Namen: Marlon Sido, Felix Pape
GIT: achilles
WEB-ide: tg6-gr2

1. **Projektbeschreibung**

Im Rahmen des Projekts muss eine Anwendung bestehend aus mehreren Microservices realisiert werden. Die Priorisierung wurde dabei in “Prio 1” (muss Kriterium), “Prio 2” (soll Kriterium) und “Prio 3” (kann Kriterium) unterteilt.
Die grundlegende Funktion der Anwendung sollte dabei das Speichern und Abrufen von Autodaten simulieren, indem ein simuliertes Auto Daten an ein Backend sendet. Diese Daten sollen mittels eines Frontends wieder abgerufen werden können.
Dementsprechend musste die Anwendung aus drei teil Anwendungen/Microservices bestehen dessen Grundfunktionalität als Prio 1 gewertet wird. Einem ADLRecorder genannten Service, der alle 2 Minuten Beispieldaten zum Zustand eines Autos auslesen, sie verändern und anschließend an einen weiteren Microservice übertragen kann. Die Funktionlität dieses zweiten Microservices, welcher ADLBackEnd genannt wird, liegt darin die vom Recorder in Form von HTTP-Requests erhaltenen Daten in eine NOSQL Datenbank zu speichern. Dieser Microservice ist in Spring Boot zu entwickeln. Der dritte Microservice wird mit ADLFrontEnd bezeichnet und stellt die Benutzeroberfläche dar.
Es muss die Möglichkeit geben sich einzuloggen. Dem eingeloggten Nutzer muss es möglich sein, sich die zu einer FIN korrespondierenden AutoDaten anzeigen zu lassen. Dazu muss das ADLFrontEnd ebenfalls über HTTP-Requests in Kontakt zu dem ADLBackEnd stehen. Alle Services müssen mit einer Dockerfile containerisierbar und mithilfe von Kubernetes Manifests im Kubernetes zu deployen sein.
Außerdem ist gewünscht (Prio 2), dass das ADLBackEnd per Basic Authentication geschützt wird und es im ADLFrontEnd möglich ist, neben den aktuellsten Daten der letzten zwei Minuten auch die der letzten 4 und 10 Minuten anzuzeigen.
Sollte es die Zeit zulassen, kann ein Account Service erstellt werden, bei dem sich Kunden registrieren können. Anzugebende Informationen sind dabei die FIN und ein Passwort. Zur Persistierung der Daten ist eine Postgresql-Datenbank zu nutzen.
Weiterhin kann ein Login umgesetzt werden, der den Account Service nutzt um Kunden

einzuloggen. Da dies für die Grundfunktionalität nicht erforderlich ist, ist dies mit Prio 3 gewertet.

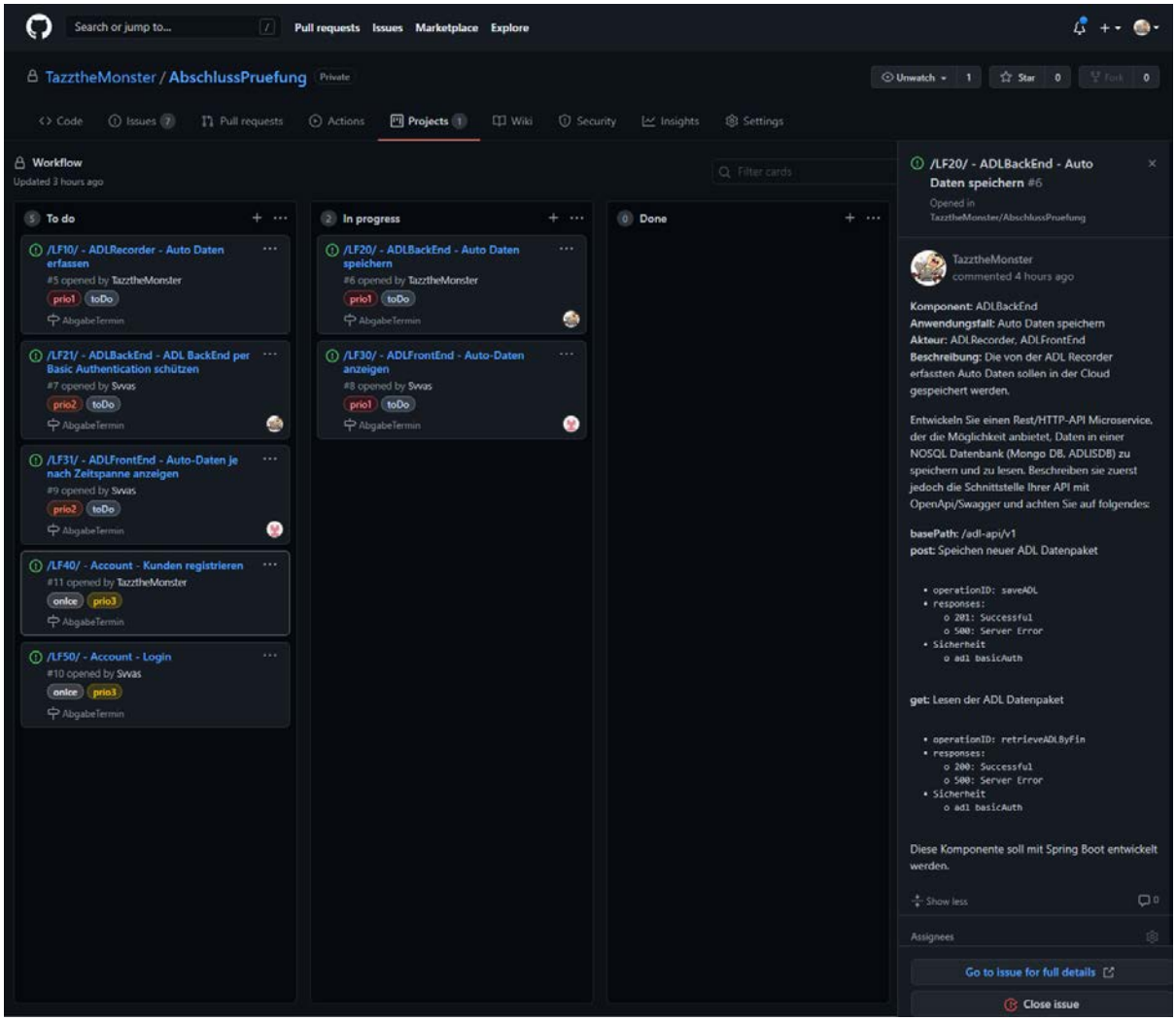
2. **Projektziel**

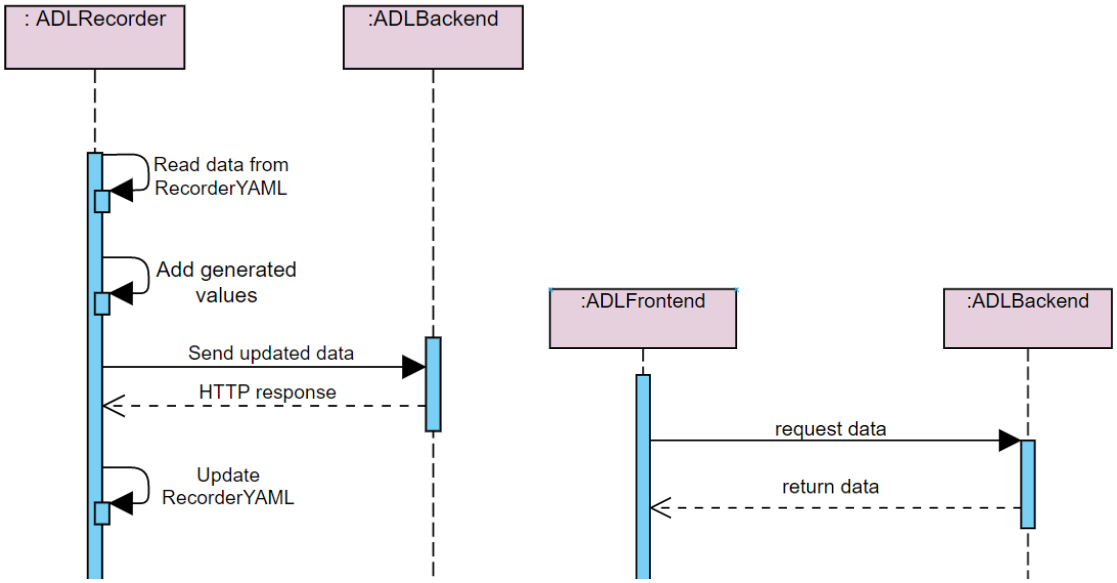
Die Anwendung soll es Fahrzeugbesitzern, Werkstätten und Service Centern ermöglichen, die von einem Auto (in diesem Fall durch den ADLRecorder) generierten Daten auszulesen um diese für unterschiedlichste Zwecke zu nutzen. Ein mögliches im Umfang dieses Projekts nicht enthaltenes Beispiel ist eine Warnung an den Fahrzeugbesitzer bei einem definierten Tankfüllungsgrad oder die Verwendung von KI zur frühzeitigen Erkennung von defekten und ausfällen.

3. **Planungsphase**

Bereits zur Planung der Umsetzung des Projekts wurde ein Git-Repository auf Github eingerichtet. Zum einen da Github diverse uns bekannter und nützlicher Planungstools bietet, zum anderen da am ersten Tag das Cloudogu Repository zu diesem Zeitpunkt noch nicht verfügbar war und wir nicht ohne eines mit der Arbeit beginnen wollten (im weiteren Verlauf des Projektes wurden immer beide Repositorys verwendet und miteinander synchronisiert). In diesem wurden die einzelnen Aufgaben aus der Aufgabenstellung in Issues unterteilt und mit ihrer Priorisierung versehen. Außerdem wurde ein Kanban Board erstellt um sich die Arbeit übersichtlich einteilen zu können. Dies war vor allem deshalb nötig da wir durch die aktuelle Pandemie Situation nicht nebeneinander sitzen sondern uns nur über digitale Kommunikationswege verständigen können.

Ferner wurde die Benennung der Domänenmodelle und deren Membervariablen abgesprochen und auf der Wiki-Seite des Repositories unter "Models" eingetragen. Anschließend wurden weitere Vorgehensweisen geklärt, wie zum Beispiel welche Programmiersprache und welches Framework für welchen Service verwendet werden soll.


Search or jump to...
Pull requests
Issues
Marketplace
Explore
TazztheMonster / AbschlussPruefung
Private
Unwatch 1
Star 0
Fork 0
Code
Issues 7
Pull requests
Actions
Projects 1
Wiki
Security
Insights
Settings
Workflow
Updated 3 hours ago
Filter cards
To do
5
/LF10/ - ADLRecorder - Auto Daten erfassen
#5 opened by TazztheMonster
prio1
toDo
AbgabeTermin
/LF21/ - ADLBackEnd - ADL BackEnd per Basic Authentication schützen
#7 opened by Svvas
prio2
toDo
AbgabeTermin
/LF31/ - ADLFrontEnd - Auto-Daten je nach Zeitspanne anzeigen
#9 opened by Svvas
prio2
toDo
AbgabeTermin
/LF40/ - Account - Kunden registrieren
#11 opened by TazztheMonster
once
prio3
AbgabeTermin
/LF50/ - Account - Login
#10 opened by Svvas
once
prio3
AbgabeTermin
In progress
2
/LF20/ - ADLBackEnd - Auto Daten speichern
#6 opened by TazztheMonster
prio1
toDo
AbgabeTermin
/LF30/ - ADLFrontEnd - Auto-Daten anzeigen
#8 opened by Svvas
prio1
toDo
AbgabeTermin
Done
0
/LF20/ - ADLBackEnd - Auto Daten speichern #6
Opened in TazztheMonster/AbschlussPruefung
TazztheMonster
commented 4 hours ago
Komponent: ADLBackEnd
Anwendungsfall: Auto Daten speichern
Akteur: ADLRecorder, ADLFrontEnd
Beschreibung: Die von der ADL Recorder erfassten Auto Daten sollen in der Cloud gespeichert werden.
Entwickeln Sie einen Rest/HTTP-API Microservice, der die Möglichkeit anbietet, Daten in einer NOSQL Datenbank (Mongo DB, ADLISDB) zu speichern und zu lesen. Beschreiben sie zuerst jedoch die Schnittstelle Ihrer API mit OpenApi/Swagger und achten Sie auf folgendes:
basePath: /adl-api/v1
post: Speichen neuer ADL Datenpaket
operationID: saveADL
responses:
200: Successful
500: Server Error
Sicherheit
adl basicAuth
get: Lesen der ADL Datenpaket
operationID: retrieveADLByFin
responses:
200: Successful
500: Server Error
Sicherheit
adl basicAuth
Diese Komponente soll mit Spring Boot entwickelt werden.
Show less
Assignees
Go to issue for full details
Close issue


: ADLRecorder
:ADLBackend
Read data from RecorderYAML
Add generated values
Send updated data
HTTP response
Update RecorderYAML
:ADLFrontend
:ADLBackend
request data
return data

3.1. **Arbeitsmethodik**

Da die Domänenmodelle klar definiert wurden sind und die einzelnen Microservices in ihrer Grundstruktur eine sehr geringe Komplexität aufweisen, wurde beschlossen die Grundlage für das ADLFrontEnd und das ADLBackEnd in Einzelarbeit umzusetzen und ein Code Review folgen zu lassen. Alles folgende wie z.B. die Implementierung von swagger im ADLBackEnd oder die einrichtung des Kubernetes sollten im Pair entwickelt werden.

3.2. **Zielplattform**

Da bereits vorgegeben war, dass das ADLBackEnd mit Spring Boot zu entwickeln sei und die Containerisierung sowie das Deployment in Kubernetes ebenfalls von Seite des Auftraggebers gesetzt waren, ergab sich die Frage nach der zu verwendenden Programmiersprache sowie dem Framework lediglich für das ADLFrontend und den ADLRecorder. Im Falle des ADLFrontEnds fiel die Wahl auf ein in Typescript mithilfe des Angular Frameworks geschriebenes Frontend, da Marlon damit bereits Erfahrungen sammeln konnte. Dadurch soll eine relativ kurze Entwicklungszeit ohne grosse Komplikationen gewährt werden.
Da im Laufe der Schulung bereits eine dem ADLRecorder in der Funktion sehr ähnliche Komponente vorgestellt wurde, welche vom Trainer in Python geschrieben wurde, fiel der Entschluss darauf, sich an dieser Komponente zu orientieren und den ADLRecorder in Python umzusetzen. Außerdem ist Python eine Sprache die sehr gut für kleine Aufgaben geeignet ist, da sie sich dort oft in wenigen Zeilen code lösen lassen.

3.3. **Tooling**

Für die Kommunikation untereinander und das übertragen des Monitors wurde der vom Auftraggeber bereitgestellt Raum bei BigBlueButton genutzt. Da der Chat dort regelmäßig gelöscht wird, haben wir das senden von Links und Verweisen jedoch über Teams umgesetzt um diese später noch verfügbar zu haben, sollten wir ähnliche Probleme ein weiteres mal erleben oder auch zum Zwecke der Dokumentation.
Der Java code für das ADLBackEnd wurde mit der IDE IntelliJ geschrieben. Diese war uns vertraut und durch die vielen Funktionen und Hilfen die sie anbietet fiel uns die Wahl dementsprechend einfach. Das ADLFrontEnd und der ADLRecorder so wie die Dockerfiles wurden mit dem Tool VisualStudioCode geschrieben. Dieses Tool ist dank

seiner vielen erweiterungsmöglichkeiten ideal für Aufgaben wie das schreiben von YAML files und Javascript/Typescript. Das schreiben der YAML files für Kubernetes haben wir direkt in der uns zur Verfügung gestellten online Umgebung Codespaces gemacht. Dort konnten wir diese direkt anwenden da wir über diese auch direkt auf unseren Kubernetes Cluster zugreifen konnten.

4 **Entwurfsphase**

4.1 **ADLFrontEnd**

Das Frontend soll aus einem Loginfenster bestehen, dieses wiederum soll aus einem Eingabefeld für den Benutzernamen, einem Passwordfeld und einem Loginbutton zusammengesetzt sein. Bei erfolgreichem Login wird das Ausgabefenster angezeigt. Im Ausgabefenster ist ganz oben ein Textfeld zur eingabe der VIN zu finden. Darunter ein Knopf, zum Absenden der Anfrage und ein Auswahlfeld, welche Anzahl von letzten Datensätzen angezeigt werden soll.
Darunter werden dann die vom Backend zurückgegebenen Daten in den korrespondierenden Textfeldern angezeigt. Pro angefragtem Datensatz werden die entsprechende Menge an Textfeldern generiert und untereinander angezeigt.

4.2 **ADLBackEnd**

Für das ADLBackEnd wurden zwei Endpunkte definiert. Ein Post-Endpunkt auf den Pfad adl-api/v1/cars/{vin} um neue Daten für ein Auto abzuspeichern und einen Get-Endpunkt auf den Pfad adl-api/v1/cars/{vin}/{amountOfDatasets} um die letzten 1-X Datensätze für eine jeweilige VIN anzuzeigen.

4.3 **ADLRecorder**

Der ADLRecorder besteht aus einer bereitgestellten .yaml Datei, in der Beispieldaten für ein Auto bereitgestellt werden. Der Recorder liest diese Datei, die sich in seinem Dateiverzeichnis befindet aus und verändert diese Werte randomisiert. Anschließend werden diese Daten auf den Post-Endpunkt des Backends (adl-api/v1/cars/{vin}) gesendet.
Da diese Funktion alle zwei Minuten ausgeführt werden soll, diese Komponente in Kubernetes als Cronjob deklariert und die ausführung wird auf zwei Minuten eingestellt.

So müssen bei einer eventuellen Änderung der Intervallgröße keine Anpassungen im Programmcode erfolgen und ein neues Image gebaut werden sondern lediglich das Cronjobfile muss modifiziert werden.

4.4 **NOSQL-Datenbank**

Zur Umsetzung des Projekts wird eine MongoDB verwendet. Dazu soll ein bereits bestehendes Image heruntergeladen und im Cluster verwendet werden. So ist lediglich noch eine Erstellung eines Users notwendig.

5 **Implementierungsphase**

5.1 **Implementierung ADLFrontEnd**

Zur Umsetzung des ADLFrontends wurde zunächst mit dem befehl npm install -g @angular-cli das Angular Framework installiert. Anschließend wurde per ng new ADLFrontEnd ein neues Angular-Projekt generiert. Es folgte die Installation von Angular-Material mithilfe der Eingabe von ng add @angular/material. Im app.module.ts wurden die Imports gesetzt und sind die Komponenten im gesamten Projekt verfügbar. Es folgte das Anlegen der Domänenmodelle sowie einer carData-Komponente, in der die vom Backend abgefragten Daten angezeigt werden, sowie ein Loginfenster, welches die carData Komponente enthält. Werden die im Quellcode hinterlegten Daten korrekt eingegeben, wird der Loginteil ausgeblendet und die carData-Komponente angezeigt.
In der carData-Komponente wurde neben den Textfeldern zum anzeigen der Daten auch der Post-Request auf den Pfad adl-api/v1/cars/{vin} realisiert, der die Daten aus dem Backend abruft. Dazu wurde der HTTP-Client genutzt.

5.2 **Implementierung ADLBackEnd**

Zur Umsetzung des ADLBackEnds wurde zunächst mit dem Erstellen eines Spring Boot Projektes begonnen. Dies war durch die Integration des Spring Initializers in die IntelliJ IDE mit keinem großen Aufwand verbunden. Anschließend haben wir die Modellierung wie zuvor modelliert eingebaut. Durch die Implementierung von Lombok konnten wir uns das händische Erstellen von gettern und settern sparen. Außerdem konnten wir so

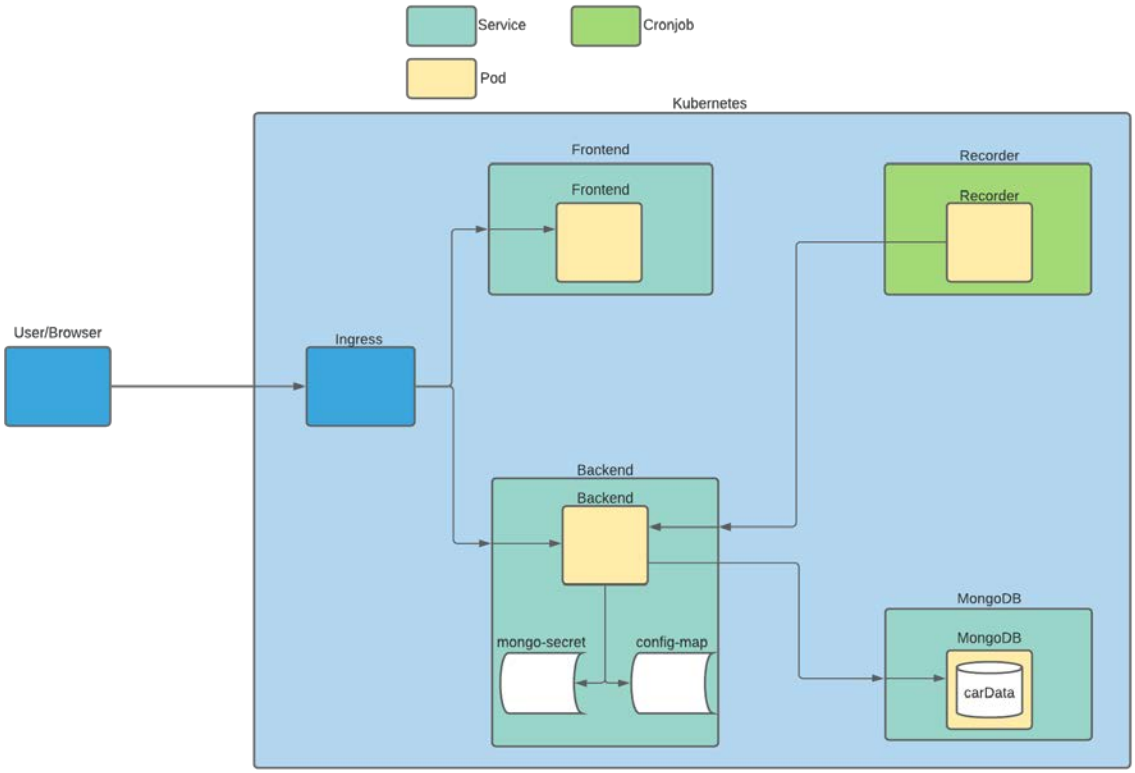
mit sehr wenig Aufwand das Builder-Pattern zum Einsatz bringen welches sich für diese sehr umfangreichen Klassen anbot.
Nachdem alle Datenklassen erstellt wurden konnte das Repository erstellt werden um nachher die Daten mit MongoDB speichern und laden zu können. Dies erwies sich dank Spring und spring.boot.data als sehr einfach. Nachdem dann der RestController mit den entsprechenden Endpunkten geschrieben war, konnte das Ganze auch schon getestet werden. Dabei stellte sich heraus, dass auf dem lokalen System optionale und nicht gesetzte Variablen in den application.properties zu problemen führten weshalb diese zum Testen auskommentiert werden mussten. Auch wenn dies keine dauerhafte lösung ist und auf lange Sicht optimiert werden sollte, war dies für diesen kurzen Prüfungszeitraum eine akzeptable Lösung.
Bevor das ganze auf den main branch auf dem Git repository gepullt werden konnte, haben wir noch einige Tests implementiert die die grundlegenden Funktionen wie das Abrufen und Speichern von Daten aus der Datenbank testen.

5.3 **ADLRecorder**

Aus den in Abschnitt 6.1 ADLRecorder erläuterten Gründen haben wir uns entschlossen den ADLRecorder in NodeJS zu implementieren. Dazu haben wir zuerst ein neues NodeJS Projekt erstellt und unsere Objektstruktur aus dem Frontend übertragen. Da das Frontend in Typescript geschrieben ist war dies mit minimalen Anpassungen leicht möglich. Da wir beide nur grundlegend vertraut mit NodeJS waren, haben wir uns im Anschluss erst nochmal die basics dazu online angelesen. Im Anschluss mit Hilfe von Google und Stackoverflow war es uns möglich unter Verwendung des fs- und js-yaml-Pakets und den dadurch zur verfügung stehenden Methoden die zuvor erstellte yaml-Datei auszulesen. Die so ausgelesenen Daten konnten mit simplen mathematischen Berechnungen verändert werden. Anschließend wurde das axios-Paket genutzt um die modifizierten Daten an das Backend per Post-Request zu senden.

5.4 **Deployment in Kubernetes**

Die aus den zuvor implementierten Applikationen wurden durch Dockerfiles Dockerimages erstellt. Diese wurden auf eine private Dockerregistry gepusht. So lassen ließen sich Deploymentfiles schreiben, in denen die zuvor gepushten Dockerimages als Basis zum Bau eines Pods angegeben wurden. Im Falle des ADLFrontEnds und ADLBackEnds wurden außerdem Services geschrieben, die als Endpunkte der jeweiligen Pods innerhalb des Clusters fungieren. Der ADLRecorder wurde als Cronjob implementiert und braucht somit keinen Service. Weiterer Bestandteil des Deployments ist ein Ingress. Dieser stellt die von Außen erreichbare IPAdresse bereit und leitet den Benutzer auf das Frontend weiter.



6 **Abweichung von der Projektplanung bei der Umsetzung**

6.1 **ADLRecorder**

Während der Entwicklung des ADLRecorders sind wir auf lokale Schwierigkeiten im Zusammenhang mit Python gestoßen. Die lokalen Compiler haben nicht so gearbeitet wie wir das laut den Dokumentationen die wir online fanden erwartet haben. Da wir beide auch keinerlei Vorerfahrung mit Python hatten, haben wir dann spontan entschieden den ADLRecorder mit Node JS zu entwickeln. Diese Sprache hat ähnliche vorzüge wie Python und ist dabei näher an dem, was wir bereits kennen und können (Java, Typescript).
Außerdem war dadurch das wir bereits das Frontend mit Angular entwickelt haben die IDE und das System bereits für den Umgang mit Node JS eingerichtet.

6.2 **ADLBackend**

Durch ein Missverständnis bei der Auftragstellung unsererseits waren wir in der Annahme, die Swagger-ui müsste öffentlich verfügbar sein. Dies hat sich zwar später als falsch erwiesen uns aber sehr viel Zeit gekostet, da wir die root domain auf das Frontend weiterleiten und die Swagger-ui sich ebenfalls auf der root domain befand. Um dieses Problem zu beheben haben unter anderem versucht den Pfad auf den die Swagger-ui reagiert zu ändern, den Ingress so zu konfigurieren, dass er den Pfad den er zum ADLBackend weiterleitet kürzt/ändert und den context path von spring zu ändern. Alle diese Änderungen haben dazu geführt, dass die readieness probe beim Ingress fehl schlug.

7.0 **Lessons …**

7.1 **… implemented**

Bei der letzten Zwischenprüfung sind wir unter anderem darüber gefallen, dass zu Beginn unsere Modellierung nicht ausreichend miteinander abgestimmt war. Dies hat im späteren Verlauf durch kleine Abweichungen zu Problemen und langen Debug- und Testzeiten geführt. Da wir dieses mal alles vorher genauestens zusammen beschrieben haben, sind solche Fehler nicht wieder aufgetreten.
Außerdem haben wir unsere Tests erst sehr spät implementiert was oft Fehler bereits beim Bauen der Anwendung angezeigt hätte. Dieses mal haben wir von Anfang an Tests für unsere essentiellen Funktionen geschrieben.

7.2 **… learned**

Bei der Vorbereitung nicht zu schnell machen!
Auch wenn wir durch unsere Zwischenprüfung den Teil der Planung dieses mal wesentlich umfangreicher ausgeführt haben, hätten wir uns dabei noch mehr Zeit nehmen sollen. Durch Unaufmerksamkeit ist uns dieses mal ein Fehler unterlaufen der so vermeidbar gewesen wäre. Und zwar haben wir in unserem Beispiel die Mileage Klasse zwar modelliert, jedoch nicht dem CarData Modell hinzugefügt weshalb sie ungenutzt war. Da wir uns bei der Implementierung anschließend natürlich an unser Modell gehalten haben war es dort dann natürlich genauso.
Git noch aktiver nutzen!
Auch wenn wir bereits sehr viel mit Git und seinen Funktionen gearbeitet haben, hätte es uns einige male viel Arbeit erspart wenn wir öfter committed hätten und so zum vorherigen Stand wechseln können.
“Have you tried turning it off and on again”
Nicht nur bei lokalen Geräten und Computern hilft manchmal ein Neustart. Auch bei diversenen Problemen mit Kubernetes kann dies helfen.
Für den MVP genau an die Vorgaben halten!
Für die erstellung des ersten MVP wäre es besser gewesen, hätten wir uns genauer an die Vorgaben gehalten. Da wir das dieses mal nicht getan haben und bereits kleine Optimierungen implementieren wollten haben wir uns auf lange Sicht viele Probleme verursacht durch Abhängigkeiten die wir zu Anfang noch nicht bedacht haben.

8.0 **Ausblick**

8.1 **OAuth**

Aus zeitlichen Gründen ist die Authentifizierung mittels OAuth-Verfahren leider nicht mehr implementiert wurden. Dies würde den Login und damit seine dahinter liegenden Services noch besser nach außen schützen und gleichzeitig einen höheren Komfort mit sich bringen da über dieses Verfahren verschiedenste Implementierungen wie z.B. ein einloggen mit VW Ausweis möglich wären (sollte das System im Intranet laufen). Bei diesem Verfahren bekommt der User beim request auf die Seite anstelle unseres Logins das eines OAuth providers zu sehen. Nachdem dieser seinen Login verifiziert hat bekommen wir von diesem den entsprechenden Authorisierungstoken und können so den User identifizieren. Da bei diesem Verfahren dem OAuth provider vertraut werden muss ist es eventuell sinnvoll diesen Service selber zu betreiben. Der Umfang und die genaue Implementierung müsste aber zu dem entsprechenden Zeitpunkt erfolgen.

8.2 **Refactoring**

Aufgrund von zeitlichen Engpässen ist die Codequalität gegen Ende der Entwicklungszeit deutlich gesunken. Dies ist auch diversen Problemen mit Ingress zuzuschreiben weshalb wir sehr häufig nicht wussten ob das Problem bei unserem Code oder beim Ingress liegt. Daher sollte eine refactoring Phase das nächste sein was erfolgt, bevor weitere Features implementiert werden.