

DSE22Gruppe10-Wartungshandbuch

Diego Krupitza, Jan Müller, Kian Pouresmaeil

26. Mai 2022

Inhaltsverzeichnis

1	Entwicklungsumgebung	1
1.1	Starten der Java Services	1
1.2	Starten der Typescript Services	1
1.3	Konfiguration der Services	1
1.3.1	Konfiguration Entity-Service	1
1.3.2	Konfiguration Tracking-Service	2
1.3.3	Konfiguration Simulator-Service	2
1.3.4	Konfiguration Flowcontrol-Service	2
1.3.5	Konfiguration Gateway-Service	2
2	Frameworks und Bibliotheken	3
2.1	Entity-Service und Tracking-Service	3
2.2	Cockpit	3
2.3	Flowcontrol-Service	4
2.4	Simulator-Service	4
2.5	Gateway-Service	4
3	Build Process	5
4	Test Process	5
4.1	Gateway	5
4.2	Flowcontrol-Service	5
4.3	Simulator-Service	6
4.4	Entity-Service	6
4.5	Tracking-Service	6
5	Deployment	6
5.1	secrets-kubernetes.yaml	7
5.2	ingress-kubernetes.yaml	7
5.3	gateway-kubernetes.yaml	7
5.4	rabbitmq-kubernetes.yaml	7
5.5	cockpit-kubernetes.yaml	8
5.6	entity-kubernetes.yaml	8
5.7	simulator-kubernetes.yaml	8
5.8	flowcontrol-kubernetes.yaml	8
5.9	tracking-kubernetes.yaml	9
6	API Documentation	9

1 Entwicklungsumgebung

Für das Entwickeln der Microservices werden Java 17, Maven 3.8.5, Node.js 16.15.0 sowie Yarn 1.22.18 oder neuer benötigt.

Bevor man die Entwicklungsumgebung verwenden kann, müssen eine Datenbank sowie eine „MoM“ mithilfe von Docker Compose gestartet werden. Dazu kann der Befehl `docker-compose up -d` im Hauptverzeichnis des Projektes ausgeführt werden.

Es gibt keine strikte Reihenfolge für das Starten der Microservices. Die folgende Reihenfolge bietet sich jedoch an, um die Wartezeit zu verringern.

1. Entity-Service
2. Tracking-Service
3. Flowcontrol-Service
4. Simulator-Service
5. Gateway-Service
6. Cockpit

1.1 Starten der Java Services

Die Services *Flowcontrol-Service*, *Simulator-Service* und *Gateway-Service* können mittels Maven gestartet werden. Hierfür muss im Verzeichnis des gewünschten Dienstes der Befehl `mvn spring-boot:run` ausgeführt werden.

1.2 Starten der Typescript Services

Die TypeScript Services sowie das Cockpit lassen sich im jeweiligen Verzeichnis über den Befehl `yarn dev` starten. Zuvor muss der Befehl `yarn install` verwendet werden, um benötigte Bibliotheken zu installieren.

1.3 Konfiguration der Services

Einige Services der Applikation bieten Anpassungsmöglichkeiten über Umgebungsvariablen, wobei vorhandene Standardwerte eine Konfiguration optional machen.

1.3.1 Konfiguration Entity-Service

Folgende Umgebungsvariablen sind anpassbar.

Name	Standardwert	Beschreibung
MONGO_DB_HOST	localhost:27017	Adresse der MongoDB Instanz
MONGO_DB_USER	admin	Username der MongoDB Instanz
MONGO_DB_PWD	admin	Passwort der MongoDB Instanz
MONGO_DB_NAME	local-entity-db	Name der MongoDB Datenbank
RABBIT_MQ_HOST	localhost	Adresse RabbitMQ Instanz

Tabelle 1: Environment variablen für Entity-Service

1.3.2 Konfiguration Tracking-Service

Folgende Umgebungsvariablen sind anpassbar.

Name	Standardwert	Beschreibung
MONGO_DB_HOST	localhost:27017	Adresse der MongoDB Instanz
MONGO_DB_USER	admin	Username der MongoDB Instanz
MONGO_DB_PWD	admin	Passwort der MongoDB Instanz
MONGO_DB_NAME	local-entity-db	Name der MongoDB Datenbank
RABBIT_MQ_HOST	localhost	Adresse RabbitMQ Instanz

Tabelle 2: Environment variablen für Tracking-Service

1.3.3 Konfiguration Simulator-Service

Folgende Umgebungsvariablen sind anpassbar.

Name	Standardwert	Beschreibung
ENTITY_SERVICE_IP	localhost	Adresse des Entity-Service
TRACKING_SERVICE_IP	localhost	Adresse des Tracking-Service
FLOW_SERVICE_IP	localhost	Adresse des Flowcontrol-Service
SPRING_RABBITMQ_HOST	localhost	Adresse der RabbitMQ Instanz

Tabelle 3: Environment variablen für Simulator-Service

1.3.4 Konfiguration Flowcontrol-Service

Folgende Umgebungsvariablen sind anpassbar.

1.3.5 Konfiguration Gateway-Service

Folgende Umgebungsvariablen sind anpassbar.

Name	Standardwert	Beschreibung
ENTITY_SERVICE_IP	localhost	Adresse des Entity-Service
TRACKING_SERVICE_IP	localhost	Adresse des Tracking-Service
SIMULATOR_SERVICE_IP	localhost	Adresse des Simulator-Service
SPRING_RABBITMQ_HOST	localhost	Adresse der RabbitMQ Instanz

Tabelle 4: Environment variablen für Simulator-Service

Name	Standardwert	Beschreibung
ENTITY_SERVICE_IP	localhost	Adresse des Entity-Service
TRACKING_SERVICE_IP	localhost	Adresse des Tracking-Service
SIMULATOR_SERVICE_IP	localhost	Adresse des Simulator-Service
FLOW_SERVICE_IP	localhost	Adresse des Flowcontrol-Service

Tabelle 5: Environment variablen für Gateway-Service

2 Frameworks und Bibliotheken

In diesem Kapitel werden die verwendeten Frameworks und Bibliotheken kurz beschrieben. Detaillierte Informationen können den Dokumentationen der jeweiligen Projekte entnommen werden.

2.1 Entity-Service und Tracking-Service

Sowohl der Entity-Service als auch der Tracking-Service wurden mithilfe des Fastify-Frameworks entwickelt. Dabei handelt es sich um ein Web-Framework für Node.js, welches sich durch seine hohe Performanz sowie Schema-Validierung auszeichnet.

Für die Kommunikation mit der MongoDB Datenbank wird die offizielle Bibliothek node-mongodb-native verwendet.

Die Kommunikation mit der „MoM“ basiert auf der Bibliothek amqpplib.

Darüber hinaus wird dotenv zum Laden der Umgebungsvariablen verwendet.

2.2 Cockpit

Für die Entwicklung des Frontends wurde eine Vielzahl an Bibliotheken eingesetzt. Die Grundlage bildet das Framework Vue 3. In Kombination mit dem Build-Tool Vite, bietet Vue 3 eine moderne Entwicklungsumgebung und beschleunigt die Umsetzung der Anforderungen durch nützliche Funktionen wie beispielsweise Hot-Reloads in wenigen Millisekunden.

Darüber hinaus wurde FormKit eingesetzt um eine Validierung von Nutzereingaben zu vereinfachen.

Axios wurde als zuverlässiger HTTP-Client in das Frontend eingebunden, um eine Kommunikation mit dem Backend umzusetzen.

Die Bibliothek vue-toastification kommt beim Anzeigen von Statusmeldungen, beispielsweise beim Starten und Stoppen von Simulationen, zum Einsatz.

Als performantere Alternative zu Tailwind wurde die Bibliothek UnoCSS verwendet.

2.3 Flowcontrol-Service

Für die Entwicklung des Flowcontrol-Services, welcher in Java implementiert wurde, wurden Spring Boot sowie Spring Cloud eingesetzt. Damit die REST-Kommunikation mit den anderen Services nur minimalen Boilerplate-Code aufweist, wurde mit

`org.springframework.cloud : spring-cloud-starter-feign` eine Implementierung von OpenFeign verwendet. Da dieser Service auch mit RabbitMQ kommuniziert, wurde zudem `spring-boot-starter-amqp` eingebunden. Dies ermöglichte uns eine simple Anbindung von mit RabbitMQ Instanzen für eine asynchrone Kommunikation. Zum Testen wurden JUnit5 sowie TestContainer verwendet.

2.4 Simulator-Service

Für die Entwicklung des Simulator-Services, welcher in Java implementiert wurde, wurden Spring Boot, Spring Cloud sowie Spring MVC eingesetzt. Damit die REST-Kommunikation mit den anderen Services nur minimalen Boilerplate-Code aufweist, wurde mit

`org.springframework.cloud : spring-cloud-starter-feign` eine Implementierung von OpenFeign verwendet. Da dieser Service auch mit RabbitMQ kommuniziert, wurde zudem `spring-boot-starter-amqp` eingebunden. Dies ermöglichte uns eine simple Anbindung von mit RabbitMQ Instanzen für eine asynchrone Kommunikation. Zum Testen wurden JUnit5 sowie TestContainer verwendet.

2.5 Gateway-Service

Für die Entwicklung des Gateway-Services, welcher in Java implementiert wurde, wurden Spring Boot sowie Spring Cloud eingesetzt. Insbesondere die Bibliothek `org.springframework.cloud : spring-cloud-starter-gateway`

war für die Umsetzung vom Gateway von Bedeutung, da diese das Erstellen eines Gateways auf Basis einer Konfigurationsdatei ermöglicht.

3 Build Process

Die Java-Dienste lassen über den Befehl `mvn package -DskipTests` bauen. Analog dazu baut der Befehl `yarn build` die TypeScript-Dienste.

Letztendlich findet dies nur beim Bauen der Docker-Images statt, für welche pro Verzeichnis ein Dockerfile angelegt wurde. Für eine kürzere Bauzeit wurden die Dockerfiles so programmiert, dass diese mehrere Ebenen verwenden und Dependencies zwischengespeichert. Um das Bauen der Docker-Images weiter zu vereinfachen, können die `dockerize`-Skripte der Datei `package.json` verwendet werden. Der lokale Docker-User muss dafür Zugriffsrechte auf die Docker Hub Repositories besitzen.

4 Test Process

Für jeden Microservice wurden mindestens zwei Modultests implementiert.

Die Java-basierten Services wurden mithilfe von *JUnit5* getestet. Hierbei wurde die Springboot Version 2.6.5 verwendet.

Die Tests der TypeScript-basierten Services nutzen den *Test Runner Vite* mit Version 0.12.6.

4.1 Gateway

Beim Gateway wird getestet, ob alle Services und deren Status vom Gateway richtig angezeigt werden. Die *Health-Check* Anfragen an die jeweiligen Services werden mithilfe von *Mockito* gemockt.

Die Tests können mit dem Befehl `mvn clean test` in der Kommandozeile ausgeführt werden.

4.2 Flowcontrol-Service

Die Tests des Flowcontrol-Service überprüfen das Berechnen von optimalen Geschwindigkeit für Fahrzeuges in der Nähe von Ampeln. Das Ziel der Berechnung ist, dass Fahrzeuge so selten wie möglich zu einem totalen Stillstand kommen.

Der Flowcontrol-Service kommuniziert in der Produktionsversion mit einer RabbitMQ Instanz. Damit die Tests unabhängig von RabbitMQ laufen können, kommt hier ein Testcontainer mit dem Image `rabbitmq:3-management`

zum Einsatz. Zusätzlich wird die Kommunikation mit anderen Microservices mithilfe von *Mockito* gemockt.

Die Tests können mit dem Befehl *mvn clean test* in der Kommandozeile ausgeführt werden.

4.3 Simulator-Service

Das korrekte Verhalten beim Starten und Stoppen einer Simulation wird mit den vorhandenen Integrationstests überprüft. Dabei werden sowohl eingehende Anfrage an den Service als auch ausgehende Anfragen an weiteren Services mithilfe von *Mockito* gemockt.

Die Tests können mit dem Befehl *mvn clean test* in der Kommandozeile ausgeführt werden.

4.4 Entity-Service

Die vorhandenen Integrationstests überprüfen, ob die Endpunkte den Anforderungen entsprechend verhalten und der Service seinen Status über seinen *health*-Endpunkt korrekt übermittelt. Die Anfragen werden mithilfe eines im Test erstellten Testservers verarbeitet. Zusätzlich wird eine *In-Memory-MongoDB* Instanz verwendet.

Die Tests können mit dem Befehl *yarn test* in der Kommandozeile ausgeführt werden.

4.5 Tracking-Service

Die vorhandenen Integrationstests überprüfen, ob die Endpunkte den Anforderungen entsprechend verhalten und der Service seinen Status über seinen *health*-Endpunkt korrekt übermittelt. Die Anfragen werden mithilfe eines im Test erstellten Testservers verarbeitet. Zusätzlich wird eine *In-Memory-MongoDB* Instanz verwendet.

Die Tests können mit dem Befehl *yarn test* in der Kommandozeile ausgeführt werden.

5 Deployment

Für das Deployment wurden GKE sowie die dem Projekt beigelegten YAML-Dateien verwendet. Die YAML-Dateien befinden sich im Verzeichnis *kubernetes*. Auf dem Kubernetes Cluster können diese Dateien mit dem Befehl `kubectl apply -f .` angewendet werden. In den folgenden Unterpunk-

ten finden sich detaillierte Beschreibungen der neun YAML-Konfigurationsdateien.

5.1 secrets-kubernetes.yaml

Für das Deployment der Applikation auf Kubernetes müssen bestimmte sensible Information vorerst als *Secrets* in das Cluster eingespielt werden. Diese Secrets werden in einer Datei mit dem Namen *secrets-kubernetes.yaml* abgelegt. Da diese Secrets je nach Deployment unterscheiden können, gibt es die Vorlage *example-secrets-kubernetes.yaml*.

Das Mapping der Secrets muss den Namen *dse-secrets-v2i* haben.

Im Deployment befinden sich folgende Secrets.

Name	Beschreibung
MONGO_DB_HOST	Der Hostname der MongoDB Instanz
MONGO_DB_USER	Der Username der MongoDB Instanz
MONGO_DB_PWD	Das Passwort der MongoDB Instanz

Tabelle 6: Secrets des Deployments

5.2 ingress-kubernetes.yaml

Für die Applikation wurde ein Ingress aufgesetzt, welches Anfragen basierend auf einem Pfadpräfix weiterleitet. Anfragen, deren Pfade mit */api/* beginnen, werden an den Gateway-Service weitergeleitet. Jegliche andere Anfragen werden per Standardeinstellung an den Cockpit-Service weitergeleitet.

5.3 gateway-kubernetes.yaml

In dieser YAML-Datei liegt die Konfiguration des Gateways-Deployments vor. Es wird ein Deployment mit dem Namen *gateway* erstellt, welches das Docker Image *deryeger/dse-gateway* verwendet. Zusätzlich werden alle Konfigurationsvariablen, die in dem Kapitel *Entwicklungsumgebung* beschrieben werden, gesetzt. Letztlich wird ein *ClusterIP-Service* mit dem Namen *gateway-service* erstellt, um den Port des Containers zugänglich zu machen.

5.4 rabbitmq-kubernetes.yaml

In dieser YAML-Datei liegt die Konfiguration einer RabbitMQ-Instanz vor. Es wird ein Deployment mit dem Namen *rabbitmq* erstellt, welches das Docker

Image *rabbitmq:management-alpine* verwendet. Letztlich wird ein ClusterIP-Service mit dem Namen *rabbitmq-service* erstellt, um den Port des Containers zugänglich zu machen.

5.5 cockpit-kubernetes.yaml

In dieser YAML-Datei liegt die Konfiguration des Cockpit-Deployments vor. Es wird ein Deployment mit dem Namen *cockpit* erstellt, welches das Docker Image *deryeger/dse-cockpit* verwendet. Letztlich wird ein ClusterIP-Service mit dem Namen *cockpit-service* erstellt, um den Port des Containers zugänglich zu machen.

5.6 entity-kubernetes.yaml

In dieser YAML-Datei liegt die Konfiguration des Entity-Service-Deployments vor. Es wird ein Deployment mit dem Namen *entity* erstellt, welches das Docker Image *deryeger/dse-entity* verwendet. Zusätzlich werden alle Konfigurationsvariablen, die in dem Kapitel *Entwicklungsumgebung* beschrieben werden, gesetzt. Beim Setzen der Konfigurationsvariablen werden unter anderem Werte aus dem Secret *dse-secrets-v2i* genutzt. Letztlich wird ein ClusterIP-Service mit dem Namen *entity-service* erstellt, um den Port des Containers zugänglich zu machen.

5.7 simulator-kubernetes.yaml

In dieser YAML-Datei liegt die Konfiguration des Simulator-Service-Deployments vor. Es wird ein Deployment mit dem Namen *simulator* erstellt, welches das Docker Image *deryeger/dse-simulator* verwendet. Zusätzlich werden alle Konfigurationsvariablen, die in dem Kapitel *Entwicklungsumgebung* beschrieben werden, gesetzt. Letztlich wird ein ClusterIP-Service mit dem Namen *simulator-service* erstellt, um den Port des Containers zugänglich zu machen.

5.8 flowcontrol-kubernetes.yaml

In dieser YAML-Datei liegt die Konfiguration des Flowcontrol-Service-Deployments vor. Es wird ein Deployment mit dem Namen *flowcontrol* erstellt, welches das Docker Image *deryeger/dse-flowcontrol* verwendet. Zusätzlich werden alle Konfigurationsvariablen, die in dem Kapitel *Entwicklungsumgebung* beschrieben werden, gesetzt. Letztlich wird ein ClusterIP-Service mit dem Namen *flowcontrol-service* erstellt, welches den passenden Port des Containers zugänglich macht.

5.9 tracking-kubernetes.yaml

In dieser YAML-Datei liegt die Konfiguration des Tracking-Service-Deployments vor. Es wird ein Deployment mit dem Namen *tracking* erstellt, welches das Docker Image *deryeger/dse-tracking* verwendet. Zusätzlich werden alle Konfigurationsvariablen, die in dem Kapitel *Entwicklungsumgebung* beschrieben werden, gesetzt. Beim Setzen der Konfigurationsvariablen werden unter anderem Werte aus dem Secret *dse-secrets-v2i* genutzt. Letztlich wird ein ClusterIP-Service mit dem Namen *tracking-service* erstellt, um den Port des Containers zugänglich zu machen.

6 API Documentation

Eine interaktive Swagger-Dokumentation findet man nach Starten der Anwendung auf http://<gateway_ip>/swagger-ui.html. Zusätzlich befinden sich im *docs* Verzeichnis des Projektes die Dokumentationen aller Services in Form von PDF- und YAML-Dateien. Für eine visualisierte Dokumentation ohne Inbetriebnahme des Gateways, lassen sich besagte YAML-Dateien auf der Webseite <https://editor.swagger.io/> importieren.