2 – Architektur- und Designdokument

Diego Krupitza, Jan Müller, Kian Pouresmaeil

15. Juni 2022

Inhaltsverzeichnis

1 Software Architektur

1.1 Services & Komponenten

Die Anwendung umfasst vier Microservices sowie ein Gateway. In den folgenden Unterkapiteln werden Microservice und Gateway vorgestellt.

1.1.1 Gateway

Die Aufgabe des Gateways ist es eingehende Anfragen basierend auf Routing-Regeln weiterzuleiten. Darüber hinaus befindet sich im Gateway auch die zentrale Swagger-UI-Dokumentationsplattform, auf welcher Endbenutzerinnen und Endbenutzer die REST-Schnittstellen der Microservices einsehen können.

1.1.2 Entity-Service

Die Aufgabe des Entity-Services ist es Metadaten sowie statische Daten von Entitäten zu verwalten. Akteure registrieren sich bei diesem Service, um Teil der Simulation zu werden. Über REST-Schnittellen können diese Daten abgefragt werden.

1.1.3 Tracking-Service

Die Aufgabe des Tracking-Services ist die Verwaltung von dynamischen Daten, welche von Entitäten generiert werden. Während einer Simulation werden diese Daten an den Tracking-Service gesendet und von diesem im Anschluss persistiert. Über REST-Schnittellen können diese Daten abgefragt werden.

1.1.4 Simulator-Service

Die Zentrale Aufgabe des Simulator-Services ist das Starten und Stoppen von Simulationen. Sobald eine Simulation gestartet wurde, simuliert dieser Service Ampeln und Fahrzeuge. Es registriert diese Akteure und spielt das Verhalten, welches im Algorithmus spezifiziert wurde, ab.

1.1.5 Flowcontrol-Service

Der Flowcontrol-Service berechnet optimalen Geschwindigkeit für Fahrzeuge basierend auf dem Zustand von Ampeln, sodass eine *grüne Welle* erreicht wird.

1.2 Schnittstellen & Kommunikation der Services

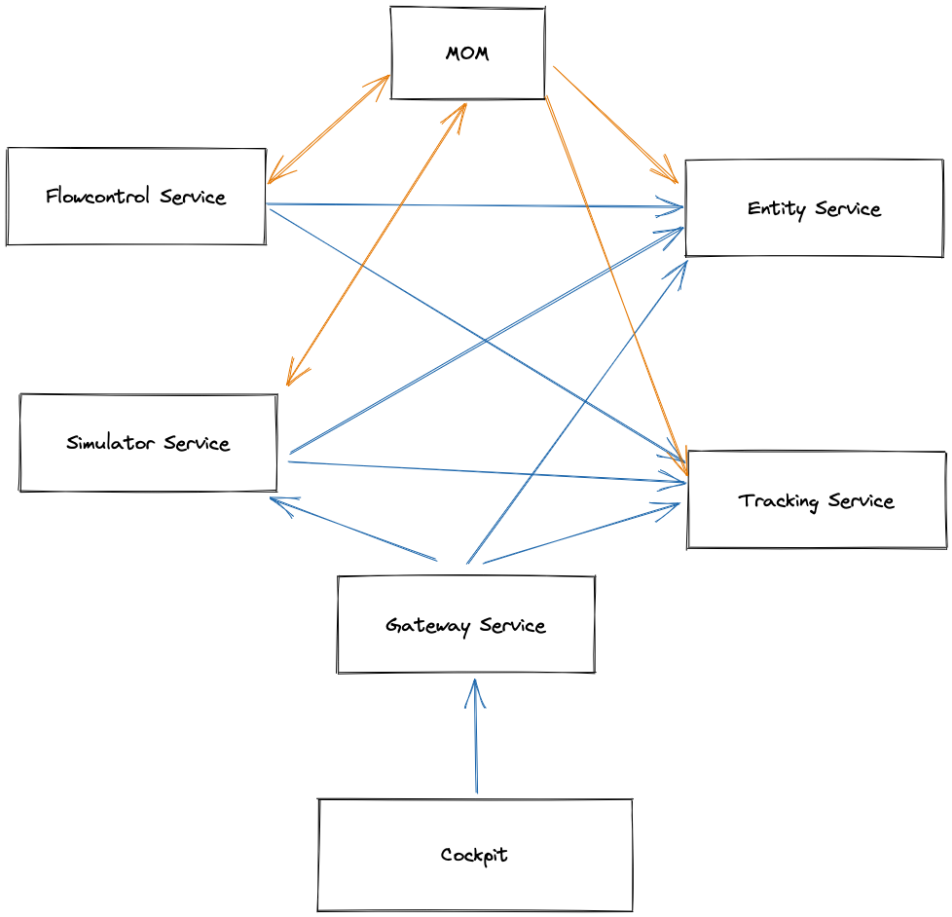

Abbildung 1: Übersicht aller Schnittstellen und Kommunikationswege

Wie man der Abbildung 1 entnehmen kann, verfügt das System über zwei Arten von Schnittstellen. Zuerst gibt es konventionelle synchrone REST-Schnittstellen (in Abbildung 1 mittels blauen Pfeilen dargestellt). Darüber hinaus existieren asynchrone Schnittstellen (in Abbildung 1 mittels orangen Pfeilen dargestellt), welche mit einer Message Oriented Middleware (in diesem Falle RabbitMQ) realisiert wurden. Grundsätzlich verwenden wir asynchrone Kommunikation, damit häufig stattfindender Datenaustausch nicht die Services blockiert. Synchrone Kommunikation wird bei Operationen eingesetzt, welche nicht zeitkritisch sind oder synchron stattfinden müssen.

1.2.1 REST

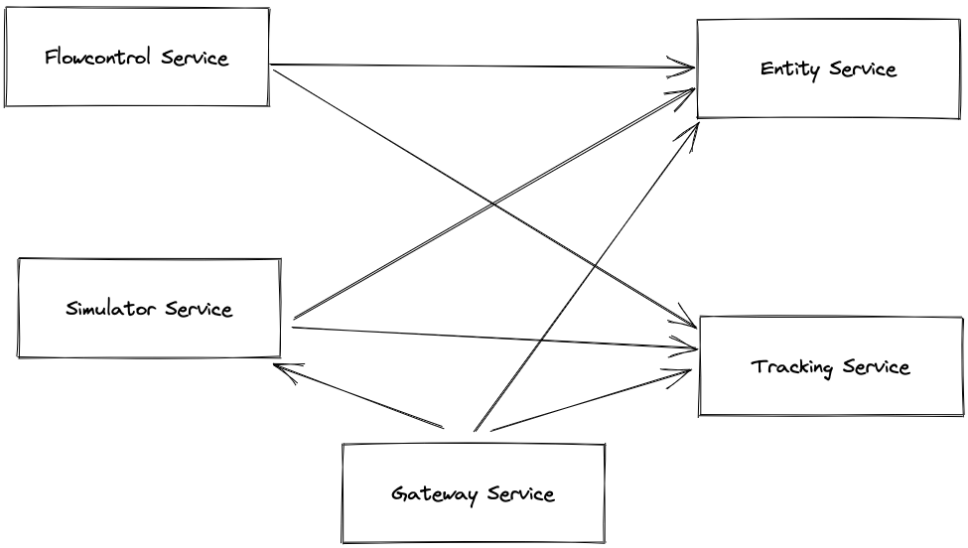

Abbildung 2: Überblick der REST-Kommunikation

Die Abbildung 2 stellt die REST-Kommunikation der Services dar.

Gateway Der Gateway kommuniziert per REST mit allen Services, welche eine solche Schnittstelle besitzen und diese auch nach außen zugänglich machen.

Flowcontrol-Service Der Flowcontrol-Service stellt keine REST-Schnittstelle bereit. Dieser Service kommuniziert mit dem *Entity-Service* sowie dem *Tracking-Service*, um alle Eingaben für den Algorithmus zur Berechnung der optimalen Geschwindigkeit zu erhalten.

Simulator-Service Der Simulator-Service stellt drei REST-Endpunkte zur Verfügung. Diese ermöglichen das Starten und Stoppen von Simulationen. Der Simulator-Service kommuniziert mit dem *Entity-Service* sowie dem *Tracking-Service*, um einen Beenden von aktiven Simulationen zu erwirken.

Entity-Service Der Entity-Service stellt REST-Endpunkte zur Verfügung, mit welchen Metadaten und statische Informationen zu den Akteuren einer

aktiven Simulation abgefragt werden können. Der Dienst selbst sendet keine Anfragen an andere Services des Systems.

Tracking-Service Der Tracking-Service stellt REST-Endpunkte zur Verfügung, mit welchen Daten zum aktuellen Zustand der Akteure einer aktiven Simulation abgefragt werden können. Der Dienst selbst sendet keine Anfragen an andere Services des Systems.

1.2.2 Message Oriented Middleware

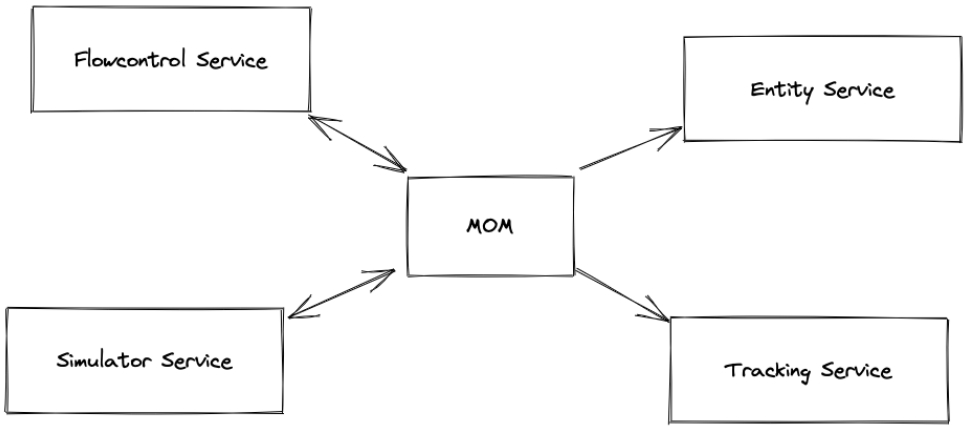

Abbildung 3: Überblick der „MoM“- Kommunikation

Die Abbildung 3 bietet einen Überblick über die „MoM“-Kommunikation der Services. Wie dort zu sehen ist, empfangen *Tracking-Service* und *Entity-Service* nur Daten von der „MoM“ und senden selbst keine Nachrichten. Auf der anderen Seite ist zu sehen, dass *Flowcontrol-Service* und *Simulator-Service* sowohl Daten an die „MoM“ senden als auch empfangen.

Insgesamt gibt es sechs Queues und einer Fanout-Exchange mit dem Namen `fanout.carState`. Die Namen der Queues sind wie folgt:

1. `car-state-tracking`
2. `car-state-flow`
3. `car`
4. `traffic-light`

5. `traffic-light-state`

6. `car-speed`

In den folgenden Abschnitten werden die Queues sowie die Fanout-Exchange vorgestellt.

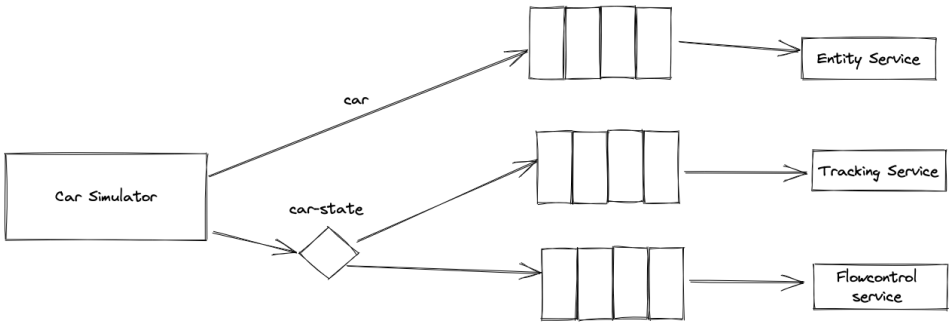

Abbildung 4: Verwendung der *Car*, *Car-State-** und Exchange

car, car-state-*, exchange In Abbildung 4 ist die Verwendung der Queues *Car*, *Car-State-** und der Exchange dargestellt. Zu Beginn einer Simulation sendet der *Car-Simulator* Registrierungsinformationen von Fahrzeugen an die Queue *car*. Diese Queue wird vom Entity-Service konsumiert. Während der Simulation werden aktuelle Positionen, Geschwindigkeiten und weiteres dynamische Daten an die Fanout-Exchange gesendet (welche in der Abbildung 4 als Raute gekennzeichnet ist). Die Fanout-Exchange leitet diese an die beiden Queues *car-state-flow* und *car-state-tracking* weiter. Diese Designentscheidung wurde getroffen, da es sich hier um ein *PubSub*-Szenario und keine *Competing-Consumers* handelt.

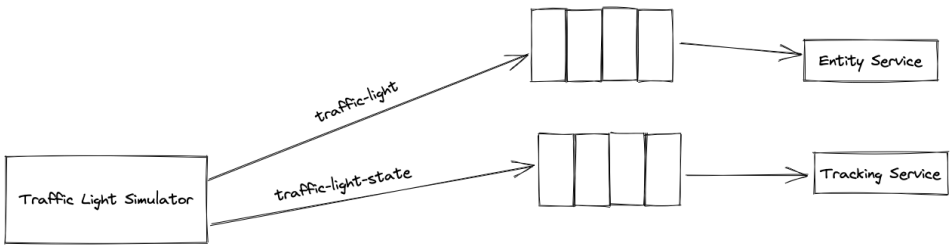

Abbildung 5: Verwendung der Queues *traffic-light*, *traffic-light-state*

taffic-light, taffic-light-state-* In Abbildung 5 wird die Verwendung der Queues *traffic-light*, *traffic-light-state* dargestellt. Zu Beginn einer Simulation sendet der *TrafficLight-Simulator* Registrierungsinformationen von Ampeln an die Queue *traffic-light*. Diese Queue wird vom Entity-Service konsumiert. Während der Simulation werden die verbleibende Zeit einer aktiven Ampelphase sowie weitere dynamische Daten an die Queue *traffic-light-state* gesendet, welche vom *tracking-service* konsumiert wird.

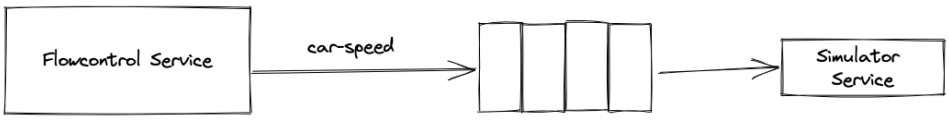

Abbildung 6: Verwendung der Queue *car-speed*

car-speed Zuletzt zeigt Abbildung 5 die Verwendung der Queue *car-speed*. Der *Flowcontrol-Service* sendet, nachdem eine optimale Geschwindigkeit für ein Fahrzeug berechnet wurde, diese mittels der Queue *car-speed* and den *Simulator-Service*.

2 Logical View

Die logische Sicht beschreibt die Funktionalität der Komponenten. Dazu wird ein UML-Klassendiagramm für jeden Microservice bereitgestellt. Um die Lesbarkeit zu verbessern, werden die Bibliotheksklassen von *Spring* und *Fastify* nicht in den Diagrammen dargestellt. Zusätzlich sind der Entity- sowie der Tracking-Service nicht in einem objektorientierten Stil mit TypeScript entwickelt worden. Da sie keine Klassen verwenden, basieren ihre Klassendiagramme nicht auf Klassen, sondern auf deren Modulen. Dadurch können die Funktionalitäten ähnlich zu einem klassischen Klassendiagramm abgebildet werden.

2.1 Entity-Service

Siehe Abbildung 7.

2.2 Tracking-Service

Siehe Abbildung 8.

2.3 Gateway-Service

Siehe Abbildung 9.

2.4 Simulator-Service

Siehe Abbildung 10.

2.5 Flowcontrol-Service

Siehe Abbildung 11.

3 Physical/Deployment View

Die *Physical View* beschreibt die Anordnung der Komponenten. Sie stellt die Topologie und Kommunikation zwischen den Komponenten dar. Da das Projekt über Kubernetes bereitgestellt wird, zeigt diese physische Ansicht das Projekt in einem Kubernetes-Cluster an. Die Physical View ist in Abbildung 12 zu sehen.

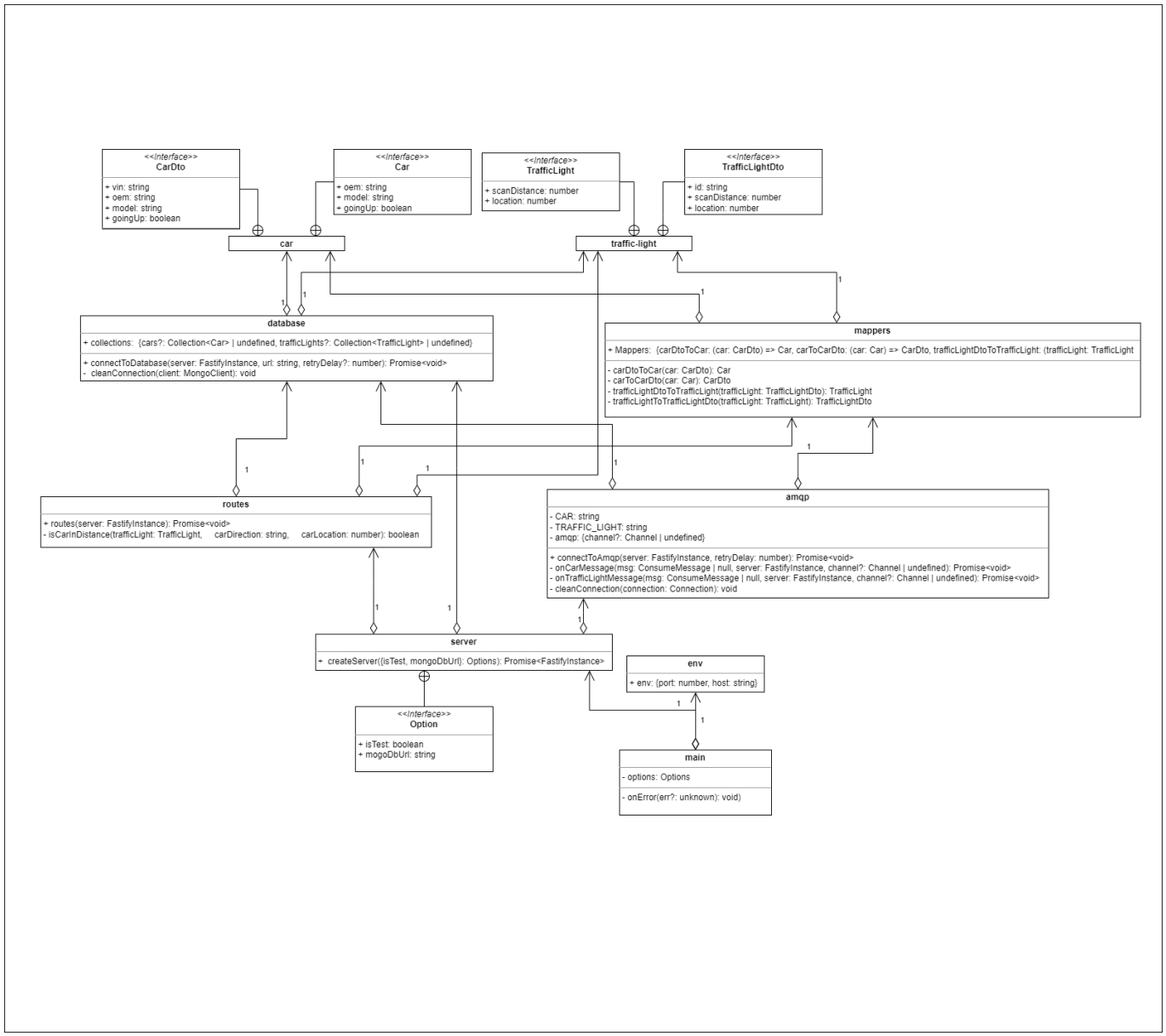

Abbildung 7: Entity-Service UML

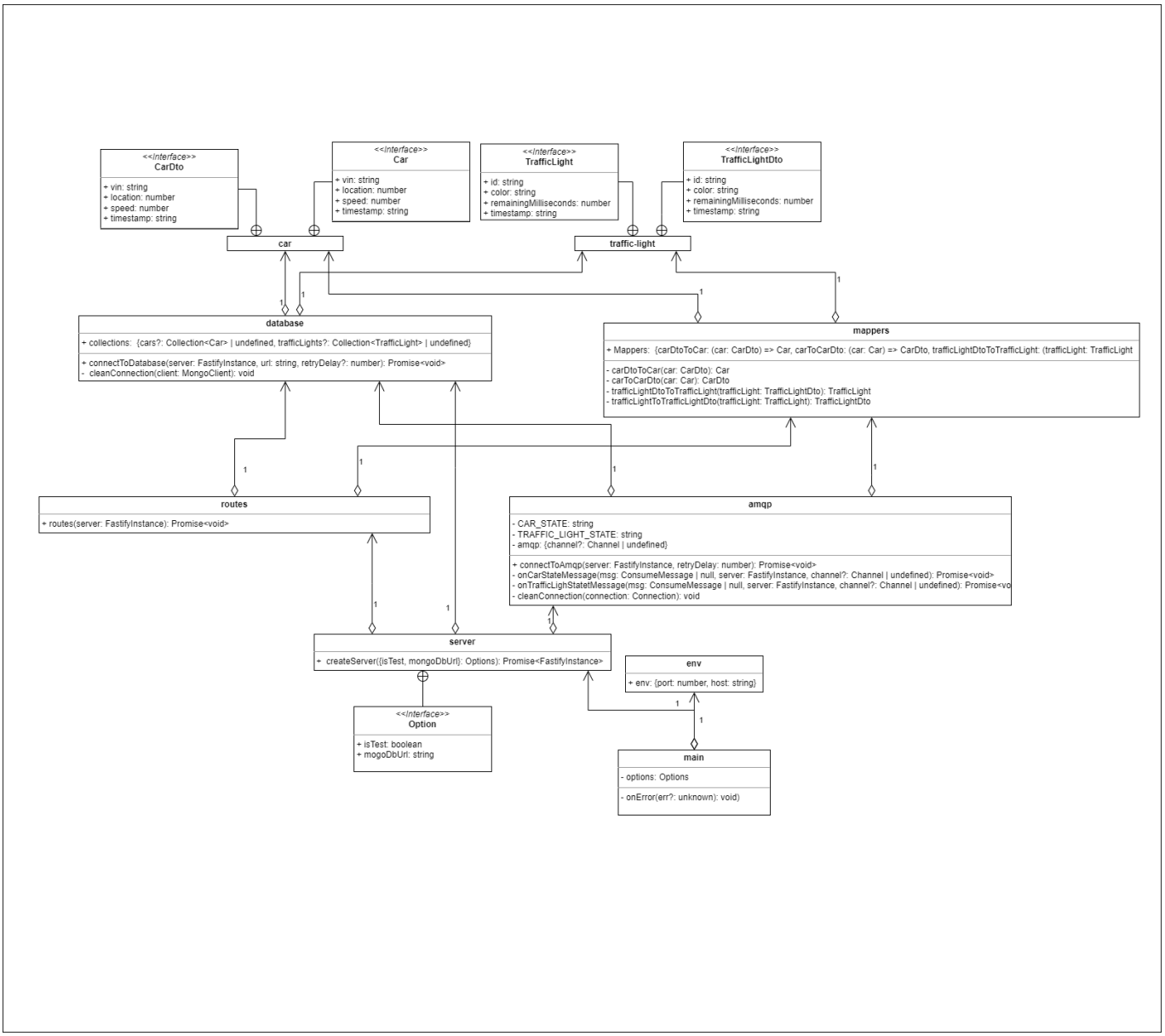

Abbildung 8: Tracking-Service UML

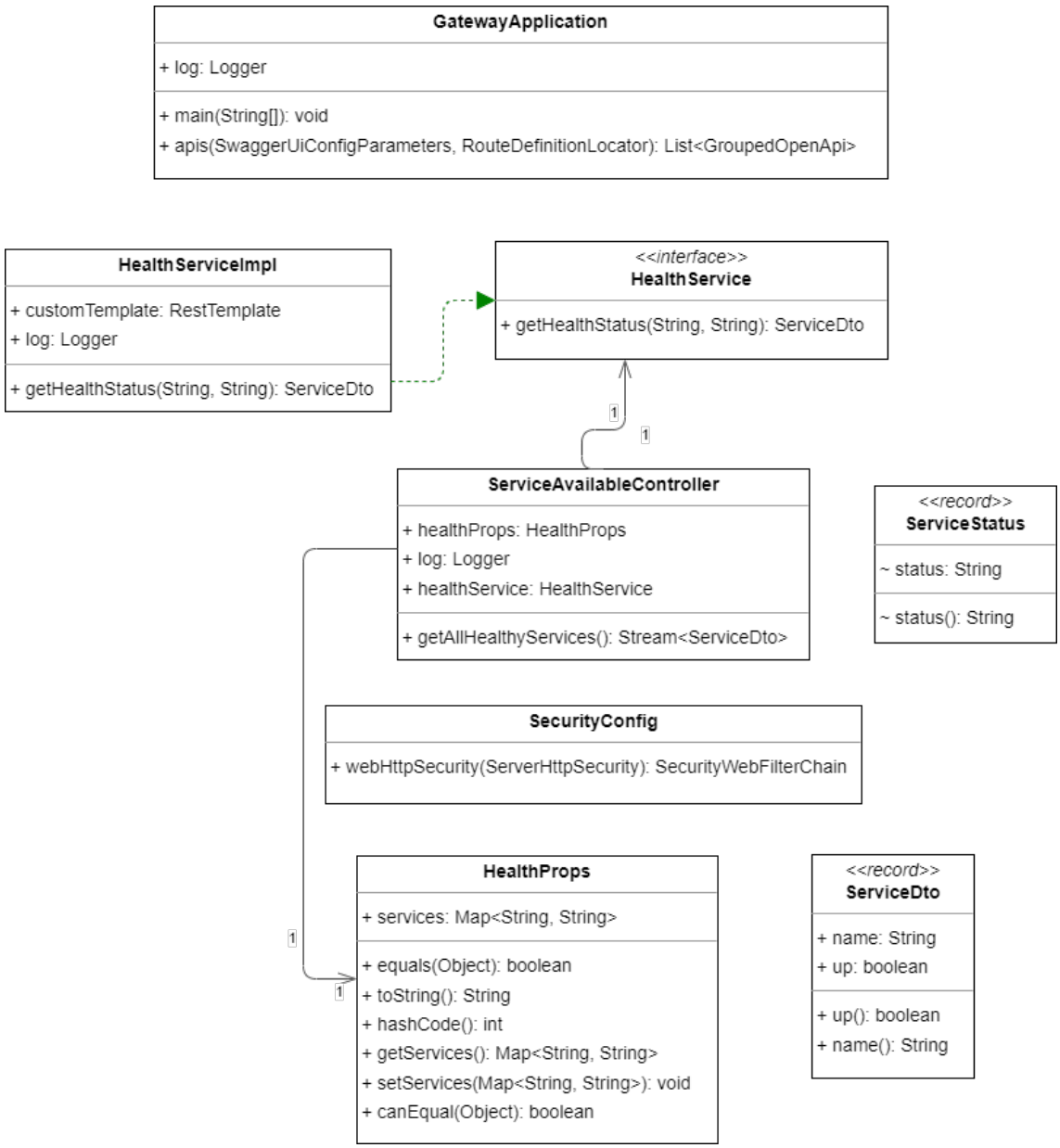

Abbildung 9: Gateway-Service UML

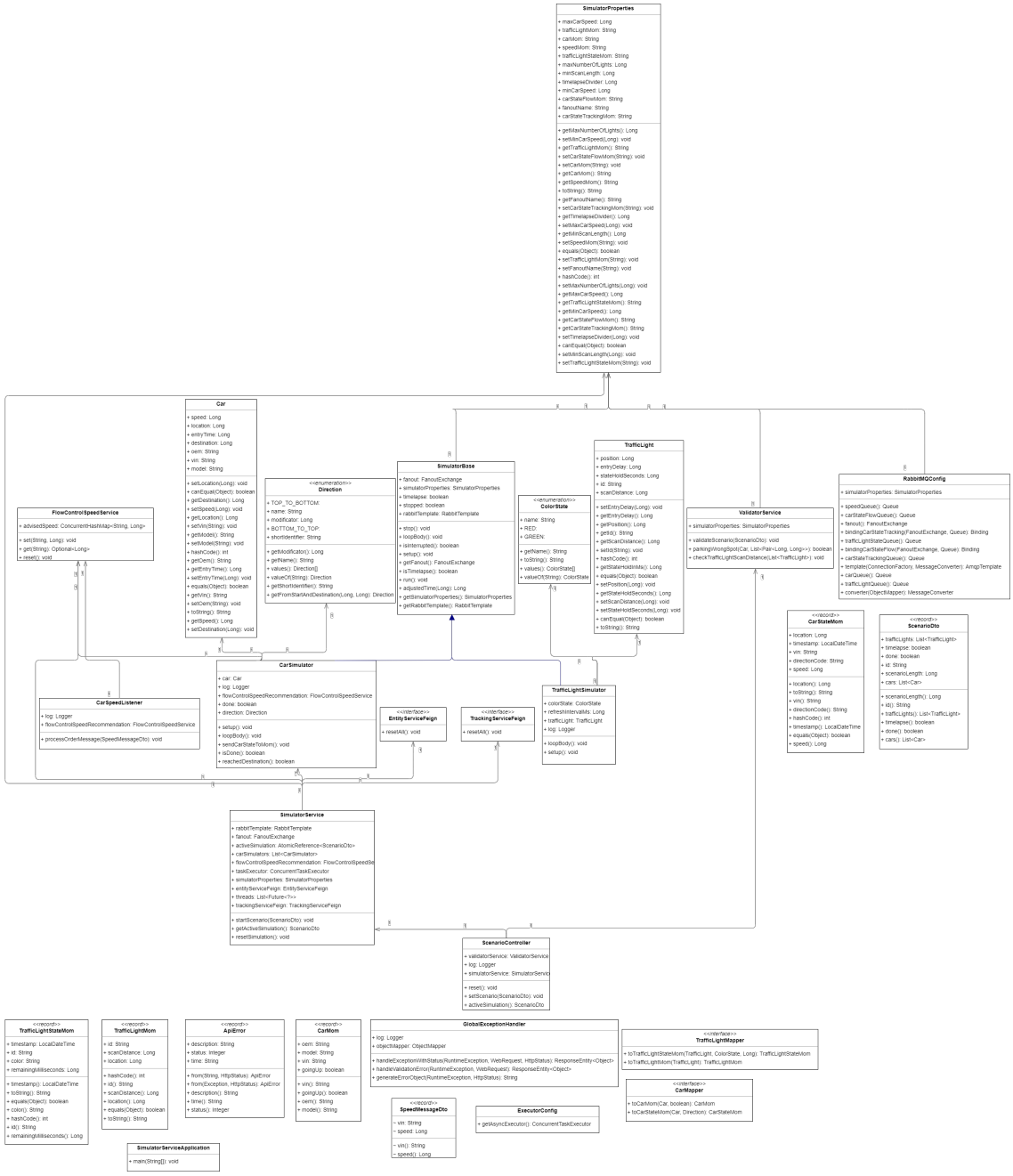

Abbildung 10: Simulator-Service UML

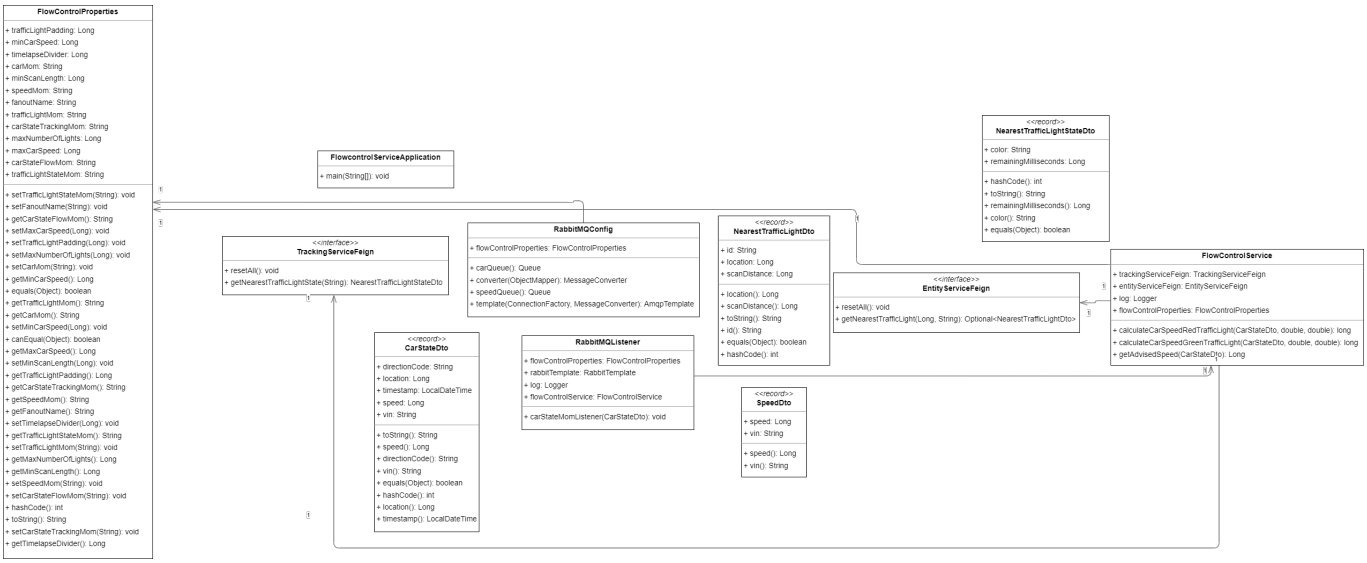

Abbildung 11: Flowcontrol-Service UML

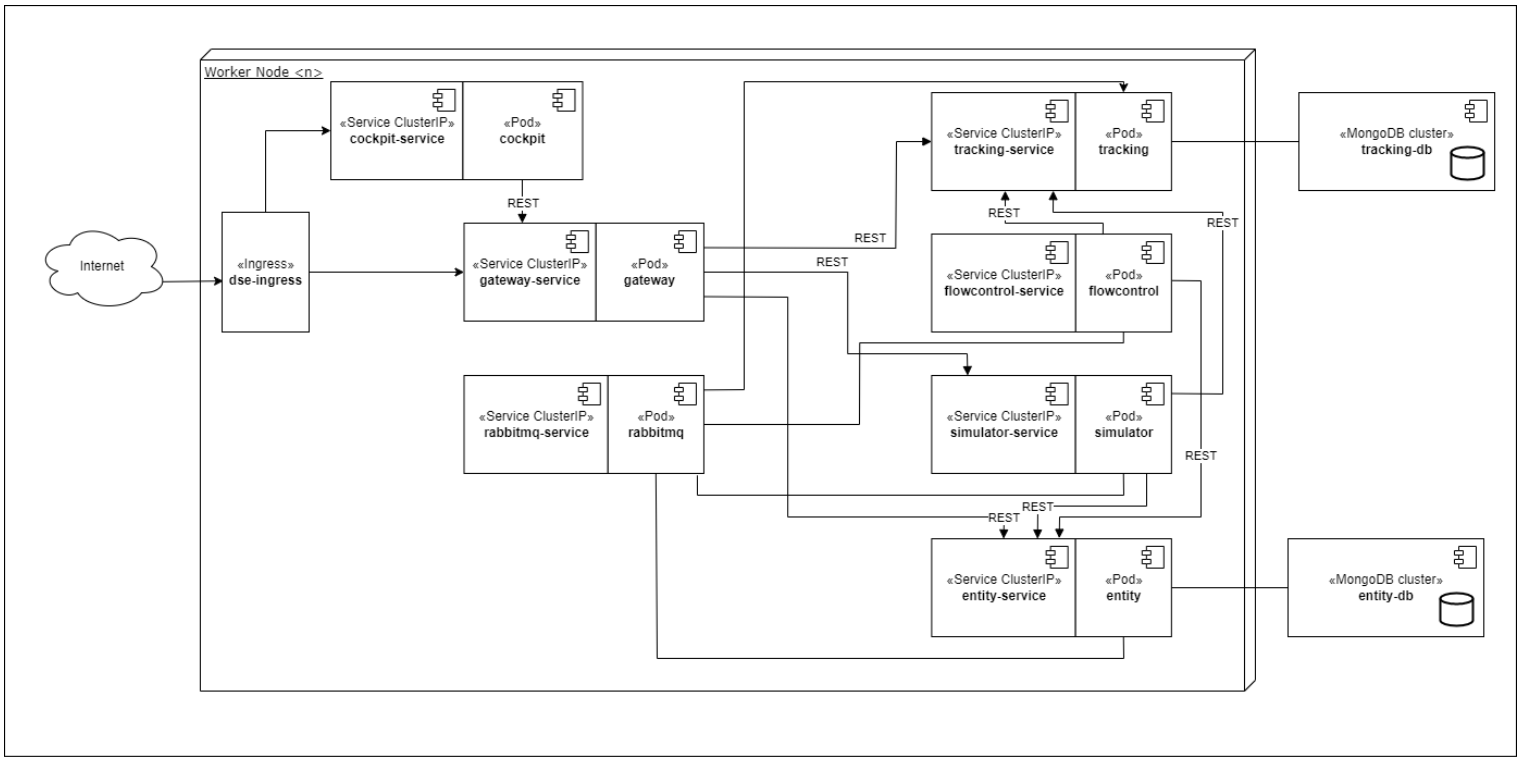

Abbildung 12: Komponentendiagramm