DHBW
Duale Hochschule
Baden-Württemberg
Mosbach

Technische Dokumentation

zum Projekt UltiMap

UltiMap

Vorlesung:	Web-Services
Dozent:	Prof. Alexander Auch
Präsentationsdatum:	23.03.2021
Abgabedatum:	29.03.2021
Gruppenname:	UltiMap
Gruppenmitglieder:	Kilian Krampf (INF19B), Lars Rickert (INF19B) Marcel Wolf (INF19A)

Inhaltsverzeichnis

Organisation

Grundidee hinter Ultimap

Mit unserem Web-Service wollen wir Privat- und Geschäftskunden die Planung und Kostenkalkulation von Reisen vereinfachen. Dafür wurde ein Web-Interface entwickelt, indem der Kunde die gewünschte Start- und Zieladresse sowie ein optionales Fahrzeugmodell oder einen Spritverbrauch eingeben kann.

Die eingegeben Daten werden dann an unseren Web-Service (realisiert als GraphQL API) gesendet. Nach erfolgreicher Routenberechnung wird dem Kunden die Route sowie die Strecke, Dauer der Reise, Temperaturbedingungen, geschätzte Reisekosten und Spritverbrauch angezeigt.

Um die Kundendaten anonym zu behandeln, werden die Einstellungen des Web-Interfaces ausschließlich lokal auf dem Gerät gespeichert.

Gruppenmitglieder

Name	**Kurs**
Kilian Krampf	INF19B
Lars Rickert	INF19B
Marcel Wolf	INF19A

Projektplan

Datum	**Beschreibung**
28.01.2021	Gruppenfindung
28.01.2021 bis 08.02.2021	Brainstorming / Themenfindung und Validierung
08.02.2021 bis 11.02.2021	Konkretisierung des Web-Services
11.02.2021	Themeneinreichung und Bestätigung, Namensfindung “Ultimap” und Logodesign
12.02.2021	Aufgabenverteilung, Festlegung von Schnittstellen, Auswahl von Technologien und APIs

13.02.2021	Erstellung Grundgerüst, Fertigstellung GraphQL-Schemata
18.02.2021, 22.02.2021 und 25.02.2021	Zwischenstand besprechen, weiteres Vorgehen planen
18.03.2021	Fertigstellung BPNM
19.03.2021	Fertigstellung Web-Service, Fertigstellung Web-Interface
20.03.2021	Integrationstest von Web-Service und Web-Interface
24.03.2021	Fertigstellung Dokumentation
23.03.2021	Präsentation
29.03.2021	Abgabe

Aufgabenverteilung

Neben der Einteilung in die folgenden “Hauptbereiche” gibt es dennoch Überschneidungen, die im Team bearbeitet wurden.

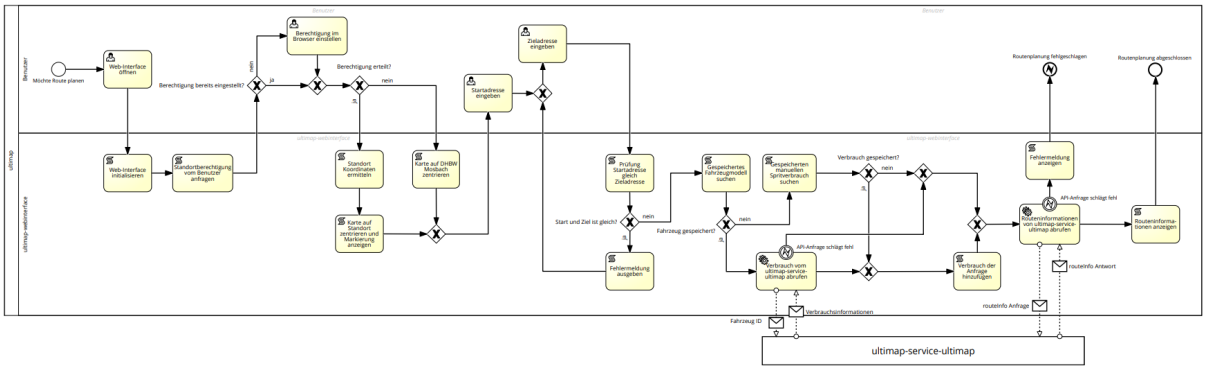
Name	**Aufgabenbereich**
Kilian Krampf	Webservice Ultimap (inkl. Dokumentation), Webservice CarInfo (inkl. Dokumentation), Build-Management
Lars Rickert	Web-Interface (inkl. Dokumentation), Logodesign
Marcel Wolf	Webservice Routing (inkl. Dokumentation), Webservice Weather (inkl. Dokumentation)
Alle	Schnittstellendefinition, BPNM, Integrationstests, restliche Dokumentation, Performancetests

Prozessmodelle

Web-Interface

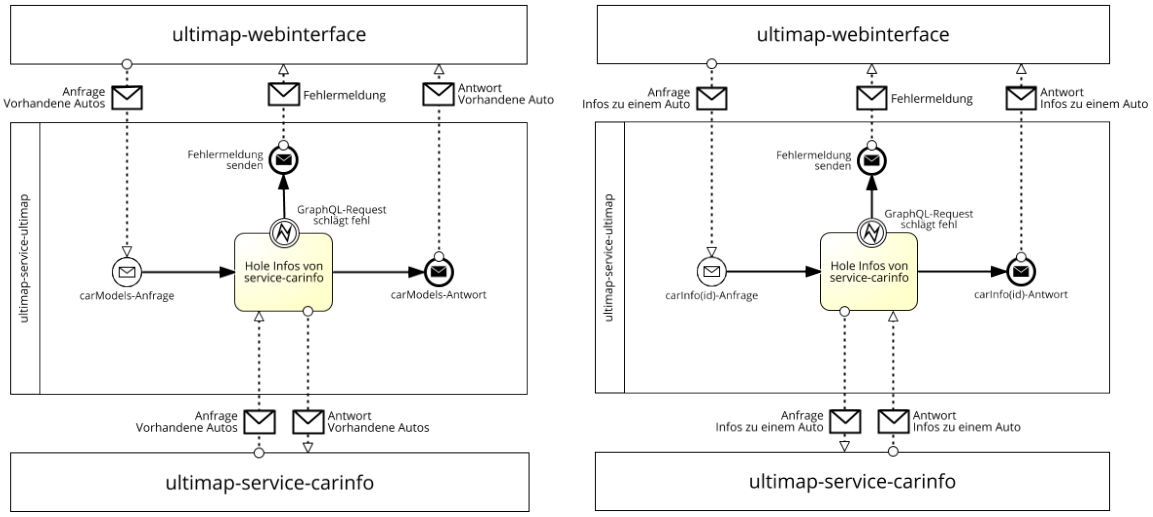
Im Folgenden ist der Ablauf einer Routen-Abfrage als BPNM-Modell zu sehen. Zusammengefasst wird zu Beginn (nachdem der Benutzer das Web-Interface geöffnet hat) die Karte auf den Standort des Benutzers zentriert. Sollte die Standort-Berechtigung nicht erteilt werden, wird sie auf die DHBW Mosbach zentriert.

Nach der Eingabe von Start- und Zieladresse findet die Anfrage an unseren Service Ultimap statt. Dafür wird zuerst geprüft, ob der Nutzer in den Einstellungen ein Fahrzeugmodell oder einen Spritverbrauch angegeben hat. Falls ein Fahrzeugmodell ausgewählt ist, werden die Verbrauchsinformationen vom Service Ultimap abgerufen. Anschließend werden die (optionalen) Verbrauchsdaten sowie Start- und Zieladresse in der Routen-Anfrage an den Service Ultimap geschickt. War die Anfrage erfolgreich werden die Routeninformationen im Web-Interface dargestellt. Ist ein Fehler aufgetreten wird dem Benutzer eine Fehlermeldung angezeigt.

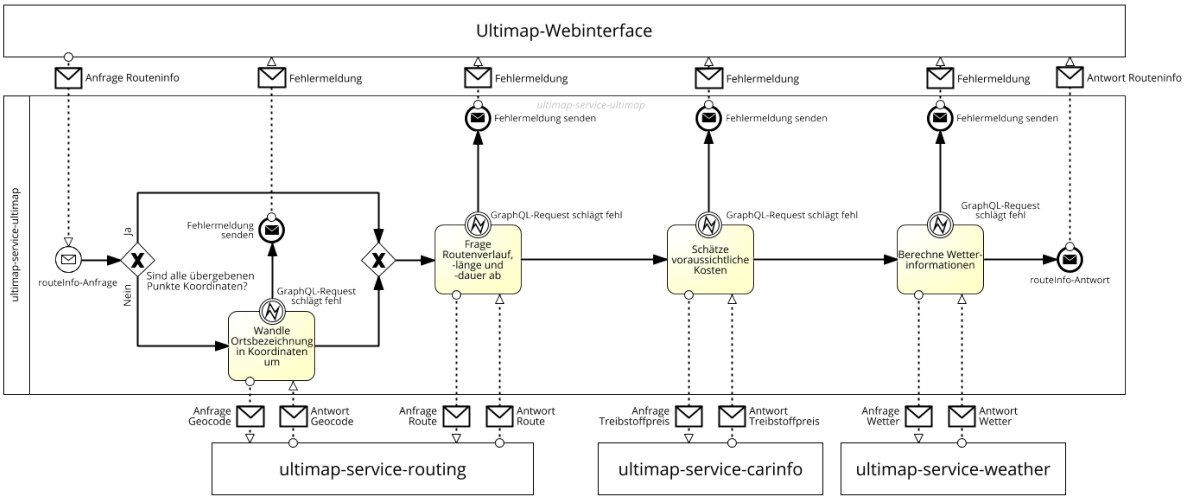


Service Ultimap

Der service-ultimap ist ein kompositioneller Webservice, der die Ergebnisse der Webservices für Routing, Wetter und Fahrzeuginformationen zusammenführt, um ein gemeinsames Ergebnis zu produzieren. Zusätzlich werden noch einige Endpunkte des service-carinfo als Gateway zur Verfügung gestellt. Dies wird getan, damit es nur ein Interface gibt, während alle anderen strukturellen Eigenschaften interne Informationen sind.



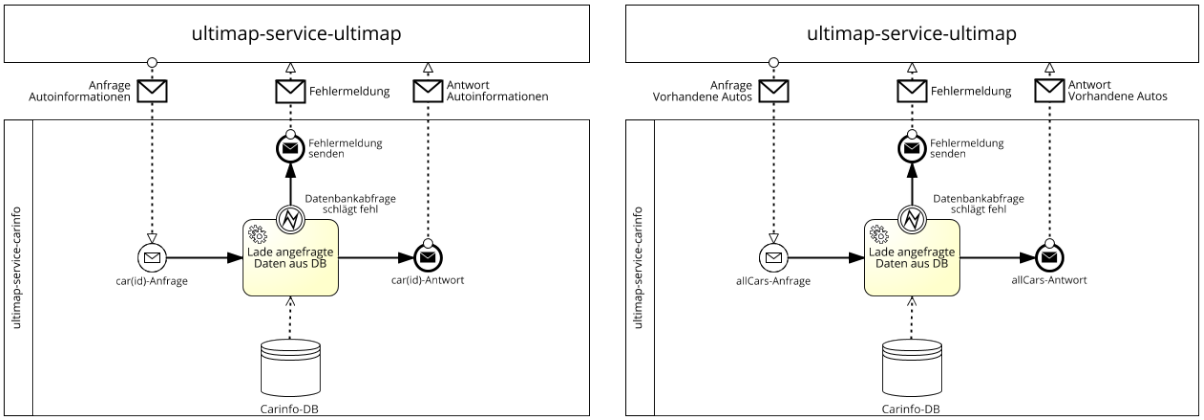
Die Prozesse zur Spiegelung der carinfo-Enpunkte ist denkbar simpel, da die Informationen nicht weiter verarbeitet werden.



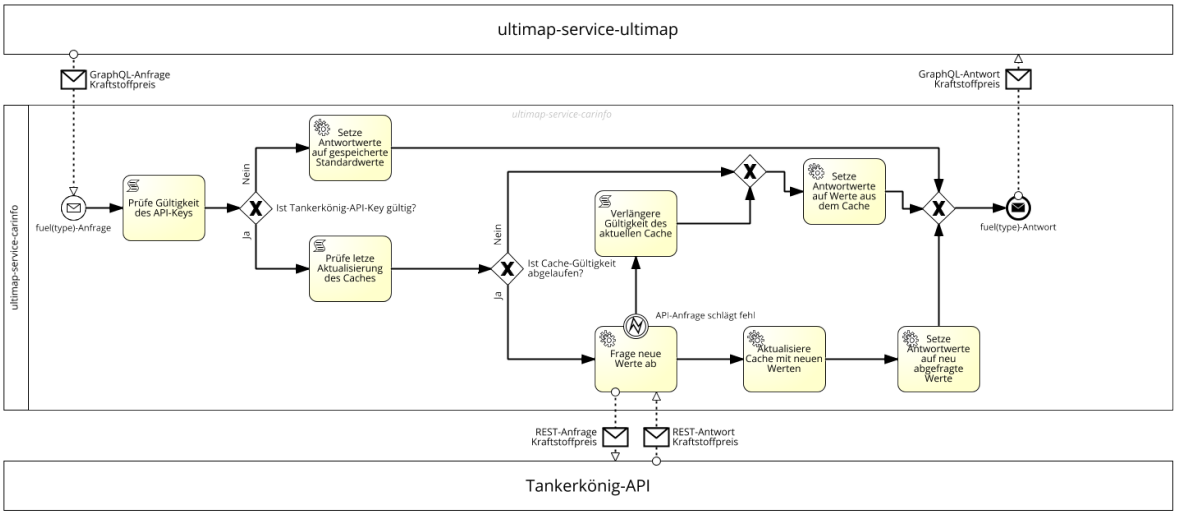
Beim Prozess des routeInfo Endpunkts werden zunächst die eingegebenen Ortsbezeichnungen in geographische Koordinaten umgewandelt. Anschließend wird eine Route zwischen diesen Punkten berechnet. Ausgehend von dieser Route werden die Fahrtkosten abgeschätzt und eine Zusammenfassung des vorhergesagten Wetters erstellt. Diese akkumulierten Informationen werden dann entsprechend der Anfrage zurückgeliefert.

Service CarInfo

Der service-carinfo kapselt alle Informationen zum Thema Fahrzeuge oder Kraftstoff. Dafür greift der Webservice auf eine REST-API und eine vorgefüllte Datenbank zu.



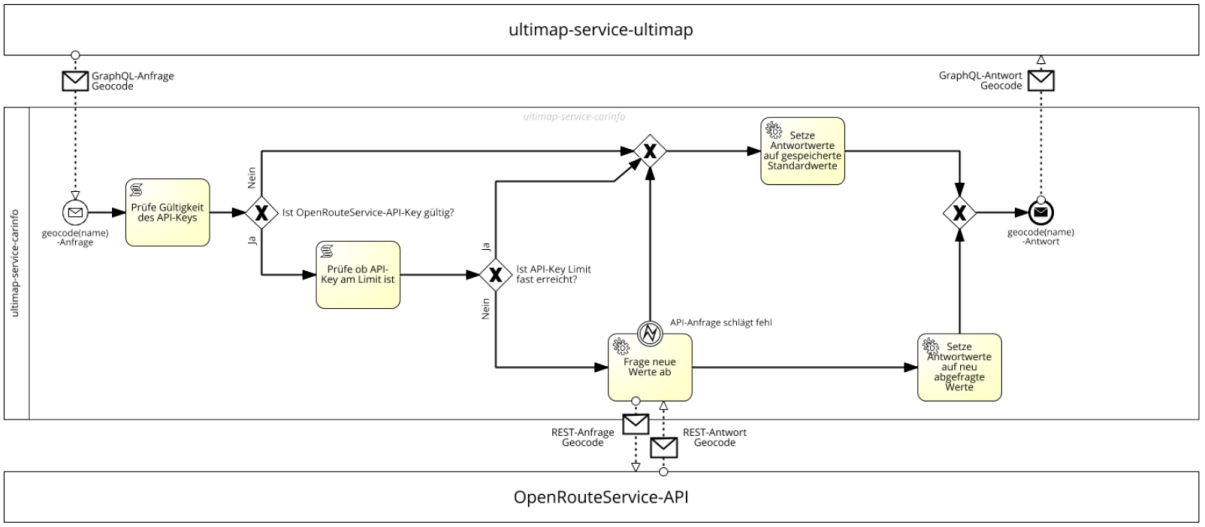
Die Prozesse zum Datenbankzugriff sind sich sehr ähnlich und benötigen aufgrund des einfachen Ablaufs auch keine gesonderte Erklärung.



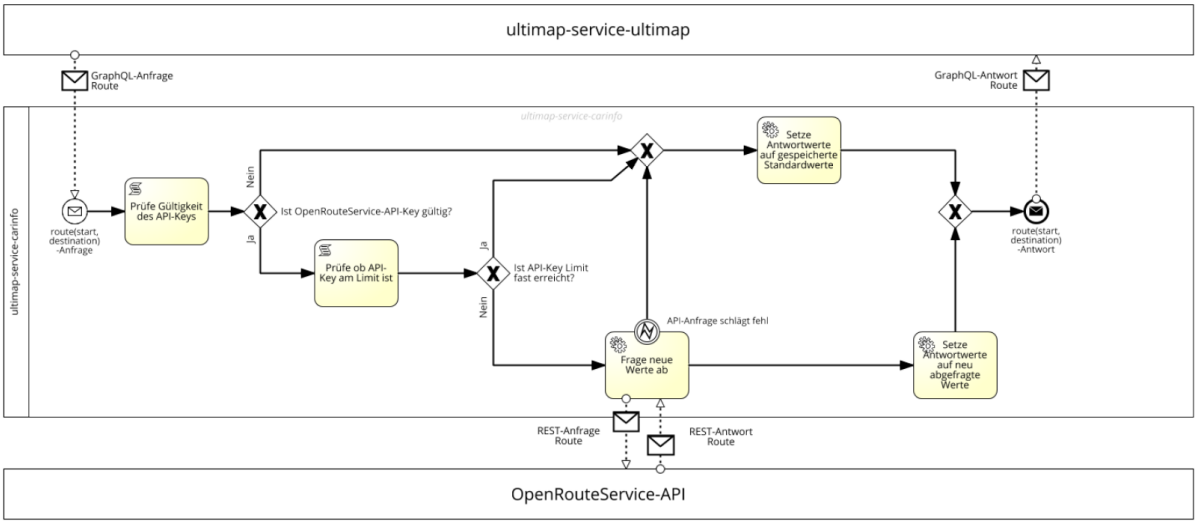
Der fuel-Endpunkt stellt den aktuellen Kraftstoffpreis im GraphQL-Format zur Verfügung. Dabei werden durch diesen Service die Verwaltung des API-Keys, das Caching der Daten und einen Teil der Fehlerbehandlung übernommen. Zunächst wird geprüft ob der vorhandene API gültig ist und ob die gecachten Daten noch gültig sind. Anschließend werden die Daten gegebenenfalls aktualisiert und zurückgegeben. Sollte bei der Aktualisierung etwas schiefgehen, so findet ein Fallback auf die letzten Daten oder gespeicherte Standarddaten statt.

Service Routing

Der service-routing ist für alle geographischen Themen zuständig. Hierfür greift der Web-Service auf eine REST-API von OpenRouteService zu.



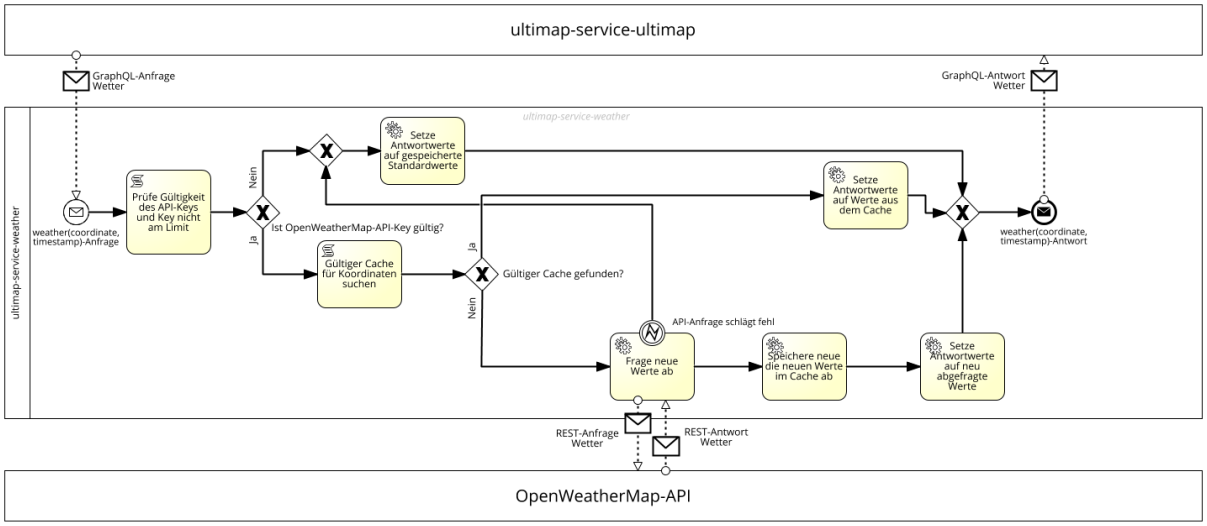
Zunächst wird ein Endpunkt benötigt um den Start- und Zielort zu verarbeiten. Dazu wird die Eingabe im Textformat, in Koordinaten umgewandelt. Hierfür ist der geocode-Endpunkt zuständig. Dieser Prüft auch ob der API-Key gültig ist und verhindert intern, dass der API-Key nicht übernutzt und dadurch ungültig wird. Falls hierbei oder bei der Anfrage an den OpenRouteService Probleme auftreten werden Standarddaten zurückgegeben.



Selbstverständlich wird noch ein Endpunkt benötigt um die Route zu ermitteln. Dazu werden im GraphQL-Format die zuvor ermittelten Koordinaten vom Start- und Endpunkt an den route-Endpunkt übermittelt und dann an den OpenRouteService weitergeleitet. Als Antwort wird die Route in Form einer Sammlung von Koordinaten erhalten. Der restliche Ablauf entspricht dem vom geocode-Endpunkt.

Service Weather

Der service-weather verwaltet die Wetterdaten der Routen. Dazu greift dieser Web-Service auf eine REST-API von OpenWeatherMap zu.



Der weather-Endpunkt stellt im GraphQL-Format die benötigten Wetterdaten zur Verfügung. Unter anderem sind dies die Temperatur und der erwartete Niederschlag in Millimeter. Diese sind von Koordinaten und einem gewissen Zeitpunkt abhängig. Der Service verarbeitet die Koordinaten und den Timestamp, prüft und schützt den API-Key, fragt die Daten bei der OpenWeatherMap an und gibt diese oder im Fehlerfall standardwerte zurück. Jede neue Anfrage an unseren Weather Service löst eine Reihe von Abfragen aus. Wurden die API-Key Prüfungen durchlaufen, wird im Cache nach bereits noch gültigen Einträgen gesucht. Die Wetterdaten im Cache werden zusammen mit den Koordinaten gemappt und abgespeichert. Damit ein Cache dazu geeignet ist, als Antwort an unseren zentralen Ultimap-Service zurückgegeben werden zu können, darf er nicht älter als die in der `application.properties` angegebenen Zeit sein. In unserem Fall sind das 10 Minuten. Außerdem reicht es, wenn die Koordinaten der neuen Anfrage, nicht weiter als 10 Kilometer von den gecachten Daten entfernt sind. Treffen mehrere Cache-Daten auf diese Eigenschaften zu, wird der zeitlich aktuellste Cache verwendet. Werden keine gültige Wetterdaten im Cache gefunden, wird eine Anfrage an die OpenWeatherMap gestartet. Bei erfolgreichen Antworten werden diese auch gecached und bevorzugt verwendet.

Implementierung

Build-Management

Als Buildsystem wird bei den Webservices Gradle eingesetzt. Das Webinterface wird mit dem Angular-Framework gebaut, welches ein eigenes Build-System hat, das auf node.js basiert.
Zusätzlich wird das Projekt automatisiert über Gitlab-CI gebaut und getestet. Dabei werden für jeden Commit die Projekte gebaut (mit Gradle, bzw. Angular) und anschließend entsprechend der Teststrategie getestet.
Falls der Commit auf den `main`-Branch getätigt wurde, so kommt zur Pipeline noch die Phase `Dockerize` hinzu. In dieser Phase werden das Webinterface, die 4 Webservices und die vorbereitete Datenbank jeweils in ein Docker-Image gepackt. Die entstehenden Images werden in eine Docker-Registry gepusht, die Gitlab bereitstellt. Dabei werden die Images jeweils mit dem `latest`-Tag und dem Commit-Hash als Tag gepusht, um eine einfache Form der Versionierung zu erhalten.

Webservices

Die Webservices sind in Java implementiert (stellenweise ergänzt durch Kotlin) mithilfe des Spring Boot Frameworks. Für die Umsetzung von GraphQL mit Spring wird das DGS (Domain-Graph-Service) Framework von Netflix genutzt. Dieses Framework ist auf Schema-First-Development ausgelegt und erlaubt so in Kombination mit einem Gradle-Plugin Boilerplate-Code zu reduzieren.

Das Source-Verzeichnis aller Services ist recht ähnlich aufgebaut:

```
└──main
    ├──filteredResources
    │       application.properties // Spring properties
    ├──java
    │   └──de.dhbw.mosbach.webservices
    │       └──service-name
    │           ├──data      // Database-Connection | ORM
    │           ├──external  // Connection to other Web-Services
    │           └──graphql   // Request handlers
    └──resources
        └──schema
               schema.graphqls  // GraphQL-Schema for this service
```

DGS stellt ein Gradle-Plugin zur Verfügung (“`com.netflix.dgs.codegen`”), mit dem Java-Klassen und Client-Code für GraphQL-Schemata erzeugt werden können. Um Anfragen an den eigenen GraphQL-Endpunkt verarbeiten zu können, mappt DGS die GraphQL-Anfragen auf sogenannte DataFetcher (identifiziert durch die `@DgsData`-Annotation). Diese befinden sich im `graphql`-Package.

Der Code um externe Services aufzurufen liegt im `external`-Package, gekapselt in Provider-Klassen. Diese werden per Dependency-Injection den DataFetcher-Objekten bereitgestellt.

Service Ultimap

GraphQL-Schema

```
type Query {
   # Advanced information for the route
   routeInfo(input: UltimapInput): UltimapType

   # Simple tunneling from service-carinfo
   carInfo(carId: Int!): CarInfoType
   carModels: [CarInfoType]!
}

enum FuelType {
   DIESEL
   BENZOL
}

## Input Types
input UltimapInput {
   geopoints: GeoInput! # Start and endpoint
   departure : Int      # UNIX Timestamp in seconds
   fuel: FuelInput      # Information for fuelage
}

input GeoInput {
   start: String!   # Either address or coordinates like (59.0,14.9); Format like (lat,lon)
   destination: String!  # Either address or coordinates like (59.0,14.9)
}

input FuelInput {
   consumption: Float!,
   typ: FuelType!
}

## Response types
type UltimapType {
   route: RouteInfoType!
   costs: CarCostInfoType  # May be null if information is missing
   weather: WeatherInfoType
}

type RouteInfoType {
   start: String!           # Returns the input
   destination: String!     # Returns the input

   duration: Int!               # The estimated duration in minutes
   distance: Int!               # The calculated distance in metres
   waypoints: [CoordinateType]  # The waypoints for the route
}

type CoordinateType {
   lat: Float!,
   lon: Float!
}
```

```
type CarCostInfoType {
   totalConsumption: Float!   # Estimated fuel consumption in litres
   fuelCosts: Float!          # Estimated costs for fuel in EUR
   wearFlatrate: Float!       # Estimated costs for general car wear in EUR
}

type WeatherInfoType {
   min: Float!  # The lowest estimated temperature in degrees of Celsius
   max: Float!  # The highest estimated temperature in degrees of Celsius
   avg: Float!  # The average estimated temperature in degrees of Celsius

   rain: Float!  # The percentage of estimated rain coverage
}

type CarInfoType {
   id: Int!,           # Incremental ID
   name: String,       # Human readable name (e.g. "VW Golf 6 GTI")
   consumption: Float, # Consumption in litres per 100 km
   typ: FuelType       # Consumed Fuel type
}
```

Implementierung

Eine Besonderheit des Service-Ultimap ist, dass GraphQL-Queries an die anderen Services gesendet werden müssen. Hierfür wird die Klasse RestTemplate (bereitgestellt von Spring) verwendet. Zusätzlich wurde Client-Code durch das DGS-Codegen-Plugin generiert. Hierfür liegen die Schemata der Services, die angesprochen werden, im Verzeichnis `resources/graphql/client/{serviceName}`. In der Gradle-Config werden dem Codegen-Plugin diese Ordner übergeben, um Clients dafür zu erzeugen.
Der Code für die eigentliche Anfrage ist in der Datei `GraphQLHelper.kt` implementiert.
Der `routeInfo` Endpunkt wird im `UltimapDataFetcher` behandelt. Die anderen beiden Endpunkte im `CarinfoDataFetcher`. Die Logik für diese Endpunkte ist auch im jeweiligen `DataFetcher` abgelegt. Die Logik für die Abfrage anderer Endpunkte hingegen befindet sich in den `Provider`-Klassen im `external`-Package. Die Verknüpfung erfolgt über Dependency-Injection mit Spring.

Teststrategie

Die vorhandene Teststrategie testet den Webservice nur mit gemockten Daten, jedoch mit unterschiedlichen Ansatzpunkten.

In der Klasse `ProvidersTest` werden die `Provider`-Klassen getestet. Dazu werden mithilfe der Spring Test-Utility die anderen Services simuliert.

In der Klasse `UltimapGraphQLTest` wird der korrekte Umgang mit GraphQL-Requests getestet. Dort werden die Provider jedoch durch Instanzen ersetzt, die nur konstante Werte zurückgeben.

Die Klasse `ServiceUltimapApplicationTests` testet nun die gesamte Applikation und die Antworten des Service.

Service CarInfo

GraphQL-Schema

```
type Query {
   # Information for a single car by carId
   car(carId: Int!): CarInfoType
   # Current pricingInformation for a certain Type of Fuel
   fuel(typ: FuelType!): FuelPriceType!
   # Information of all cars in the database
   allCars: [CarInfoType]!
}

# The supported types of fuel
enum FuelType {
   DIESEL
   BENZOL
}

## Return types
type CarInfoType {
   id: Int!,           # Incremental ID
   name: String,       # Human readable name (e.g. "VW Golf 6 GTI")
   consumption: Float, # Consumption in litres per 100 km
   typ: FuelType       # Consumed Fuel type
}

type FuelPriceType {
   price: Float!
}
```

Implementierung

Der Carinfo-Service stellt einen Adapter für die Tankerkönig-API und eine Datenbank dar. Zum Zugriff auf die Datenbank wird `spring-data-jpa` mit Hibernate für ORM genutzt.
Die Informationen für die Datenbankverbindung wird in der `application.properties`-Datei angegeben. Die eigentliche Abfrage der Daten wird nun automatisch von Spring implementiert, als Realisierung des `ICarRepository`-Interface. Auf das Objekt mit dieser Implementierung wird per Dependency-Injection zugegriffen.
Die Anfrage des Fuel-Price, sowie das Caching sind in der Klasse DefaultFuelPriceProvider implementiert. Dort ist auch die Prozesslogik abgelegt, die im BPMN-Modell beschrieben ist. Bei der Anfrage an die API wird das RestTemplate von Spring genutzt um die Antwort zu einem Java-Objekt zu übersetzen. Sollte bei einer Preisabfrage etwas schiefgehen, so werden die letzten Informationen erneut verwendet.
Da nicht alle Informationen benötigt werden, die die externe API bereitstellt, werden nur die relevanten Daten in den Java-Objekten abgebildet und die übrigen Daten ignoriert und verworfen.

Teststrategie

Die vorhandene Teststrategie testet den Webservice nur mit gemockten Daten, jedoch mit unterschiedlichen Ansatzpunkten.

In der Klasse `ProvidersTest` wird der API-Zugriff, Caching und der Datenbankzugriff getestet. Dazu werden mithilfe der Spring Test-Utility die API simuliert und eine H2 Datenbank (in Memory) aufgesetzt. Die Cache-Gültigkeit, sowie der API-Key werden in der application.properties aufgesetzt, die speziell für die Tests genutzt wird.

In der Klasse `CarinfoGraphQLTest` wird der korrekte Umgang mit GraphQL-Requests getestet. Dort werden die Provider jedoch durch Instanzen ersetzt, die nur konstante Werte zurückgeben (definiert in MockObjects).

Die Klasse `ServiceCarinfoApplicationTests` testet nun die gesamte Applikation und die Antworten des Service.

Service Routing

GraphQL-Schema

```
type Query {
   route(start: CoordinateInput!, destination: CoordinateInput!): RouteType!
   geocode(name: String!) : CoordinateType!
}

## Input types
input CoordinateInput {
   lat: Float!
   lon: Float!
}

## Return types
type CoordinateType {
   lat: Float!
   lon: Float!
}

type RouteType {
   time: Int!                  # Time in minutes
   distance: Int!              # distance in metres
   waypoints: [CoordinateType] # Waypoints
}
```

Implementierung

Der Weather-Service stellt einen Adapter für die OpenRouteService-API dar.
Die Anfrage des Wetters wird in zwei Endpunkte gespalten. Die Anfrage sowie das Caching sind in der Klasse `DefaultRoutingProvider` implementiert. Dort ist auch die Prozesslogik abgelegt, wie im BPMN-Modell beschrieben.
Bei der Anfrage an die API wird hier auch wieder das RestTemplate von Spring genutzt um die Antwort zu einem Java-Objekt zu übersetzen. Sollte bei einer Abfrage an die OpenWeatherMap-API ein Fehler auftreten, so werden Standardwerte verwendet.
Da nicht alle Informationen benötigt werden, die die externe API bereitstellt, werden nur die relevanten Daten in den “Response Java-Objekten” abgebildet und der Rest ignoriert.

Service Weather

GraphQL-Schema

```
type Query {
   # coordinate is to give the point to determine the weather at
   # time is a UNIX Timestamp in seconds
   weather(coordinate: CoordinateInput!, timestamp: Int): WeatherType!
}

## input types
input CoordinateInput {
   lat: Float!,
   lon: Float!
}

## return types
type WeatherType {
   temp: Float!,   # Temperature in degrees of Celsius
   rain: Float!    # Amount of rain in mm
}
```

Implementierung

Der Weather-Service stellt einen Adapter für die OpenWeatherMap-API dar.
Die Anfrage des Wetters, sowie das Caching sind in der Klasse `DefaultWeatherProvider` implementiert. Dort ist auch die Prozesslogik abgelegt, die im BPMN-Modell beschrieben ist.
Bei der Anfrage an die API wird das RestTemplate von Spring genutzt um die Antwort zu einem Java-Objekt zu übersetzen. Sollte bei einer Abfrage an die OpenWeatherMap-API etwas schiefgehen, so werden Standardwerte verwendet.
Das Attribut, in dem der Niederschlag in Millimeter gespeichert wird, heißt in der REST JSON-Response `"1h"`. Dies ist in Java an ungültiger Variablenname. In diesem Fall, kann das Attribut nach belieben umbenannt werden und mit dem Dekorator `JsonProperty("1h")` versehen werden.
Da nicht alle Informationen benötigt werden, die die externe API bereitstellt, werden nur die relevanten Daten in den “Response Java-Objekten” abgebildet und der Rest ignoriert.

Web-Interface

Da die Benutzung der Ultimap Services benutzerfreundlich gestaltet werden soll, wurde ein Web-Interface (also eine Website) erstellt. Diese ermöglicht dem Nutzer eine Routen-Berechnung durch die Eingabe von Start- und Zieladresse sowie die Auswahl eines Fahrzeugmodells oder benutzerdefiniertem Spritverbrauch. Die Eingabe der Fahrzeugdaten ist dabei optional und kann für die Kostenkalkulation der Route eingetragen werden.

Realisiert wird das Web-Interfaces durch zwei Haupt Technologien: Angular und Capacitor.

Angular Framework

Angular ist ein **JavaScript Framework**. JavaScript ist die für Web-Anwendungen am weitesten verbreitete Skriptsprache / Programmiersprache. Sie wird häufig zusammen mit HTML und CSS verwendet. HTML (Hypertext Markup Language) ist dabei eine Auszeichnungssprache, mit der die Struktur der Web-Anwendung bestimmt wird. CSS (Cascading Style Sheets) ist für die optische Gestaltung des HTMLs zuständig.

Mithilfe von Angular ist es möglich, Inhalte und Logik der Web-Anwendung in **Komponenten** zu kapseln und beliebig oft wiederzuverwenden. Im HTML können außerdem JavaScript und spezielle Angular Ausdrücke platziert und somit einfacher **dynamische bzw. bedingte Inhalte** erstellt werden. Zudem werden dem Entwickler verschiedene Funktionen, wie z.B. das einfachere navigieren etc. geboten (siehe Angular Dokumentation für weitere Informationen).

Des Weiteren verwendet Angular Typescript (.ts Dateien). TypeScript ist eine auf JavaScript aufbauende Programmiersprache von Microsoft. Sie bietet gegenüber JavaScript ein Typensystem, sodass z.B. Variablen ein eindeutiger Typ zugewiesen werden kann. Somit können Fehler einfacher während der Entwicklung vorgebeugt oder behoben werden. Da TypeScript auf JavaScript aufbaut, ist jeder JavaScript Code auch gültiger TypeScript Code. Damit TypeScript Code allerdings im Browser läuft, muss er zuerst durch den TypeScript Compiler in JavaScript Code transpiliert werden. Dies übernimmt der Angular Compiler.

Ist die Anwendung bereit zur Produktion wird die Anwendung durch den Command Line Befehl "***ng build --prod***" von Angular gebaut, in JavaScript transpiliert und alle benötigten Ressourcen gebündelt in einem Verzeichnis (standardmäßig "dist") abgelegt. Die dabei entstehende Web-Anwendung ist eine sog. Single Page Application (SPA). Anders als bei "herkömmlichen" Web-Anwendungen, bei dem jede Seite als dedizierte HTML-Datei realisiert ist, gibt es bei SPAs nur eine HTML-Datei. Die Inhalte werden dann ausschließlich durch JavaScript verändert. Der Nutzer (Client) ruft also beim Laden die gesamten Inhalte und Funktionalität ab.

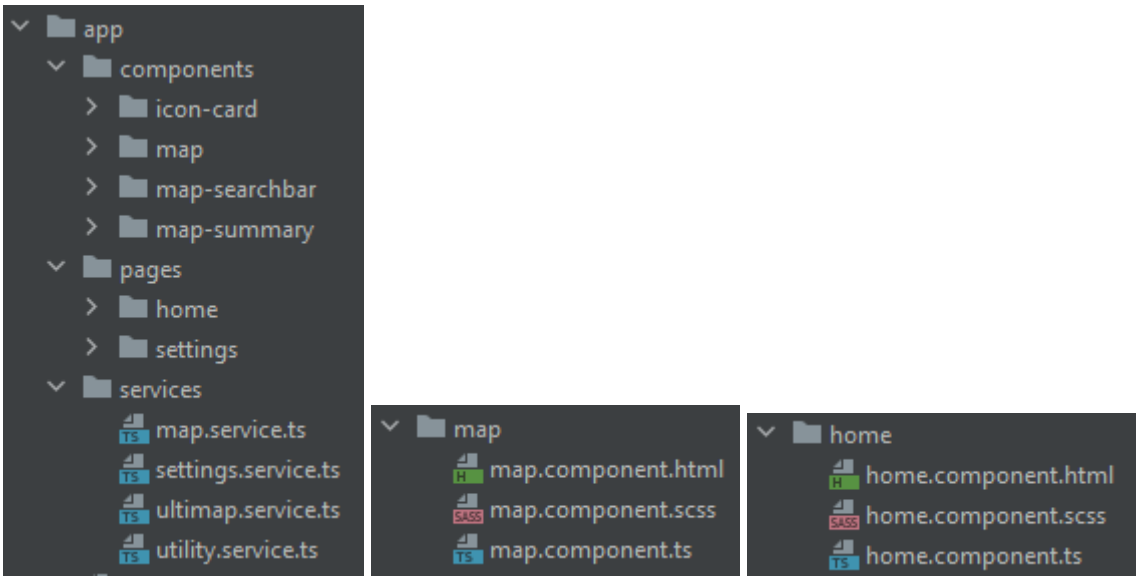
Capacitor

Um das Web-Interface ggf. auch über eine native Android und iOS App bereitzustellen wird Capacitor verwendet. Hierbei kann eine native (hybrid) App generiert werden, in der dann das Web-Interface wie in einem Browser läuft. Zusätzlich werden einige native Funktionen

geboten, die auf allen Plattformen verwendet werden können. Darunter zählt z.B. der native Zugriff auf die den Standort, das Dateisystem oder die Sensoren des Geräts.

Aus Vereinfachungsgründen haben wir das Web-Interface zwar auf die Web-Anwendung beschränkt, dennoch wird Capacitor für die Standorterkennung und das Speichern der Einstellungen auf dem Gerät des Benutzers verwendet. So sind alle Funktionen des Web-Interfaces auch potentiellen in hybrid Apps verfügbar.

Struktur



Komponenten:
In den obigen Abbildungen ist eine gekürzte Darstellung der Struktur des Web-Interfaces dargestellt. Hierbei lassen sich drei Hauptverzeichnisse erkennen: **components** (Komponente), **pages** (Seiten) und **services** (Services).

Die Verzeichnisse components und pages enthalten jeweils die bereits erwähnten Angular-Komponenten. Im Falle der “icon-card” wird diese an mehreren Stellen des Web-Interfaces verwendet. Die restlichen Komponent beinhalten umfangreichere Funktionen und sind daher gekapselt. Obwohl die Komponenten der beiden Verzeichnisse technisch alles Angular-Komponenten sind, sind die pages-Komponenten nicht zur Mehrfachen Verwendung gedacht, sondern fungieren als “Ersatz” für herkömmliche Seiten (einzelne HTML-Dateien) und besitzen eine URL, über die sie gezielt erreichbar sind.

Wie auf den Abbildungen zu sehen, besteht jede Komponente aus einer HTML, SCSS (Erweiterung von CSS) und TS Datei und können so strukturiert, gestaltet und mit Funktionen bereichert werden. Das CSS ist dabei **scoped**, d.h. nur innerhalb der jeweiligen Komponente effektiv (siehe Angular Dokumentation für weitere Informationen).

Services:

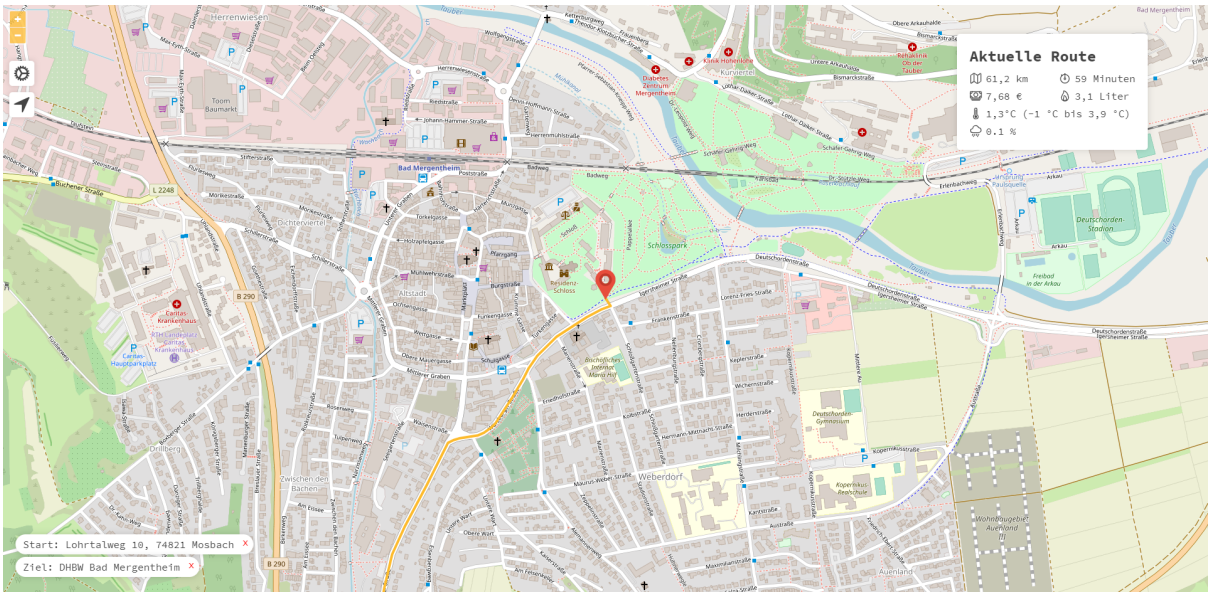
Angular-Services sind sog. “Injectables”, d.h. es existiert jeweils nur eine einzige Instanz des Services in der ganzen Anwendung. Sie sind vergleichbar mit Singletons aus anderen Programmiersprachen. Services können bei Bedarf in den Konstruktor einer Komponente “injected”, also zur Verfügung gestellt werden. Die Services enthalten in der Regel zentrale Steuerungslogik der App, die in beliebig vielen Komponenten benötigt wird. Hinweis: Die Angular-Services sollten nicht mit den Web-Services von Ultimap verwechselt werden, die Teil des Backends sind.
Die vier Services des Web-Interfaces stellen u.A. die Erstellung und Steuerung der Karte, Anfragen an die Ultimap GraphQL API sowie die Verwaltung von lokal gespeicherten Daten global zur Verfügung.

Funktionen

Auf der Startseite des Web-Interfaces findet sich eine vollflächige Karte, die initial auf den Standort des Nutzers (sofern dieser die Berechtigung dazu erteilt) zentriert und mit einer Markierung versehen ist. Der Standort wird hierbei wie bereits erwähnt mit hilfe von Capacitor abgerufen, damit die Abfrage auch auf nativen Apps möglich ist. Aus Privatsphäre-Gründen ist dieser hier nicht zu sehen. Falls keine Standort-Berechtigung erteilt wurde ist die Karte standardmäßig auf die DHBW Mosbach zentriert.

Für die Darstellung der Karte wird die Openlayers Library verwendet. Sie ermöglicht es unter Anderem die Karte darzustellen sowie eine Route und Markierungen einzutragen. Als Kartenmaterial (sog. “Tile Server”) wird OpenStreetMap verwendet.

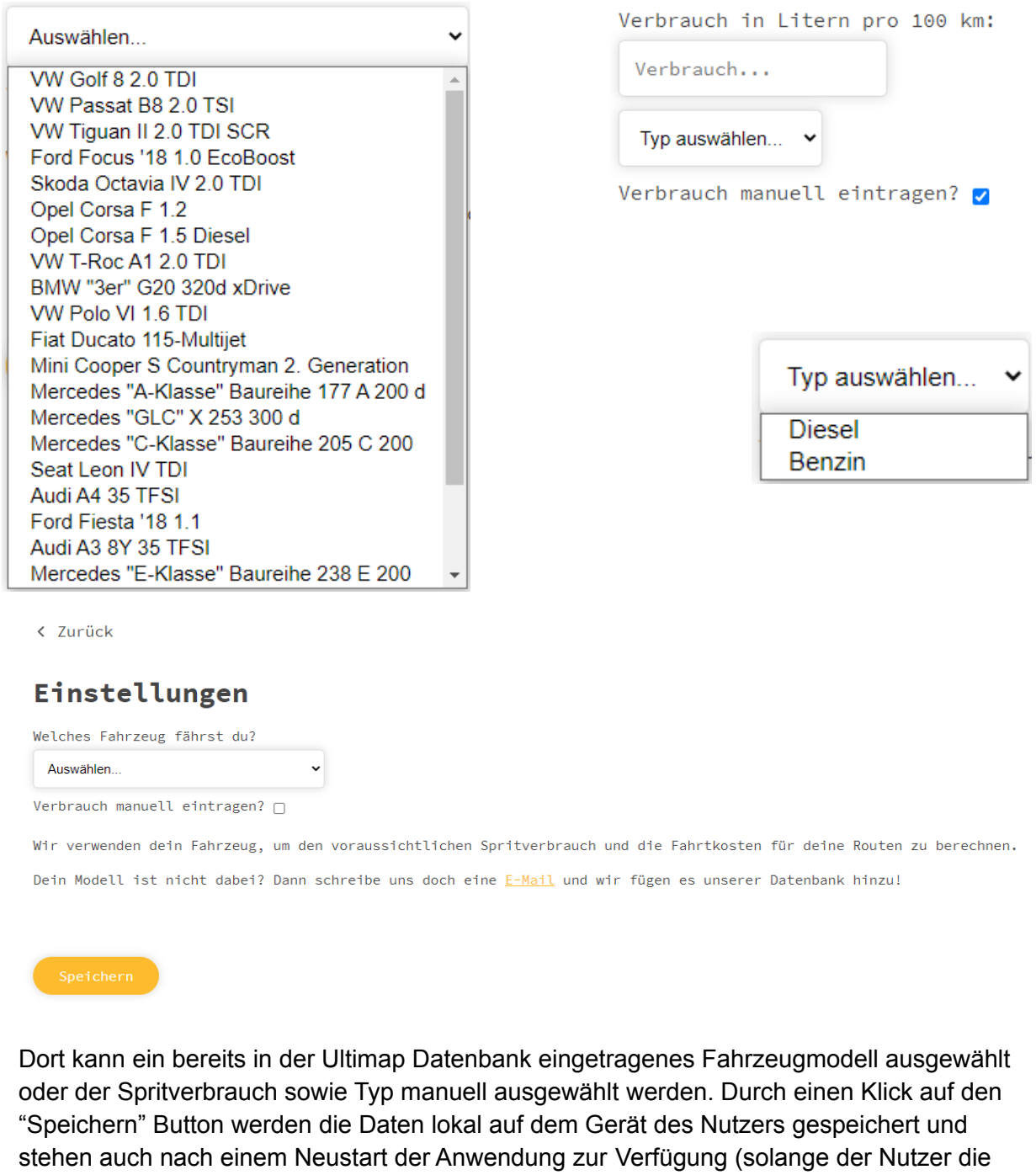
Durch die Eingabeleiste unten links kann der Nutzer eine Start- und Zieladresse eingeben, wodurch nach dem Drücken der Enter Taste oder Klick auf die Lupe die Route automatisch von der GraphQL API abgerufen wird.



Sobald die Route berechnet wurde, wird sie auf der Karte orange angezeigt und es erscheint eine Nachricht am unteren Bildschirmrand. Außerdem zentriert die Karte auf die eingegebene Zieladresse. Oben rechts ist eine Übersicht aller Metadaten der Reise zu sehen. Dargestellt ist die Strecke, voraussichtliche Dauer, Durchschnitts-, Mindest und Maximaltemperatur, Regenwahrscheinlichkeit sowie die geschätzten Gesamtkosten und Spritverbrauch. Damit der Spritverbrauch und die Kosten berechnet werden können, muss vorher ein Fahrzeug bzw. der Spritverbrauch pro 100 km in den Einstellungen ausgewählt werden. Dies wird im Laufe der Dokumentation noch detaillierter erläutert. Die Gesamtkosten setzen sich aus den Kosten für den Spritverbrauch sowie der Verschleiß-Pauschale zusammen.

Ein Klick auf die Kachel zentriert die Karte wieder auf das Reiseziel während ein Klick auf das Standortsymbol oben links die Karte auf den Standort des Nutzers zentriert (falls die Berechtigung erteilt ist). Mit den +/- Symbolen oben links kann die Karte heraus- und hinein gezoomt werden.

Um die Fahrzeugeinstellungen zu ändern, kann der Nutzer auf das Einstellungs-Symbol oben links klicken, um auf die Einstellungsseite zu gelangen.



Dort kann ein bereits in der Ultimap Datenbank eingetragenes Fahrzeugmodell ausgewählt oder der Spritverbrauch sowie Typ manuell ausgewählt werden. Durch einen Klick auf den “Speichern” Button werden die Daten lokal auf dem Gerät des Nutzers gespeichert und stehen auch nach einem Neustart der Anwendung zur Verfügung (solange der Nutzer die Browserdaten nicht leert). Hierfür wird ebenfalls Capacitor verwendet, um die Speicherung sowohl im Web, als auch auf nativen Apps zu ermöglichen. Durch einen Klick auf “Zurück” gelangt der Nutzer wieder auf die Startseite mit der Karte / Routenübersicht.

Codebeispiele

Da die Struktur und Funktionen des Web-Interfaces nun klar sind, sollen zwei Code-Beispiele die Verwendung von Angular und die Interaktion mit der GraphQL API detaillierter erklären.

1. Angular: Interpolation im HTML

Wie bereits bei der Einführung in das Angular Framework angesprochen, können JavaScript Ausdrücke direkt im HTML platziert werden und so z.B. dynamisch Variablen an den entsprechenden Stellen angezeigt werden. Das HTML wird dann automatisch aktualisiert, sobald sich der Wert der Variable ändert. Dieses Feature wird Interpolation genannt.

```
<div *ngIf="routeInfo.costs">
  <ion-icon name="cash-outline"></ion-icon>
  <span>{{ routeInfo.costs.fuelCosts + routeInfo.costs.wearFlatrate | currency: currencyCode: "EUR" }}</span>
</div>

<div *ngIf="routeInfo.costs">
  <ion-icon name="flame-outline"></ion-icon>
  <span>{{ routeInfo.costs.totalConsumption | number: digitsInfo: "1.0-1" }} Liter</span>
</div>
```

In dieser Abbildung ist ein vereinfachter Ausschnitt (ohne CSS-Klassen etc.) der “map-summary” Komponente, also der Kachel, die bei einer aktiven Route oben rechts angezeigt wird und die Metadaten enthält, dargestellt. Optisch handelt es sich um folgenden Ausschnitt:

Im Code ist zu sehen, dass die beiden Elemente ein ***ngIf=”routeInfo.costs”*** innerhalb des öffnenden Tags besitzen. ngIf ist eine Angular-Directive, die ist erlaubt das jeweilige HTML-Element nur anzuzeigen, wenn die boolesche Bedingung true ergibt. Da die Metadaten für den Spritverbrauch/-kosten optional sind, können/sollen sie auch nur angezeigt werden, wenn diese auch in dem routeInfo Objekt vorhanden sind. routeInfo enthält hier alle verfügbaren Daten aus dem bereits gezeigten routeInfo GraphQL Endpunkt.

Innerhalb des divs befindet sich ein Icon aus der Open Source Icon-Bibliothek IonIcons sowie ein -Element, indem sich der eigentliche Wert der Kosten bzw. Spritverbrauch befindet. Die doppelten geschweiften Klammern signalisieren eine Interpolation. Innerhalb diese kann dann ein JavaScript Ausdruck geschrieben werden, der dann im HTML angezeigt wird. Im Fall der Kosten wird hier der Spritpreis und die Verschleiß-Pauschale addiert. Beim Spritverbrauch wird lediglich der Wert aus dem routeInfo Objekt verwendet.

Bei beiden Interpolationen wird zudem eine sog. Angular-Pipe verwendet. Diese sind an dem horizontalen Strich “|” zu erkennen. Pipes sind eine von Angular gebotene Möglichkeit, um Werte einfach zu formatieren, ohne dafür eigene Methoden schreiben zu müssen. Für die Kosten wird die CurrencyPipe verwendet, die eine Zahl als Währung formatiert. Da das Währungssymbol standardmäßig auf Dollar ($) eingestellt ist, wird explizit angegeben, dass € verwendet werden soll. Beim Spritverbrauch ist die DecimalPipe eingesetzt, die eine Zahl mit der angegeben Anzahl an Ganzzahl- und Dezimalstellen formatiert. Die Anzahl der jeweiligen Stellen wird mit dem String im Format ***{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}*** spezifiziert.

2. Requests an die Ultimap GraphQL API

Für die Anfragen an unsere GraphQP API wird Apollo Angular verwendet. Apollo ist ein GraphQL Client für das von uns verwendete Angular Framework. Exemplarisch wird im Folgenden eine Query für die Fahrzeugdaten des vom Nutzer gespeicherten Fahrzeugmodells gezeigt. Wie alle vom Web-Interface getätigten Queries and unsere API befindet sich die Funktionalität im UltimapService (ultimap.service.ts) gekapselt. Aus Vereinfachungsgründen wird die Konfiguration des Clients (z.B. die Server URL etc.) nicht erläutert. Interessierte finden die Konfiguration in der Datei “src/app/graphql.module.ts”.

```
const response = await this.apollo.query<IUltimapCarInfoResponse>( options: {
  query: gql`
    {
      carInfo(carId: ${carSetting.value}) {
        id, consumption, typ
      }
    }
  `,
}).toPromise();
```

Der hier zu sehende Code führt die Query an unseren GraphQL Server aus. Dafür wird der query() Methode des Apollo Clients ein Objekt übergeben, das die Query als String beinhaltet.

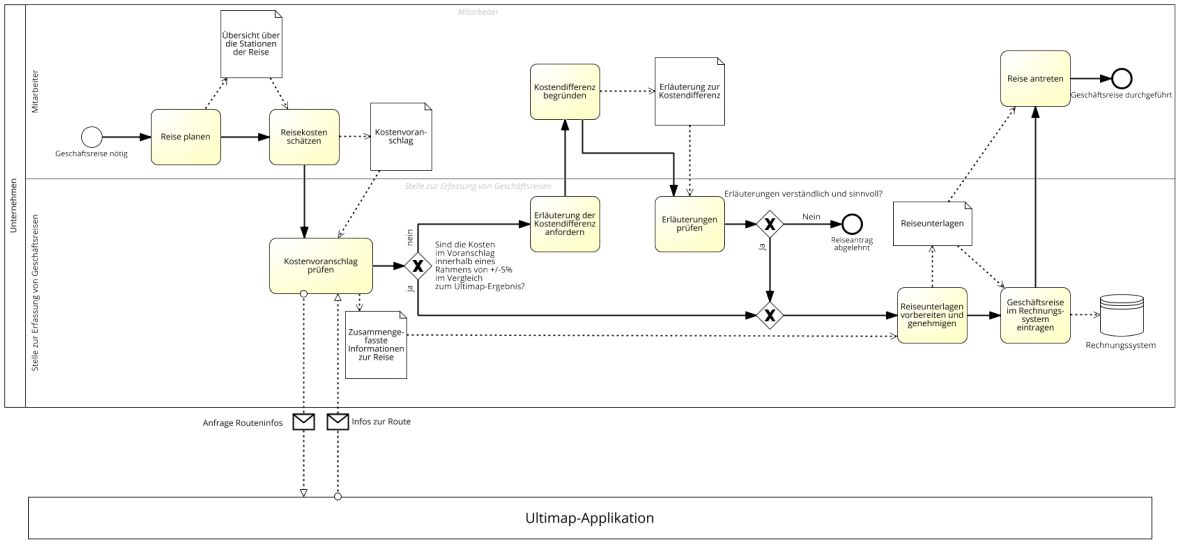
Vor der eigentlichen Query steht das gql template literal. Dieses parsed die folgende Query in einen standard GraphQL AST (Syntaxbaum) und vereinfacht so das Schreiben von GraphQL Queries. Die hier in grün dargestellte Query wird als Template-String geschrieben. Template-Strings in JavaScript sind Strings, die über mehrere Zeilen gesschrieben werden können und JavaScript Ausdrücke durch ${<eigener-js-ausdruck>} beinhalten können.

Hier wird der carInfo Endpunkt abgerufen und die carId des vom Nutzer gespeicherten Fahrzeuges übergeben. Angefordert werden die Felder id, consumption und typ.

Auf diese Weise können beliebige GraphQL Queries geschrieben und ausgeführt werden.

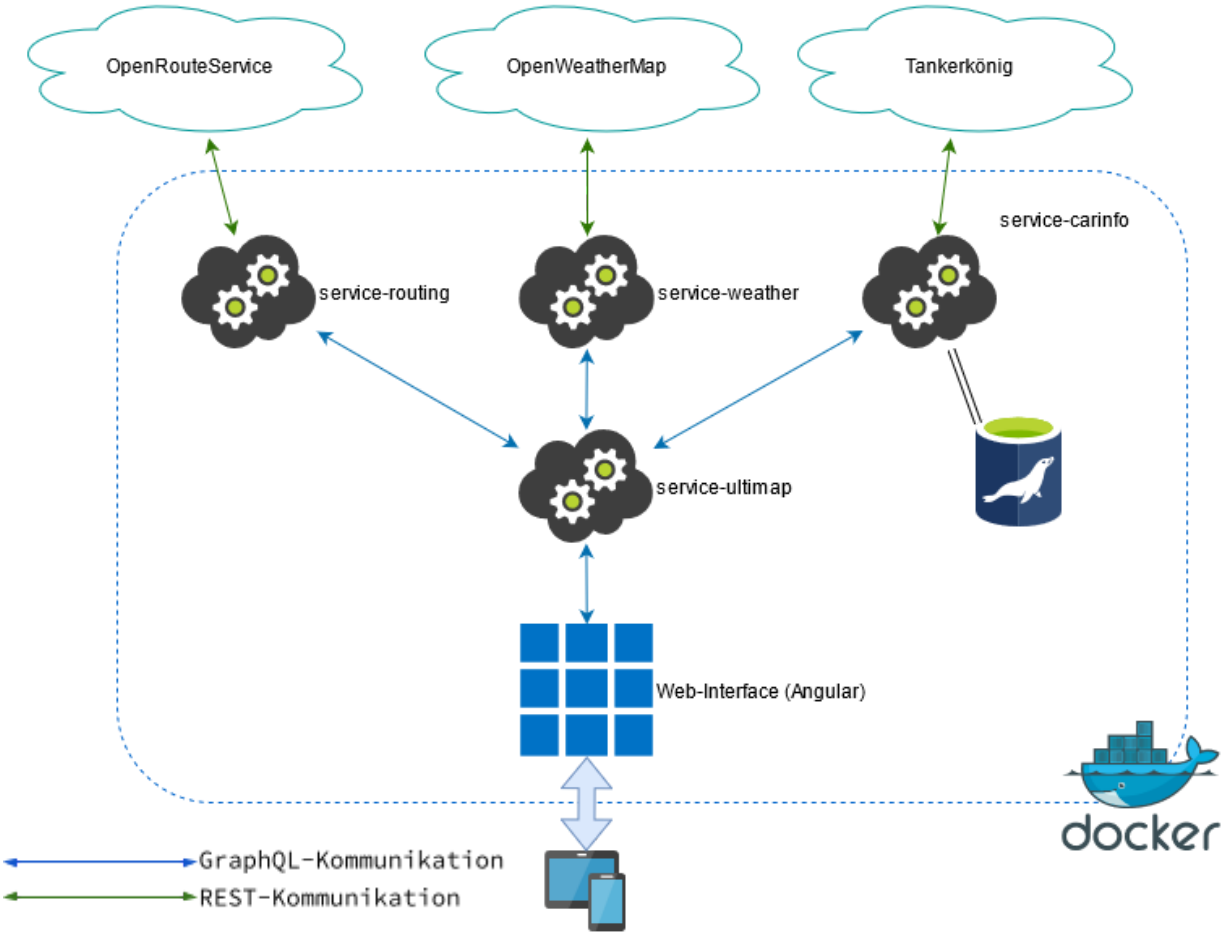
Anhang/Weitere Informationen

Beispiel Use-Case



Ein beispielhafter Business-Prozess in dem Ultimap eingesetzt werden könnte ist die Planung und Genehmigung von Geschäftsreisen. In unserem Beispiel wird Ultimap nur zur Verifikation genutzt, man könnte es aber auch direkt zur Planung einsetzen oder nur zur Kostenberechnung.
Ebenso kann man das Tool zur privaten Reiseplanung nutzen, falls man die Kosten dafür berechnen möchte.

Schematische Darstellung der Architektur



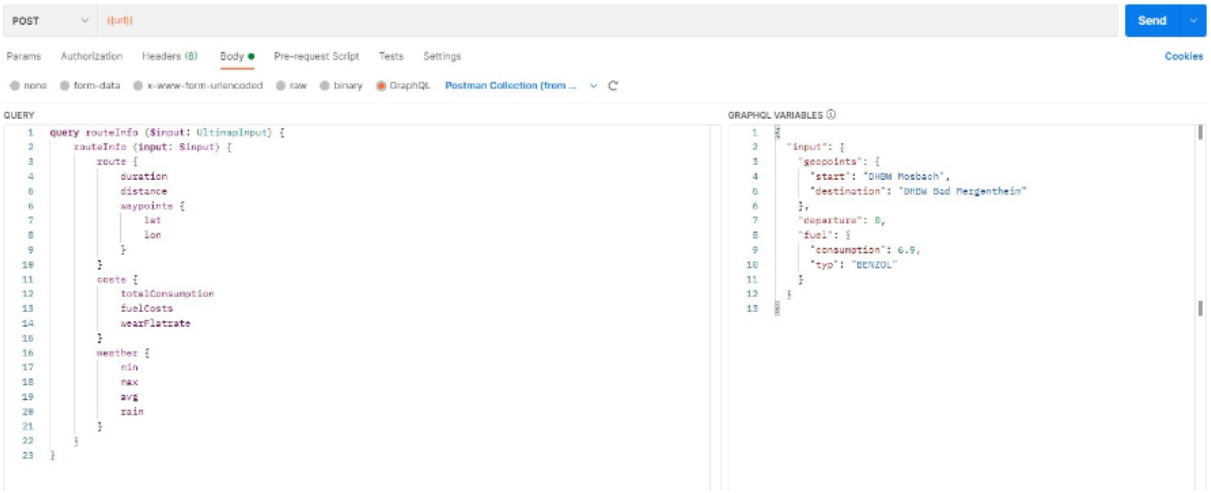
Performancetest mit Postman

Zum Performancetest des Webservices wurde Postman genutzt. Dabei waren keine gültigen API-Keys gegeben, da diese sonst (vermutlich) gesperrt worden wären. Es wurde nur der routeInfo-Endpunkt (im service-ultimap) getestet, da dieser die meiste Funktionalität bietet. Zusätzlich wurden einfache Tests hinzugefügt um zu verifizieren, dass alle Anfragen erfolgreich waren.
Die genauen Ergebnisse können in der Datei `performance_test.json` im Repository angesehen werden. Im folgenden Bild sind die Daten noch einmal zusammengefasst:

Durchschnitt:	68,7 ms
Median:	64 ms
Maximum:	163 ms
Minimum:	51 ms
95%-Quantil	103 ms
99%-Quantil	115,48 ms
Varianz:	299,3434343 ms²
Standardabweichung:	17,30154428 ms

Diese Ergebnisse sind positiv und zeigen eine niedrige Antwortzeit von unseren erstellten Services. Dabei muss bedacht, werden, dass die Anfragen an die externen APIs diese Zeit deutlich erhöhen. Jedoch zeigt sich, dass unsere Prozesslogik wenig Zeit benötigt und somit keine großen Zeitverluste durch unsere Verarbeitung entstehen.

Konfiguration



These tests will execute after every request in this collection. Learn more about Postman's execution order.

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Body matches string", function () {
    pm.expect(pm.response.text()).to.include("{\"data\":{\"routeInfo\":");
});
```

Zusammenfassung von Postman