicklet, wobei während der Entwicklung auf die bereits erstellte Backend Services zugegriffen wurde. Durch diese Reihenfolge müssten die Services während der Entwicklung nicht nachgebildet werden, sodass Zeit in der Entwicklung eingespart werden konnte. Darauf folgend wurde der Import Scheduler entwickelt.

4. Implementierung

4.1. Aufbau der Applikationen

4.1.1. Aufbau der Spring Boot Applikationen

Um den Anforderungen des Single Response Principle gerecht zu werden, wird bei den Spring Boot Applikationen einen Aufteilung in drei Layer vorgenommen. Der erste Layer ist der Controller, welcher die REST-Schnittstelle zur Verfügung stellt und Anfragen an den nächsten Layer weiterleitet. Dieser ist der Service Layer. Hier werden die Anfragen entgegengenommen und gemäß der Programmlogik verarbeitet. Falls persistente Daten benötigt werden, erfolgt von hier aus der Zugriff auf den dritten Layer, den Persistence Layer. Dieser speichert oder lädt die persistenten Daten aus der Datenbank. Da die öffentliche REST-API in diesem Projekt nicht über eine Datenbankanbindung verfügt, wird hier auf den dritten Layer verzichtet.

4.1.2. Aufbau der React Applikationen

Die React Applikationen sind in Ansichten, Komponenten und Service unterteilt. Die Ansichten stellen die Seiten dar, welche für den Benutzer verfügbar sind. Diese sind wiederum aus Komponenten ausgeteilt, die jeweils eine Funktione haben und an verschiedenen Stellen in den Ansichten verwendet werden. Wenn im Kontext der Applikation Informationen vom Backend benötigt werden, wird auf die Services zurückgegriffen, welche die Verbindung zum Backend herstellen und entsprechende Anfragen an dessen REST-Schnittstellen senden. Da das Import-Frontend nur aus einer Ansicht besteht, wird hier auf die Ansichten verzichtet und diese benötigte Ansicht direkt in der App.js abgebildet.

4.2. Aufbau der Dockerfiles

4.2.1. Dockerfiles für Spring Boot Services

Für den Bau der Docker Images auf Spring Boot-Basis wurde eine Multi-Stage Ansatz gewählt, um eine minimale Imagegröße zu erreichen. Im ersten Schritt wurde eine Maven-Image genutzt, um die Applikation aus dem Quellcode zu bauen und das .jar-File entpackt. Im zweiten Schritt wurde ein Alpine Image mit OpenJDK 8 genutzt. In dieses wurden die

für die Anwendung notwendigen Dateien aus dem JAR-File in 3 Layern kopiert um einen minimal Bauzeit beim Neukompilieren der Applikation zu ermöglichen.

4.2.2. Dockerfiles für React/Node Applikation

Wie bei den Spring Boot Containern wurde auch für die React/Node Container ein Multi-Stage Build verwendet. Der erste Stage baut die Anwendung mit npm mit einem Node.js-Image auf Basis von Alpine. Im zweiten Stage wird ein nginx Image, ebenfalls auf Alpine-Basis geladen und die nginx-Konfiguration sowie die statischen Dateien der React Anwendung hinein kopiert um auch hier einen schnell gebauten und verfügbaren Container bereit zustellen.

4.2.3. Dockerfile für den Scheduler

Für den Scheduler wurde eine Node.js Image auf Basis von Alpine verwendet. Es werden die benötigten Pakete installiert und der Quellcode in das Image kopiert.

4.2.4. Einsatz von Dockerize

Trotz des der Nutzung von „depends_on" bei docker-compose kann es vorkommen, dass die Datenbank noch nicht verfügbar ist, wenn das Spring Boot Backend eine Verbindung herstellen möchte und dieser abstürzt. Dieses Problem ist bekannt und es wurden verschiedene Lösungsmöglichkeiten im Internet diskutiert: *https://github.com/spring-projects/spring-boot/issues/4779*. Für den Fall, dass der fehlende Zugriff auf Datenbank nicht in der Applikation selber behandelt wird, schlägt die Dokumentation von Docker selber vor, Scripte zu nutzen, um die richtige Startreihenfolge sicherzustellen: *https://docs.docker.com/compose/startup-order/*. Auf Grund des einfachen Nutzung und guten Dokumentation wurde sich für Dockerize entschieden. Für die Nutzung wird im Dockerfile im finalen Image Dockerize installiert und im Entrypoint der Befehl „-wait" und der Server, welcher vor dem Start des Containers bereitstehen muss angegeben. So können auch mehrere Container angegeben werden. In der Applikation wird Dockerize für die Verordnungsservice-API und den Import-Scheduler genutzt. Die regulationserviceapi wartet auf die Bereitschaft der Datenbank. Der Import-Scheduler wartet auf die Verordnungsservice-API und die Öffentliche Rest-API.

4.3. Aufbau der docker-compose.yml

Das docker-compose File hat die Aufgabe alle zum Projekt gehörenden Images zu bauen und anschließend die Container zu starten. Für den Build Prozess nutzt das File die Dockerfiles der einzelnen Services. Zusätzlich wird noch ein externes Image für die MySQL Datenbank verwendet, siehe A.1 Zeile 10. Dieses Image bekommt durch Umgebungsvariablen den Datenbanknamen, sowie Benutzer und Passwort mitgegeben, sodass es nicht weiter bearbeitet werden muss. Außerdem wird ein Volume zugeordnet, sodass die Daten auch nach einem Neustart persistent sind. Bei den übrigen Services werden neben dem üblichen Build Prozess noch Abhängigkeiten definiert, sodass Services nicht gestarten werden, wenn ein anderer notwendiger Service nicht vorhanden ist. Beispielsweise ist der Verordnungs Service abhängig vom Datenbankcontainer. Weiterhin werden Netwerke definiert in denen die jeweiligen Container kommunizieren können. Der Aufbau der netzwerke ist auch in Abbildung 3.1 dargestellt.

4.4. Absicherung der API

Es ist wichtig, dass nicht jeder schreibenden Zugriff auf die Datenbank hat, um Missbrauch zu verhindern. Daher werden alle schreibenden Endpunkte der VerordnungsserviceAPI gesichert. Die erfolgt, indem im Header ein Benutzer und Passwort mitgesendet wird. Stimmen Benutzer und Passwort mit den hinterlegten Daten überein, werden die gesendeten Daten in die Datenbank geschrieben. Unberechtigte Zugriffe werden werden mit dem Status 401 (unauthorized) zurückgewiesen.

4.5. Komprimierung des Public Dashboard

Da für das Public Dashboard ein hoher Traffic zu erwarten ist, sollen die Daten komprimiert an den Browser ausgeliefert werden. Um dies erreichen wird der nginx Server, der die Webseite ausliefert, durch das nginx.conf File so konfiguriert, dass die Dateien durch gzip komprimiert und dann ausgeliefert werden.

5. Ausblick

5.1. Sicherheitskonzept (TLS)

Die im Rahmen der kurzen Projektlaufzeit erstellten Anwendungen sind nicht vor unautorisierten Zugriffen geschützt und kommunizieren unverschlüsselt. Um die Applikation für den produktiven Betrieb abzusichern, müssten sich die Frontend Applikationen bei den Backend Applikationen authentifizieren und jeder Zugriff autorisiert werden. Um eine sichere Kommunikation zwischen den einzelnen Applikationen zu gewährleisten und Man-in-the-Middle-Angriffen vorzubeugen, sollte diese mit TLS verschlüsselt werden. Hierfür muss für jede Domain ein Zertifikat von einer Zertifizierungsstelle, zum Beispiel Lets Encrypt, erworben werden. Dieses kann dann auf den Servern hinterlegt werden und mit der entsprechenden Konfiguration in Spring Boot bzw. nginx genutzt werden.

5.2. Usability

Die erstellte Applikation erfüllt alle gestellten Anforderungen. Es gibt allerdings weitere mögliche Optimierungen für die Usability. Beispielsweise kann nach der Überprüfung eines Vorhabens genau angezeigt werden, welche Regeln eingehalten wurden und welche nicht. Dadurch würde die User Experience deutlich gesteigert werden und die Planung von Veranstaltung weiter vereinfacht werden.

5.3. Testing

Zu einer guten Software gehören Tests, die das Verhalten der Software automatisiert überprüfen, sodass Fehler bei Änderungen sofort erkannt werden. Dafür sollten in alles Services Unit Tests implementiert werden, die die Funktion der einzelnen Methoden prüfen. Zusätzlich sollten Integration Tests erstellt werden, in denen das Zusammenspiel verschiedener Services, oder verschiedener Layer innerhalb eines Service getestet werden. Auch für die Frontends sollten Tests erstellt werden, die die Benutzerinteraktion nachahmen, sodass das gewünschte Verhalten sichergestellt wird. Diese Tests können z.B. durch cypress auch als End-To-End Tests ausgelegt werden, sodass die gesamte Applikation getestet wird.

6. Fazit

Ausgehend von der Ausgangssituation, in der beschrieben ist, dass die Zugänglichkeit zu aktuellen Corona Regularien nicht unbeschränkt gegeben ist, wurde eine Applikation zur Lösung des Problems erstellt. Der Wunsch Informationen einfach und übersichtlich für verschiedene Bundesländer zu erhalten wurde durch ein Public Dashboard umgesetzt. Dieses Dashboard bietet die Möglich die Regularien für ein gewähltes Bundesland übersichtlich einzusehen. Zusätzlich bietet das Dashboard die Möglichkeit Daten zu einem Vorhaben einzutragen und automatisch prüfen zu lassen, ob die Regularien erfüllt sind. Damit die Daten stets auf aktuellem Stand sind lädt der Import-Scheduler diese regelmäßig von der öffentlichen API und sendet sie an die VerordnungsserviceAPI. Zusätzlich wird ein Import-Dashboard bereitgestellt, indem der Import manuell gestartet werden kann. Durch die erstellte Applikation wird das zugrunde gelegte Problem gelöst. Durch die moderne Microservice Architektur und die Strukturierung des Quellcodes ist die Applikation zudem leicht erweiterbar.

A. Codeauszüge

A.1. docker-compose.yaml

```
version: "3.3"

services:

  regulation-db:
    container_name: regulation-db
    image: mysql:5.7
    ports:
      - 3306:3306
    environment:
      MYSQL_DATABASE: regulations
      MYSQL_USER: regulations
      MYSQL_PASSWORD: regulations
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - regulation-db-data:/var/lib/mysql
    networks:
      - dbnet

  importdashboard:
    container_name: importdashboard
    build:
      context: importdashboard
      dockerfile: Dockerfile
    ports:
      - 1080:82
    depends_on:
      - regulationdataserver
      - regulationserviceapi
    networks:
      - importnet
    stdin_open: true

```

```
  importscheduler:
    container_name: importscheduler
    build:
      context: importscheduler
      dockerfile: Dockerfile
    depends_on:
      - regulationdataserver
      - regulationserviceapi
    networks:
      - importnet

  publicdashboard:
    container_name: publicdashboard
    build:
      context: publicdashboard
      dockerfile: Dockerfile
    ports:
      - 1081:81
    depends_on:
      - regulationserviceapi
    networks:
      - publicnet
    stdin_open: true

  regulationserviceapi:
    container_name: regulationserviceapi
    build:
      context: regulationserviceapi
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    expose:
      - "8080"
    environment:
      SPRING_JPA_PROPERTIES_HIBERNATE_DIALECT:
       ↪ org.hibernate.dialect.MySQL5InnoDBDialect
      SPRING_JPA_HIBERNATE_DDL_AUTO: update
      SPRING_DATASOURCE_URL:
       ↪ jdbc:mysql://regulation-db:3306/regulations
```

```
      SPRING_DATASOURCE_USERNAME: regulations
      SPRING_DATASOURCE_PASSWORD: regulations
    depends_on:
      - regulation-db
    networks:
      - dbnet
      - publicnet
      - importnet

  regulationdataserver:
    container_name: regulationdataserver
    build:
      context: regulationdataserver
      dockerfile: Dockerfile
    ports:
      - "8081:8081"
    expose:
      - "8081"
    environment:
      SERVER_PORT: 8081
    networks:
      - importnet

volumes:
  regulation-db-data:

networks:
  dbnet:
  publicnet:
  importnet:
```