

Plattform für Verkehrsinformationen

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2017

Autoren: Oliviero Chiodo, Matteo Patisso
Projektpartner BERNMOBIL
Christian Loosli, Thomas Bodemann
Betreuer: Prof. Dr. Luc Bläser
Experte: Ivan Bütler
Gegenleser: Dipl. Inf.-Ing. ETH Jean-Daniel Merkli

Aufgabenstellung Bachelorarbeit: Plattform für Verkehrsinformationen bei BERNMOBIL

Auftraggeber und Betreuer

Diese Studienarbeit findet in Zusammenarbeit mit BERNMOBIL, der städtischen Verkehrsbetriebe Bern statt.

Ansprechpartner Auftraggeber:

- Thomas Bodenmann, BERNMOBIL, Leiter Informatik, thomas.bodenmann@bernmobil.ch
- Christian Loosli, BERNMOBIL, Software- und Systemingenieur, christian.loosli@bernmobil.ch

Betreuer HSR:

- Prof. Dr. Luc Bläser, Institut für vernetzte Systeme, lblaeser@hsr.ch

Ausgangslage

BERNMOBIL sind die städtischen Verkehrsbetriebe Bern, welche das Bus- und Tramnetz der Stadt Bern inklusive Agglomeration betreiben. Sie verfügen über eine Datendrehscheibe zwecks Verarbeitung von Echtzeitdaten (gemäss VDV 453/454) von verschiedenen Transportunternehmen. Der Datendrehscheibe nachgelagert ist eine so genannte „Webplattform“, welche diese Daten mit weiteren Informationen (zum Beispiel Störungsmeldungen) anreichert, endverbrauchergerecht aufbereitet und darstellt.

Die aktuelle Implementation der Webplattform hat derzeit Einschränkungen hinsichtlich Web- /Cloud-Fähigkeit, Flexibilität und funktionalen Möglichkeiten. Daher soll eine neue Software- Lösung basierend auf einer neuen Architektur realisiert werden, die es erlaubt, möglichst generisch verschiedene Module von individualisierbaren Reiseinformationen/Berechnungen Kunden über das Web zur Verfügung zu stellen. Besondere Herausforderungen sind hierbei die flexible und harmonisierte Integration von den verschiedenen Datenquellen von Fahrgastinformationen mit verschiedenen Formaten und Protokollen, die Modularisierung der Architektur sowie individualisierbare Repräsentation als Web-Frontend.

Ziele und Aufgabestellung

Das Ziel dieser Bachelorarbeit ist es, einen Prototyp einer neuen Plattform für Verkehrsinformationen für BERNMOBIL zu realisieren, welche für die Anforderungen des Auftraggebers gerüstet ist.

Folgende Fragestellungen sind dabei zu untersuchen:

- Flexible und harmonisierte Integration von Fahrgastinformationen aus diversen Quellen und mit verschiedenen Formaten/Protokollen, z.B. Open Data Plattform von SBB, GTFS etc. mit Soll- und Ist-Daten, Konzept der Metabahnhöfe etc.
- Erweiterbarkeit für neue web-fähige Module mit verschiedenen Inputs, Outputs, Verarbeitungen von Fahrgastinformationen, Reiserouten etc.
- Web- und Cloud-Fähigkeit der Software-Architektur mit entsprechender Skalierbarkeit/Elastizität.

Technologisch sollen dafür primär Java mit geeigneten weiteren Frameworks, Technologien, Sprachen und Tools eingesetzt werden. Das Vorgehen ist mit dem Auftraggeber abzusprechen.

Die Arbeit hat folgende spezifische Ziele:

- Aufnahme der Anforderungen, Analyse der Domäne, inkl. Datenformate/Protokolle, Analyse der Schwächen der aktuellen Lösung
- Entwurf einer Software-Architektur für die Datenintegration, Modularisierung sowie Web- und Cloud-Fähigkeit

- Implementation mindestens eines Referenz-Moduls, z.B. zur Live-Darstellung von Abfahrtsverbindungen als Web-Frontend, die vom Kunden individualisiert werden kann.
- Tests und Dokumentation der Software-Architektur und des implementierten Moduls.
- Schlusspräsentation der Resultate beim Auftraggeber

Diese Arbeit soll nach den modernsten Software Engineering Methoden umgesetzt werden. Insbesondere ist auf agiles Projektvorgehen, umfassendes Testing und Continuous Integration zu achten.

Zur Durchführung

Mit dem HSR-Betreuer finden wöchentliche Besprechungen statt. Zusätzliche Besprechungen sind nach Bedarf durch die Studierenden zu veranlassen. Besprechungen mit dem Auftraggeber werden nach Bedarf durchgeführt.

Alle Besprechungen sind von den Studenten mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren, das dem Betreuer und dem Auftraggeber per E-Mail zu gestellt wird.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsergebnisse in vorläufigen Versionen abzugeben. Über die abgegebenen Arbeitsergebnisse erhalten die Studierenden ein vorläufiges Feedback. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen¹. Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren. Die Dokumentation ist vollständig auf CD/DVD/USB-Stick in 3 Exemplaren abzugeben. Auf Wunsch ist für den Auftraggeber und den Betreuer eine gedruckte Version zu erstellen.

Termine

Siehe auch Terminplan auf Skripteserver Informatik > Fachbereich > Bachelor-Arbeit_Informatik:

20.2.17 Beginn der Bachelorarbeit, Ausgabe der Aufgabenstellung durch die Betreuer.

9.6.17 Die Studierenden geben den Abstract für die Diplomarbeitsbroschüre zur Kontrolle an ihren Betreuer/Examinator frei. Die Studierenden erhalten vorgängig vom Studiengangsekretariat die Aufforderung und die Zugangsdaten zur Online-Erfassung des Abstracts im DAB-Tool. Die Studierenden senden per Email das A0-Poster zur Prüfung an ihren Examinator/Betreuer. Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen unter den allgemeinen Infos Diplom-, Bachelor- und Studienarbeiten zur Verfügung.

14.6.17 Der Betreuer/Examinator gibt das Dokument mit dem korrekten und vollständigen Abstract für die Broschüre zur Weiterverarbeitung an das Studiengangsekretariat frei. Fertigstellung des A0-Posters bis 10.00 Uhr an das Studiengangsekretariat.

16.6.17 Abgabe des Berichtes an den Betreuer bis 12.00 Uhr.

16.6.17 Präsentation und Ausstellung der Bachelorarbeiten, 16 bis 20 Uhr

¹siehe <https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html?&L=0>

Beurteilung

Eine erfolgreiche Bachelorarbeit zählt 12 ECTS-Punkte pro Studierenden. Für 1 ECTS Punkt ist eine Arbeitsleistung von ca. 25 bis 30 Stunden budgetiert.

Für die Beurteilung sind die HSR-Betreuer verantwortlich.

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/6
2. Berichte (Abstract, Mgmt Summary, technischer u. persönliche Berichte) sowie Gliederung, Darstellung, Sprache der gesamten Dokumentation	1/6
3. Inhalt *	3/6
4. Mündliche Prüfung zur Bachelorarbeit	1/6

*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit mit den Studierenden festgelegt.

Im Übrigen gelten die Bestimmungen der Abt. Informatik zur Durchführung von Bachelorarbeiten.

Rapperswil, den 10. Februar 2017
Der verantwortliche Dozent

Prof. Dr. Luc Bläser
Institut für vernetzte Systeme

Hochschule für Technik Rapperswil

Abstract

Ausgangslage

BERNMOBIL betreibt eine Webplattform, auf welcher ein Fahrgast Verbindungen, Abfahrten, Ankünfte und weitere Informationen über das öffentliche Verkehrsnetz der ganzen Schweiz sowie vom nahen Ausland anzeigen kann. Das aktuelle System bei BERNMOBIL ist in vielen Punkten limitiert. Es ist weder modular noch cloudfähig, dementsprechend werden auch Alternativen evaluiert. Ziel dieser Arbeit ist es, eine von Grund auf neue, moderne Softwarelösung für Verkehrsinformationen zu realisieren.

Vorgehen/Technologien

Der entwickelte Prototyp bindet die statischen und Echtzeit-Daten aus den Fahrplänen des Auftraggebers an. Diese werden im General Transport Feed Specification (GTFS) Format zur Verfügung gestellt. Als Technologie wird das Spring Framework mit Spring Boot, Spring Data und Spring Batch eingesetzt. Für die Anwendung ist eine cloudfähige und ein skalierbarer Architektur-Prototyp konzipiert worden. Besonderes Gewicht wurde auf die Möglichkeit der Anbindung von weiteren Datenquellen und Standards gelegt. Die gesamte Anwendung ist in der Amazon-Cloud mit EC2, RDS und Elastic Beanstalk getestet worden.

Ergebnis

Der Prototyp besteht aus 3 Komponenten. Zwei davon importieren Daten aus dem GTFS-Format und konvertieren diese in das entworfene Schema. Dieses Schema ist für eine schnelle Anzeige konzipiert worden und fasst alle für einen Abfahrtsanzeiger relevanten Daten zusammen. Es ist besonders darauf geachtet worden, dass die Anwendung in Zukunft auch weitere weitere Formate wie HAFAS oder VDV-453/4 unterstützen kann. Die erste Import-Komponente verarbeitet statische Fahrplan Daten, welche für längere Zeiträume gültig sind. Die zweite Import-Komponente verarbeitet Echtzeit-Fahrplan Daten, welche sehr oft aktualisiert werden und deren Gültigkeit auf einen kurzen Zeitraum beschränkt ist. Die dritte und letzte Komponente realisiert die Anzeige der Daten. Der Prototyp zeigt, dass die modulare Struktur der Komponenten sowie der leichtgewichtige Aufbau der Anzeigekomponente eine einfache Skalierung ermöglicht. Mit der Verwendung eines Load Balancers und einer Datenbank kann der Prototyp ohne Weiteres einige 100 Anfragen pro Sekunde verarbeiten.

Management Summary

Ausgangslage

BERNMOBIL betreibt eine Webplattform, auf welcher ein Fahrgast Verbindungen, Abfahrten, Ankünfte und viele weitere Informationen über das öffentliche Verkehrsnetz der ganzen Schweiz und sogar darüber hinaus anzeigen kann. Die aktuelle Lösung ist in einigen entscheidenden Fällen limitiert. Sie kann beispielsweise schlecht in eine modulare Struktur einer Cloud Anwendungen integriert werden. In dieser Arbeit wurden die Grundbausteine für eine neue Lösung der Webplattform gelegt. Dabei wurde viel Wert auf dessen Modularisierung und Cloudfähigkeit gelegt.

Vorgehen/Technologien

Durch die sehr offene Aufgabenstellung hing der Erfolg des Projektes von einer guten Evaluation der verwendeten Produkte und einem sauberen Design der Architektur ab. Um eine flexible Weiterentwicklung zu ermöglichen, ist die Aufteilung der Anwendung in verschiedene Module, aber auch die Cloud-Fähigkeit des Produktes ein zentraler Punkt. Durch die Modularisierung soll sichergestellt werden, dass die Anwendung in einer Cloud betrieben werden und flexibel skalieren kann. Die grössten Herausforderungen bestanden einerseits darin, einen guten Mittelweg der Flexibilität zu finden, und andererseits, Flaschenhalse des gesamten Prozesses zu eliminieren. So sollte schliesslich ein skalierbares und zukunftsträchtig System gebaut werden.

Ergebnisse

Der entwickelte Prototyp beinhaltet drei Module zum Import und zur Anzeige. Die Skalierbarkeit ist mehr als zufriedenstellend und bietet eine saubere Basis für zukünftige Module. Alle Module bieten eine sehr gute Performance und sind ideal für die Anwendung in einer Cloud. Der gesamte Prototyp ist in der Amazon Cloud getestet worden.

Inhaltsverzeichnis

I	Einleitung	3
1	Ausgangslage	4
1.1	General Transit Feed Specification	4
1.1.1	GTFS Static	4
1.1.2	GTFS Realtime	6
1.2	Datenquellen	7
1.3	Vorgaben	8
1.4	Anforderungen	8
1.4.1	Funktionale Anforderungen	8
1.4.2	Nicht-funktionale Anforderungen	9
II	Analyse	10
2	Evaluation	11
2.1	Web Framework	11
2.1.1	Einfachheit	11
2.1.2	Skalierbarkeit	13
2.1.3	Performance	16
2.1.4	Cloud-Awareness	19
2.1.5	Ergebnis	21
2.2	Persistenz Framework	21
2.2.1	Fazit	22
2.3	Datenimport	22
2.3.1	Batch Import	22
2.3.2	Realtime Import	23
2.4	Kommunikation	23
2.4.1	Architektur	23
2.4.2	Frameworks	24
3	Architektur	29
3.1	Systemarchitektur	29
3.1.1	Organisation	30
3.1.2	Datenquellen	30
3.1.3	Fahrplan Datenbank	31
3.1.4	Mapping Datenbank	31
3.1.5	Importkomponente	31
3.1.6	Anzeigekomponente	32
3.1.7	Skalierbarkeit	32
3.1.8	Cloudfähigkeit	32
3.1.9	Fazit	33
3.2	Datenmodell	33

III Umsetzung	36
4 Implementation	37
4.1 Infrastruktur	37
4.1.1 Amazon AWS	37
4.2 Prototyp	38
4.2.1 Fahrplandatenbank	38
4.2.2 Mapping Datenbank	40
4.2.3 Static Import	41
4.2.4 Realtime Import	44
4.2.5 Anzeige Modul	45
4.2.6 Ausfallszenarien	48
4.3 Versionierung der Fahrplandaten	49
4.3.1 Static-Update	49
4.3.2 Realtime-Update	50
4.4 Amazon Web Services	51
4.4.1 Anzeigemodul	51
4.4.2 Datenbanken	52
4.4.3 Import Module	52
4.4.4 Deployment Architektur	53
5 Testing	55
5.1 Cloud	55
5.1.1 Ergebnis	55
5.2 Lasttests	55
5.2.1 Ergebnis	56
IV Ergebnisse	58
5.3 Funktionalität	59
5.4 Erweiterbarkeit	59
5.5 Skalierbarkeit	59
5.6 Ausblick	60
5.6.1 Anzeigemodul	60
5.6.2 Cloud-Environment	60
5.6.3 Erweiterung	60
Anhang	66

Teil I

Einleitung

Kapitel 1

Ausgangslage

Als städtisches Verkehrsunternehmen, wünscht sich BERNMOBIL eine Plattform, welche die Verarbeitung von Fahrplandaten vereinheitlicht und vereinfacht. Um eine Basis für solch eine Plattform zu schaffen, ist ein Prototyp entwickelt worden, welcher auf Grundlage von GTFS Transitdaten, Abfahrten ab verschiedenen Haltestellen anzeigen kann. In dieser Arbeit wird die Evaluation der verwendeten Frameworks, die Architektur und die entwickelte Implementation beschrieben.

1.1 General Transit Feed Specification

GTFS ist eine Schnittstellendefinition von Google. Sie ist Teil der Google Transit API und beinhaltet Definitionen für Daten des öffentlichen Verkehrs. GTFS definiert wie Fahrpläne, Routen, Abfahrtszeiten und vieles mehr strukturiert und veröffentlicht werden. Der Standard erlaubt es Verkehrsbetrieben, Verkehrsdaten zu veröffentlichen, sodass Drittanbieter diese Daten in Services einbinden können. Die Google Transit API ist öffentlich und wird von vielen bekannten Agenturen, wie auch BERNMOBIL, verwendet. Es gibt zwei Unterkategorien: „GTFS Static“ und „GTFS Realtime“.

1.1.1 GTFS Static

Der GTFS Static Standard definiert wie die statischen ÖV Daten strukturiert sind. Ein Beispiel für statische Daten ist der Fahrplan der SBB, welcher alle geplanten Abfahrtszeiten beinhaltet und wöchentlich aktualisiert wird.

Der auf GTFS Static abgebildete Fahrplan wird mithilfe eines sogenannten GTFS Static Feed über eine API zur Verfügung gestellt. Ein GTFS Static Feed besteht aus Textdateien mit vordefinierten Namen. Die Namenskonvention, sowie der interne Aufbau der Textdatei muss eingehalten werden, damit der Feed gültig ist. Bei der Überprüfung des GTFS Feeds werden auch weitere Faktoren wie Felder oder Beziehungen berücksichtigt.

Jede Datei des Feeds hat einen vordefinierten Aufbau. Die erste Zeile bildet jeweils den Header, alle darauffolgende Zeilen bilden je einen Datensatz. Die Felder innerhalb einer Zeile werden durch ein Komma getrennt, wie man es von einer klassischen CSV Datei kennt. Es sind jedoch zusätzlich optionale Spalten erlaubt.

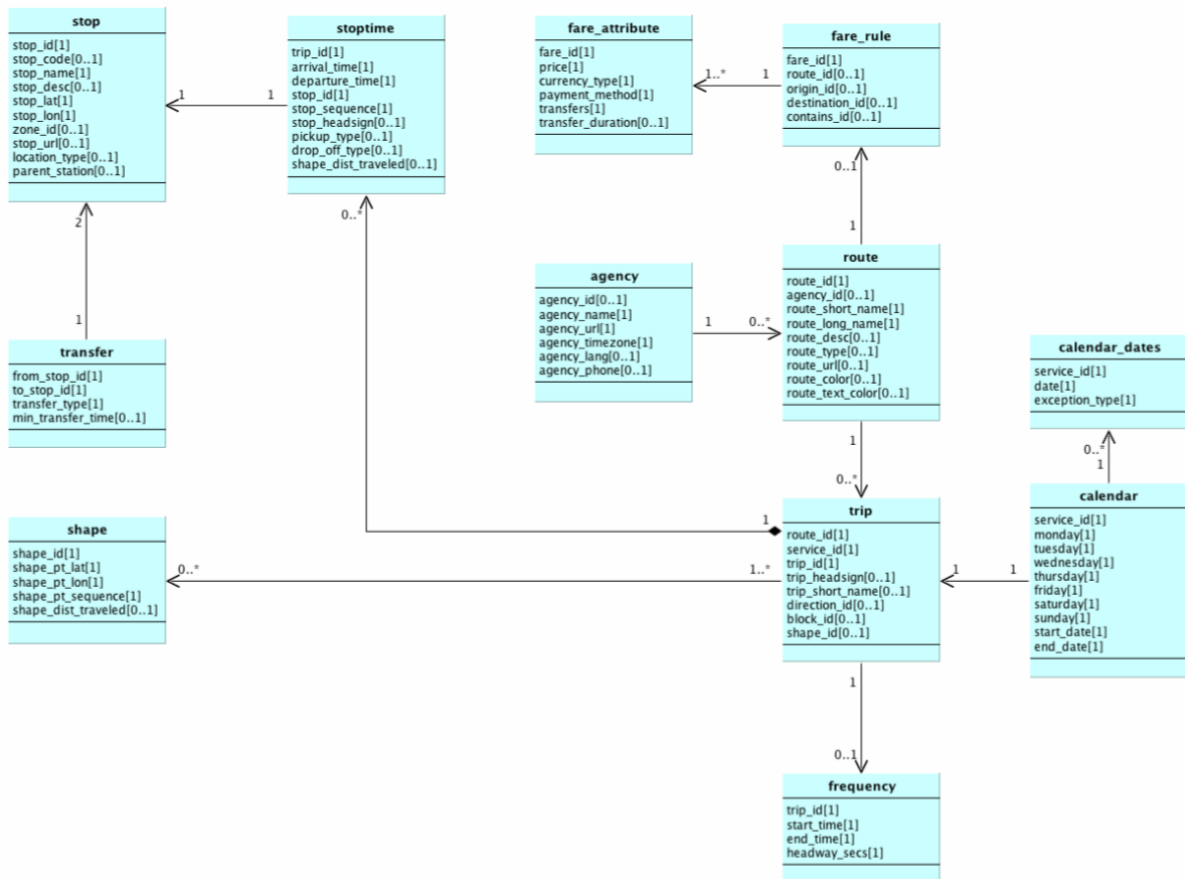


Abbildung 1.1: Übersicht GTFS Static Datenmodell

Abbildung 1.1 zeigt das Datenschema, wie es im GTFS Standard definiert wird. Die einzelnen Dateien beinhalten Informationen zum Durchsuchen und Anzeigen der Verkehrsinformationen. In Tabelle 1.1 sind die Bedeutungen der einzelnen Dateien erläutert [7].

Dateiname	Beschreibung
agency.txt	Ein oder mehrere Verkehrsbetriebe, welche Daten in diesem Feed bereitstellen.
stops.txt	Individuelle Standorte, an denen Verkehrsmittel Fahrgäste abladen und abholen.
routes.txt	Verkehrsrouten. Eine Verkehrsrouten ist eine Gruppe von Fahrten (Trips), welche den Fahrgästen als einzelner Service dargestellt wird.
trips.txt	Die Fahrten für jede Route. Eine Fahrt ist eine Sequenz von einem oder mehreren Stopps zu einer bestimmten Zeit
stop_times.txt	Zeiten, zu denen ein Verkehrsmittel an einem individuellen Stopp ankommt und abfährt.
calendar.txt	Daten für einen Service in einem wöchentlichen Fahrplan. Definiert wann ein Service startet und endet, sowie die Wochentage an denen ein Service verfügbar ist.
calendar_dates.txt	Ausnahmen für einen Service, welcher in calendar.txt definiert ist. Wenn diese Datei alle Daten der Services beinhaltet, kann diese die Datei calendar.txt ersetzen.
fare_attributes.txt	Informationen zum Beförderungsentgelt eines Verkehrsbetriebs.
fare_rules.txt	Regeln für die Anwendung der Tarifinformationen für die Strecken eines Verkehrsbetriebes.
shapes.txt	Regeln der dargestellten Linien auf einer Karte, welche den Routen des Verkehrsbetriebes entsprechen.
frequencies.txt	Zeit zwischen Fahrten für Strecken welche mit variabler Häufigkeit bedient werden.
transfers.txt	Regeln um Verbindungen an Transferpunkten zwischen Routen herstellen zu können.
feed_info.txt	Zusätzliche Informationen über den Feed. Einschliesslich Herausgeber, Version und Informationen zum Ablauf der Daten.

Tabelle 1.1: GTFS Static Spezifikation

1.1.2 GTFS Realtime

GTFS Realtime ergänzt GTFS Static mit Echtzeitdaten. BERNMOBIL aktualisiert die Echtzeitdaten im Intervall von 30 Sekunden. Echtzeitdaten im öffentlichen Verkehr sind in vielen Bereichen nützlich und sind in drei Kategorien unterteilt:

Trip Updates Die Trip Updates beinhalten alle Informationen, welche die Daten aus dem statischen Fahrplan betreffen. Dabei kann es sich um eine einfache Verspätung oder eine kompliziertere Meldung wie einen Ausfall oder eine Umleitung handeln.

Beispiel: *„Die S15 in Richtung Rapperswil trifft in Rüti ZH um 16:15 anstatt 16:09 ein.“*

Service Alerts Ein Service Alert beinhaltet Informationen, welche nicht individuelle Routen, sondern einen grösseren Teil des Netzwerks betreffen. Ein Alert beinhaltet immer eine Zeitspanne, in welcher fahrgäste mit einer Meldung über einen Vorfall informiert werden.

Beispiel: *„Aufgrund von Bauarbeiten kann heute die Haltestelle Rüti ZH nicht bedient werden.“*

Vehicle Positions Die Vehicle Positions Informationen werden oft versendet. Idealerweise geschieht dies automatisch durch ein GPS-fähiges Gerät oder etwas Ähnlichem. In diesen Informationen ist der aktuelle Standort eines Fahrzeuges zu einem definierten Zeitpunkt zu finden. Es können auch weitere Informationen wie Verkehrsauslastung oder Fahrzeugbelegung mitgeliefert werden.

Beispiel: *„Abfahrt S15 in Richtung Rapperswil ab Haltestelle Rüti ZH um 16:09:10.“*

Die Daten von GTFS Realtime haben, wie die von GTFS Static, vordefinierte Felder mit Namenskonvention und unterstützen ebenfalls optionale Felder. Die Informationen werden jedoch nicht in einer Sammlung von CSV-, sondern im Protocol Buffers Format ausgeliefert.

1.2 Datenquellen

Die Datenquellen wurden in dieser Arbeit durch BERNMOBIL zur Verfügung gestellt. Die Testdaten sind von Google abgenommen worden, und bilden somit gültige GTFS Feeds ab. Es wurden sowohl Static- wie auch Realtime-Daten durch eine HTTP API zur Verfügung gestellt.

1.3 Vorgaben

Der Prototyp baut auf keiner bestehenden Struktur auf und hat nur wenige Einschränkungen bezüglich Technologien oder Lösungsansätze.

Java 8

Der Auftraggeber BERNMOBIL wünscht sich als Programmiersprache Java 8. Es gibt keine Wünsche bezüglich Frameworks oder Libraries.

Modularisierung

BERNMOBIL wünscht sich den Einsatz von Docker für die Modularisierung mit Hilfe von Containern. Es darf eine alternative Technologie verwendet werden, insofern diese klaren Vorteile gegenüber Docker aufweisen kann.

Deployment

Aufgrund bereits bestehender Infrastruktur wird als Cloudanbieter Amazon bevorzugt. Amazon eignet sich einerseits wegen des breiten Angebotssortiments und andererseits aufgrund der kostenfreien Testumgebung sehr gut als Anbieter.

GTFS

GTFS ist ein Datenformat von Google welches sich bei Anwendungen im Bereich des öffentlichen Verkehrs etabliert hat. Der aus dieser Arbeit resultierende Prototyp soll mit einer GTFS-Datenquelle umgehen können. Ein Wechsel des Dateiformats zu einem späteren Zeitpunkt soll ebenfalls vereinfacht implementiert werden können.

1.4 Anforderungen

1.4.1 Funktionale Anforderungen

Der Prototyp ist ein Abbild eines Abfahrtsanzeigers mit limitierter Funktionalität, um einen Durchstich durch alle Schichten zu erreichen. Um den Scope des Prototyps abzugrenzen sind User Stories definiert worden.

User Story 1.1 Ich als Student will die nächste Abfahrt ab dem Bahnhof Rapperswil sehen, weil ich wissen möchte, wann ich als nächstes nach Hause kann.

Erfüllungskriterium: Ich sehe die nächste Abfahrt in Richtung Zürich und weiss, wann ich bei der HSR loslaufen muss.

User Story 1.2 Ich als Dozent will die nächsten Abfahrten ab Rapperswil sehen, weil ich wissen möchte welche Verbindungen mir für den Heimweg zur Verfügung stehen.

Erfüllungskriterium: Ich sehe die nächsten Abfahrten in Richtung Zürich und weiss, wann ich bei der HSR loslaufen muss

User Story 2 Ich als Fahrgast möchte die Verspätung der nächsten Abfahrten ab Rapperswil sehen, um meine tatsächliche Ankunftszeit bestimmen zu können.

Erfüllungskriterium: Ich sehe allfällige Verspätungen bei den Abfahrten ab Rapperswil und weiss, um wieviel sich meine Reise verzögern wird.

User Story 3 Ich als Mitarbeiter bei BERNMOBIL möchte den Status der Linie 10 in Richtung Bahnhof sehen, weil ich wissen möchte, ob die Linie unterbrochen ist.

Erfüllungskriterium: Ich sehe, ob die Linie 10 in Richtung Bahnhof unterbrochen ist.

User Story 4.1 Ich als Betreiber dieser Plattform will diese sowohl in einer Cloud (z.B. Amazon EC/AWS, MS Azure) wie auch On-Premise installieren können, damit ich diese entsprechend meiner IT Infrastruktur / Ressourcen betreiben kann.

Erfüllungskriterium: Die Applikation unterstützt mehrere Deployment-Szenarien und lässt einen Betrieb in möglichst vielen Cloud-Umgebungen zu und kann auch On-Premise gehostet werden.

User Story 4.2 Ich als Betreiber dieser Plattform will die einzelnen Module (z.B. Input für unterschiedliche Transportunternehmen, Output für mehrere Anzeiger) als mehrere Instanzen betreiben können, damit die Lösung skalierbar ist.

Erfüllungskriterium: Die Architektur des Prototyps ermöglicht das dynamische Einsetzen und Entfernen von Modulen und bietet Schnittstellen für weitere Module an.

1.4.2 Nicht-funktionale Anforderungen

Skalierung/Elastizität

Die Last, welche ein Abfahrtsanzeiger tragen muss, ist sehr konstant. Die Anzahl Anfragen kann gut kontrolliert und an die verfügbaren Leistungen angepasst werden. Trotzdem soll die Anwendung so designed werden, dass sie möglichst reibungslos horizontal sowie vertikal skalieren kann. BERNMOBIL betreibt ca. 350 Haltestellen auf ihrem gesamten Netz. Davon ausgegangen, dass bei vielen Haltestellen zwei Anzeigen stehen, an grösseren Bahnhofsanlagen mehr, werden für das gesamte Netz etwa 900 Anzeiger notwendig sein. Aufgrund dieser Anzahl sollte ein System mit mindestens 1000 Monitoren stabil funktionieren. Da BERNMOBIL einer der grösseren Verkehrsbetriebe der Schweiz ist, ist anzunehmen, dass kleinere Betriebe lediglich einen Drittel an Haltestellen betreiben müssen. Eine einzelne Instanz der Anzeige sollte mindestens 300 Monitore mit Daten beliefern können. Diese Anforderung wird mittels Lasttest und den Skalierungsangeboten von Amazon validiert und verifiziert.

Cloud-fähig

Als zukunftsichere Anwendung muss es möglich sein, die Anwendung in einer Cloud sowie auch selbst (on premise) hosten zu können. Die Architektur und das Design des Prototyps sollen es zulassen, dass es in jeder beliebigen Cloud, welche die passenden Services anbietet, möglich ist, die Anwendung zu betreiben. Dabei soll sich die Software einfach in die Services verschiedener Cloudanbieter integrieren lassen, so dass für einen Host der Software möglichst wenig Aufwand entsteht. Diese Anforderung wird mit den Angeboten von Amazon validiert und verifiziert.

Erweiterbar/Modular

Der Prototyp soll ein modulares Zusammenspiel der Komponenten ermöglichen. Mit der Modularität soll erreicht werden, dass flexibel verschiedene Datenquellen und unterschiedliche Anzeigekomponenten verwendet werden können. Die Verwendung von GTFS für einen Prototyp ist ideal, da dieser Standard weniger komplex ist als die VDV Normen. Jedoch soll es in Zukunft möglich sein, weitere Standards und Datenquellen an die Anwendung anbinden zu können. Deshalb ist es besonders wichtig, dass die Komponenten einfach zu erweitern sind und die Architektur mit vielen Quellen und Standards umgehen kann. Besonders wichtig ist es, dass das System auch bei vielen Komponenten noch skalieren kann. Die Erfüllung dieser Anforderung wird in der Architekturphase ausgearbeitet und durch Beschreiben der Konzepte überprüft. Eine Verifizierung oder Validierung ist für diese Arbeit nicht vorgesehen.

Performance

Eine mögliche Schwachstelle von verteilten Systemen ist die Performance. Bei der Verteilung von Komponenten muss auf Engpässe geachtet werden. Die Verarbeitungszeit einer Anfrage muss bei jeder Komponente so klein wie möglich gehalten werden. Durch Analyse von Konkurrenzprodukten wurde eine mittlere Antwortzeit von 2 Sekunden ermittelt. Es wird pro Anfrage ebenfalls eine mittlere Antwortzeit von 2 Sekunden angestrebt, sowie eine maximale Antwortzeit von 3 Sekunden toleriert.

Teil II

Analyse

Kapitel 2

Evaluation

Die wichtigsten Komponenten des Prototyps sind Anzeige und Datenimport. Für all diese Komponenten sind Frameworks evaluiert worden, um die Problemdomäne an diese auszulagern. Schlussendlich soll ein Framework oder eine Library dem Anwendungscode so viel wie möglich abnehmen, denn diese sind schnell, sie sind stabil und sie wurden getestet. In diesem Kapitel wird beschrieben, wie die Frameworks evaluiert worden sind und welche Entscheidung aufgrund dieser Evaluation getroffen worden ist.

2.1 Web Framework

Die Entwicklung des Prototyps begann mit dem Front-End des Anzeigemanagers. Mit dieser Komponente interagiert ein Benutzer, weshalb diese zustandslos und leichtgewichtig sein soll. Die gesamte User-Experience hängt von den gewählten Skalierungsstrategien dieser Komponente ab. Für die Anzeige und die Auslieferung der Daten zum Kunden ist ein schneller Webserver nötig.

Um einen Austausch des Frameworks (beispielsweise aus Performance Gründen) während der Arbeit zu vermeiden, mussten diverse Kriterien für die Wahl des Web Frameworks sehr genau recherchiert und evaluiert werden. Folgende Kriterien wurden dabei berücksichtigt:

- Einfachheit: Wie einfach ist es, eine lauffähige Anwendung zu erstellen
- Skalierbarkeit: Wie gross ist das Framework
- Performance: Wie schnell ist das Framework mit wenig Belastung durch Anwendungscode
- Cloud-Awareness: Ist es möglich, cloudfähige Anwendungen mit dem Framework zu bauen

In der Java-Welt gibt es zwei sehr bekannte Web-Frameworks, welche es ermöglichen, bequem und ohne viel Aufwand einen Webserver zur Verfügung zu stellen: Spring Boot mit Spring MVC, Spring HATEOAS etc. oder das Play Framework. Bei beiden Frameworks werden die Klassen, welche Routen definieren, als Controller bezeichnet.

2.1.1 Einfachheit

Dieses Kriterium wurde anhand von „Hello World“ Beispielen evaluiert. Augenmerk wurde besonders auf Klarheit und Lesbarkeit von Methoden und Klassen gelegt, welche die Routen definieren. Zudem wurde darauf geachtet, wie das Framework es erlaubt, komplexe Routen zu definieren und URL- sowie Body-Parameter zu empfangen.

Play Framework

In Play erbt jeder Controller von einer abstrakten Klasse Namens `Controller`. Diese enthält Objekte wie den `Request`, die `Response` und bietet Methoden für die Verarbeitung und Beantwortung eines Requests an. Listing 1 zeigt ein kleines Beispiel eines simplen Play Controllers mit einer Route, welche mit der Methode `render()` HTML an den Aufrufer zurückgibt. Damit die Methode so aufgerufen werden kann, muss die View, also das zu rendernde HTML, in einem bestimmten Ordner nach Konvention abgespeichert sein.

```

public class Application extends Controller {

    public static void index() {
        render();
    }

    public static void sayHello(String myName) {
        render(myName);
    }

    public static Result sayMyName(String name) {
        return ok("Hello" + name);
    }

}

```

Listing 1: Implementation eines Play Controllers

Ein Template HTML Dokument könnte beispielsweise folgendermassen aussehen:

```

#{extends 'main.html' /}
#{set title:'Home' /}

<form action="@{Application.sayHello()}" method="GET">
    <input type="text" name="myName" />
    <input type="submit" value="Say hello!" />
</form>

```

Listing 2: Beispiel eines Scala HTML Templates in Play

Bei der Analyse von 1 fällt auf, dass Play funktional aufgebaut ist. Denn seit Version 2 ist Play komplett in Scala geschrieben. Am sichtbarsten wird dies durch die statischen Methodenaufrufe im Controller, sowie die Definition von statischen Methoden für die Routen.

Spring

In Spring wird für eine kleine Web-Applikation Spring Boot verwendet. Spring selbst kann durch viele verschiedenen Spring Module erweitert werden, wie beispielsweise Spring Data, Spring Web, Spring Security etc. In Spring werden Controller und Routen mit Java Annotations markiert.

```

@Controller
public class GreetingController {

    @RequestMapping("/greeting")
    public String greeting(
        @RequestParam(value="name", required=false, defaultValue="World") String name,
        Model model) {
        model.addAttribute("name", name);
        return "greeting";
    }

}

```

Listing 3: Implementation eines Spring Boot Controller

Ein Template in Thymeleaf, der Template Engine von Spring, könnte so aussehen:

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Getting Started: Serving Web Content</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
  <p th:text="'Hello, ' + ${name} + '!'" />
</body>
</html>

```

Listing 4: Beispiel eines Thymeleaf Template

Fazit

Beide Frameworks erlauben saubere und gut lesbare Klassen in das Framework zu integrieren. Das Play Framework wird durch die Verwendung von Vererbung und statischen Methoden etwas aufgeräumter und lesbarer. Spring Boot bietet aber mehr Flexibilität bezüglich der Verarbeitung von URL- und Body-Parameter. Ausserdem wirkt es auf den ersten Blick intuitiver und einfacher zu konfigurieren als Play. Bei beiden Frameworks ist die minimale Implementation eines Webservers sehr einfach. Auffällig bei der Suche nach diesen Beispielen war, dass Spring sehr gute und akkurate Dokumentationen seiner Produkte bereitstellt, während bei Play Informationen teilweise gar nicht vorhanden oder veraltet waren. Aus diesen Gründen ist Spring für diese Arbeit die bessere Wahl, da die Dokumentation für die Weiterentwicklung des Prototyps ein sehr wichtiger Faktor ist.

2.1.2 Skalierbarkeit

Das Kriterium der Skalierbarkeit ist eines der wichtigsten einer cloudbasierten Anwendung. Die Anzahl Anfragen sollten die benötigte Zeit pro Anfrage linear ansteigen lassen. Unglücklicherweise ist so ein System sehr schwer zu entwickeln. Zur Evaluation dieses Kriterium wurden zwei verschiedene Informationsquellen verwendet. Zuerst wurde in diversen Foren nach Hinweisen, Problemen oder Meinungen gesucht.

Die vorherrschende Meinung vieler ist, dass Play schneller als Spring sei. Für diese These gibt es auch einige Beispiele grosser Firmen. Beispielsweise den URL Shortener, „Hootsuite“, welche ihre Infrastruktur komplett auf Play und Akka umgeschrieben haben [13], oder PayPal, welches ihre gesamte Infrastruktur dank Akka und Netty (das Play Framework verwendet standardmässig Netty als Webserver) auf nur 8 JVMs ausführen können[8].

Was bei dem Durchlesen solcher Posts auffällt, ist, dass diese Anwendungen nicht nur durch Play ihre Performance verbessern konnten. In fast allen gefundenen Fällen wurde die Kommunikation der Komponenten auf ein Actorbasiertes Framework umgestellt, was einen Direktvergleich nicht möglich macht. Die Kommunikation der Komponenten kann auch in Spring auf Akka portiert werden. Spring bietet zudem native Unterstützung für asynchrone Verarbeitungen von Anfragen an [23].

Diese unausgewogene Meinung, welche in vielen Foren vorherrscht sowie die Vergleiche, welche nicht ganz fair sind, machen es unmöglich mithilfe dieser Fakten eine gute Grundlage für die Wahl des Frameworks zu schaffen. Um direkt und unvoreingenommen vergleichen zu können, müssen Tests mit ähnlichen Setups durchgeführt werden. Wichtig bei diesen Tests ist eine Datenbankverbindung oder die Simulation einer solchen. Mithilfe der „Hello World“ Anwendungen aus Unterabschnitt 2.1.1 wurden Anwendungen gebaut, welche Daten aus einer H2 Datenbank lesen, diese verarbeiten, ein Template rendern und dieses an den Browser senden. Für die Datenbankverbindung wurden für beide Frameworks Hibernate bzw. Spring Data verwendet (was lediglich ein Wrapper von Spring um Hibernate ist). Beides sind Setups, welche in produktiven Systemen vorkommen und sich bewährt haben.

Szenario

Die Fähigkeit zu skalieren zeigt sich bei einer Anwendung in einer Veränderung der Reaktionszeit unter verschiedenen Belastungen. Bei einem linearen Anstieg der Anfragen sollte die Antwortzeit der Anwendung maximal linear steigen, sich im Idealfall jedoch gar nicht verändern. Die Tests simulieren

nacheinander einen, zehn und hundert Benutzer, die innerhalb von fünf Minuten konstant Anfragen an die Anwendung senden. Innerhalb dieser Zeit werden die Antwortzeiten gemessen.

Resultate

Für die Auswertung der Resultate werden die Antwortzeiten der drei Phasen miteinander verglichen und versucht, ein Trend abzulesen. Dieser Trend soll eine Prognose auf das Verhalten bei unterschiedlicher Belastung der Anwendung sein.

Play Die Ergebnisse des Lasttests vom Play Framework sind eher ungewöhnlich. Die Antwortzeit aus der Phase mit einem Benutzer ist viel höher als in der zweiten und sogar dritten Phase. Die Antwortzeiten der zweiten Phase sind sehr dicht beieinander, die der dritten Phase wiederum sehr stark verstreut. Diese Werte änderten sich bei mehreren Durchführungen des Tests kaum. Tabelle 2.1 zeigt die Antwortzeiten gemessenen Antwortzeiten in Millisekunden.

Perzentil	1 Benutzer	10 Benutzer	100 Benutzer
min	1635	180	172
25	1635	192	740
50	1635	200	1060
75	1635	210	1412
80	1635	215	1526
85	1635	223	1691
90	1635	230	1814
95	1635	231	1889
99	1635	231	1984
max	1635	232	1968

Tabelle 2.1: Antwortzeiten im Play Framework bei Skalierungstest

Beim Visualisieren dieser Werte können die Unterschiede der Antwortzeiten gut dargestellt werden.

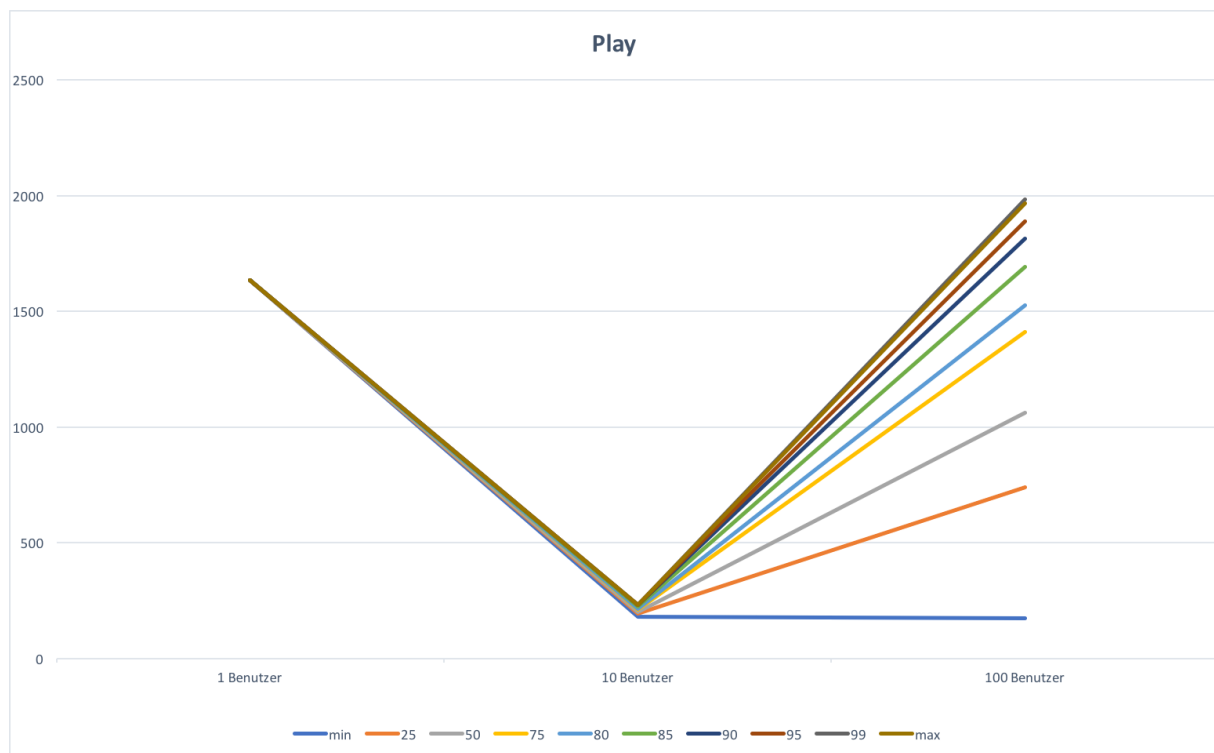


Abbildung 2.1: Diagramm der Antwortzeiten von Play im Skalierungstest

In Abbildung 2.1 fällt auf, dass die Antwortzeiten zwischen der Phase der 10 Benutzern und der Phase der 100 Benutzern ein sehr starker Unterschied entsteht. In der letzten sind die Antwortzeiten fast doppelt so hoch als in der zweiten.

Spring Spring hingegen zeigt ein erwartetes Verhalten. Die Antwortzeiten steigen im Verlauf des Tests stetig an. Die konstant hohe Antwortzeit in der ersten Phase ist auf die Scala Templates, welche Play verwendet, zurückzuführen. Denn diese werden in Play bei einem ersten Request kompiliert und danach in einem Cache abgespeichert, wodurch weitere Anfragen nicht mehr davon beeinflusst werden.

Perzentil	1 Benutzer	10 Benutzer	100 Benutzer
min	13	14	9
25	18	51	190
50	24	92	263
75	30	134	325
80	31	138	340
85	32	148	469
90	33	154	639
95	34	157	669
99	35	166	779
max	36	169	1010

Tabelle 2.2: Antwortzeiten im Spring Framework bei Skalierungstest

Beim Betrachten der Werte erkennt man, dass diese beinahe linear steigen. Lediglich das 95. und das 99. Perzentil sowie die maximal gemessene Antwortzeit zeigen ein exponentielles Wachstum.

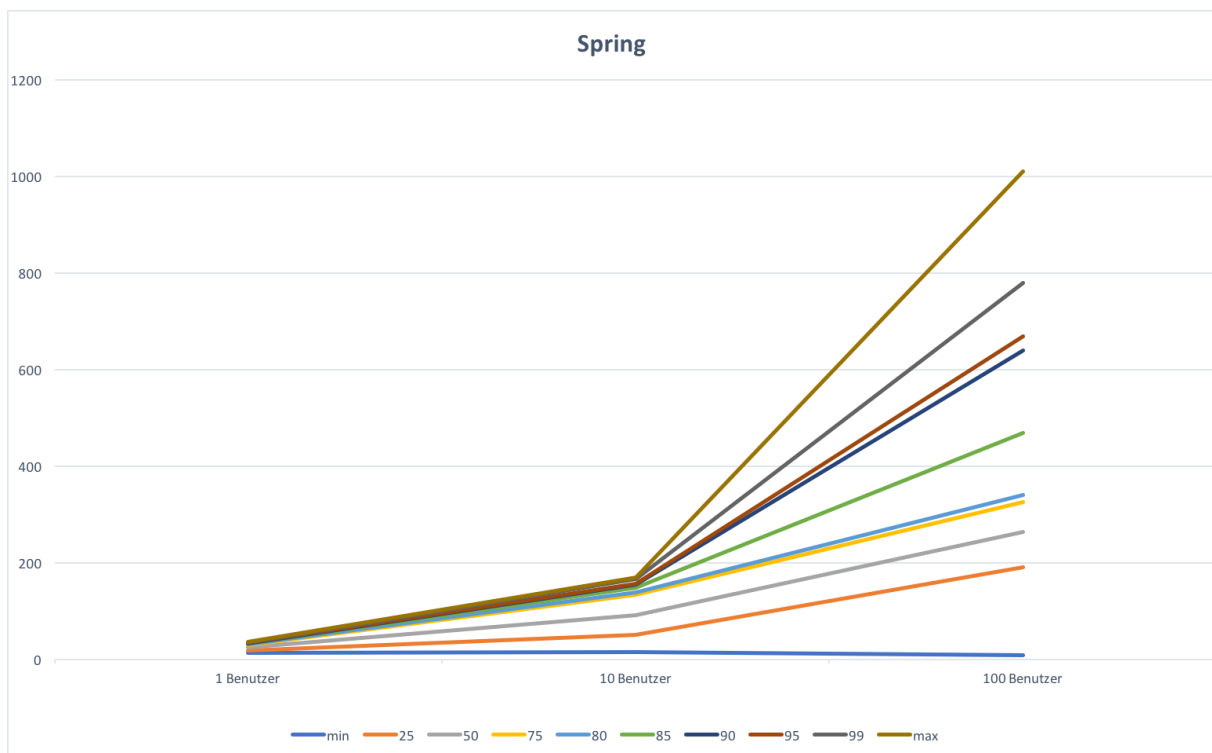


Abbildung 2.2: Diagramm der Antwortzeiten von Spring im Skalierungstest

In Abbildung 2.2 erkennt man, dass die Antwortzeiten von 85% aller Anfragen fast linear steigen. Die schnellste Antwort bleibt sogar beinahe konstant.

Fazit

In diesem Test hat das Play Framework leider sehr schlecht abgeschnitten. Die Werte des Tests sind teilweise unerklärlich, aber auch die plausiblen Resultate sind viel schlechter als die des Tests mit Spring Boot. Für dieses Kriterium ist Spring die deutlich bessere Wahl.

2.1.3 Performance

Die Performance der Anwendung ist ein weiterer wichtiger Aspekt. Das Framework soll den Code der Anwendung so effizient wie möglich einbinden und dessen Ausführung durch den Framework Code möglichst nicht beeinflussen. Um diese Eigenschaft zu quantifizieren wird ein Java Profiler verwendet, mit dem die CPU-Utilization und die Heap Grösse verglichen werden. Diese beiden Werte sind alleine nicht besonders aussagekräftig. Deshalb werden für beide Frameworks je eine Anwendung gebaut, welche dasselbe Verhalten hat. Die Werte werden mittels verschiedenen Szenarien, in denen Aktionen eines Benutzers als Flow abgebildet werden, verglichen. Der Benutzer führt einen GET-Request aus, schreibt Daten in ein Formular, sendet dieses ab und wartet auf eine Antwort. Diese gewählten Szenarien sind:

1. Der Flow wird von einem Benutzer ausgeführt
2. Der Flow wird von 10 Benutzern gleichzeitig ausgeführt
3. Der Flow wird von 100 Benutzern gleichzeitig ausgeführt

Resultate

Zur Auswertung der ermittelten Resultate werden drei verschiedene Werte verglichen.

1. CPU Auslastung: Wie stark belasten eine gewisse Anzahl von Requests das System.
2. Heap Grösse: Wie viel Heap Speicher benötigt die Anwendung um die Requests zu verarbeiten.
3. Anzahl gestarteter Threads: Wie verhält sich die Anwendung bei steigender Anzahl Requests bezüglich starten und stoppen von neuen Threads.

Diese Werte können mit der VisualVM von Oracle ermittelt und verglichen werden.

Play Play startet die JVM mit einem Heap von 1 GB. Tatsächlich benötigt das Framework im Test nie mehr als 300 MB. Die Grösse des Heap ist während des Tests etwas gewachsen, pendelt aber immer etwa zwischen 200 und 300 MB hin und her.

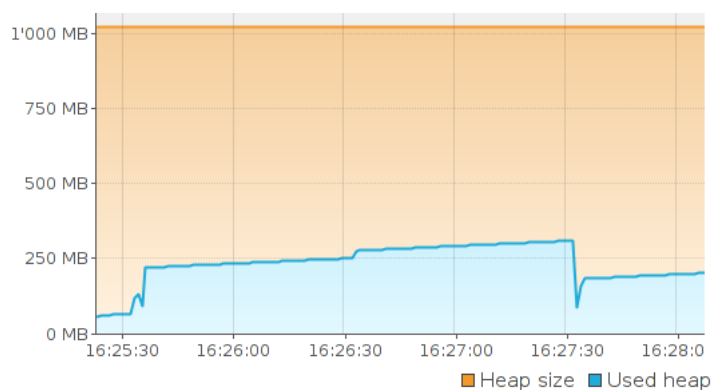


Abbildung 2.3: Heap Grösse während Lasttests im Play Framework

Die Auslastung der CPU ist etwas interessanter als des Heap. Bei diesem Diagramm sind deutliche Spitzen der Auslastung zu erkennen. Sie sind nicht besorgniserregend, deuten aber doch auf einen potentiellen Flaschenhals einer Anwendung mit mehr Applikationscode hin. In Abbildung 2.4 ist zu erkennen, dass beim Start des Tests die CPU Last stark gestiegen ist, nach der ein-minütigen Pause,

als die Phase mit den zehn Benutzern beginnt, steigt sie nochmals ein wenig, und wiederum eine Minute später, als 100 Benutzer gleichzeitig Anfragen an die Anwendung senden, steigt die Auslastung nochmals kurz an.

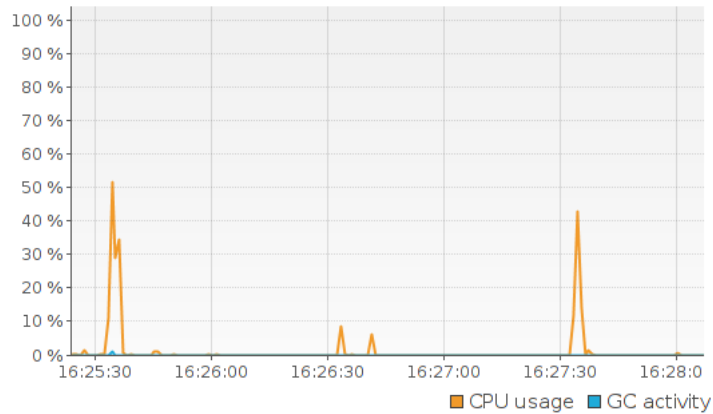


Abbildung 2.4: CPU Auslastung während Lasttests im Play Framework

Während der gesamten Ausführung des Tests hat das Play Framework nie mehr als 78 Threads gestartet. Wichtiger ist jedoch, wie oft das Framework neue Threads erstellt und abräumt, denn diese Vorgänge sind sehr teuer. Wenn man das Diagramm mit Abbildung 2.4 vergleicht, erkennt man, dass neue Threads gestartet werden, sobald die Auslastung der CPU steigt.

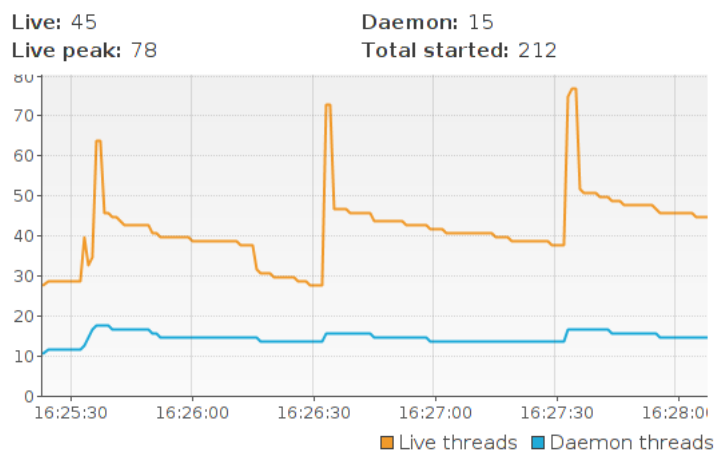


Abbildung 2.5: Anzahl Threads während Lasttests im Play Framework

Spring Das Spring Boot Framework startet mit einer Heap-Grösse von 500 MB, effektiv braucht es ebenfalls nur 200 - 300 MB. Wie auch bei Play korreliert die Allokation von neuem Speicher mit den Anzahl Requests im Test Case.

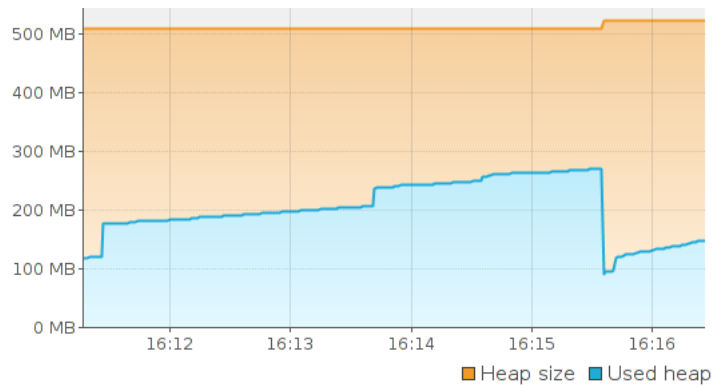


Abbildung 2.6: Heap Grösse während Lasttests mit Spring Boot

Im Vergleich zu Play sind weniger Spitzen erkennbar in der CPU Auslastung erkennbar. Diese reichen zudem lediglich bis maximal 25%. Zu Beginn des Tests gibt es einen kleinen Peak sowohl auch beim Start der Phase mit 100 gleichzeitigen Benutzern.

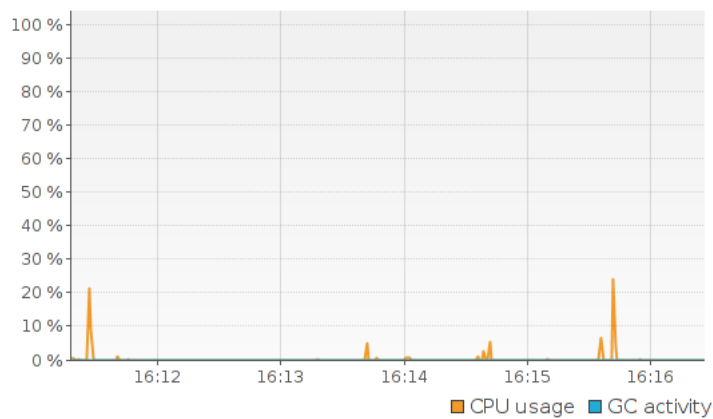


Abbildung 2.7: CPU Auslastung während Lasttests mit Spring Boot

Das Framework benötigt mehr Threads als das Play Framework. Beinahe jeder Thread, welcher das Framework startet, ist ein Daemon Thread. Die Unterschiede zu normalen Threads in Java ist, dass die Laufzeit beim Herunterfahren nicht auf eine ordentliche Beendigung dieser Threads wartet. Insgesamt startet Spring 110 Threads, wobei ein Sprung erkennbar ist, als die Phase mit den 100 Benutzern startet, vorher benötigt die Anwendung etwa 30 Threads.

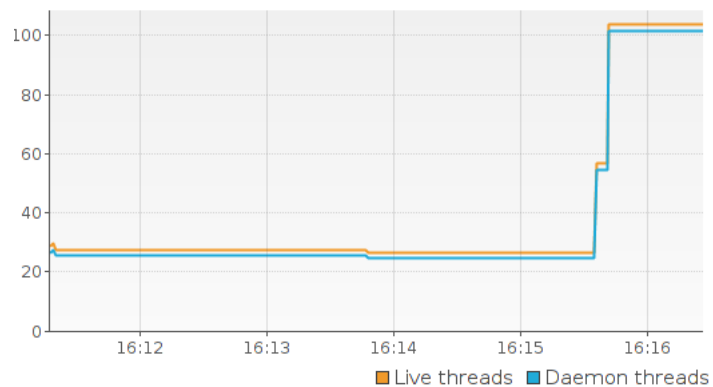


Abbildung 2.8: Anzahl Threads während Lasttests mit Spring Boot

Fazit Beide Anwendungen bieten eine gute Performance bezüglich Objekterzeugung. Play hat in diesem Beispiel eine höhere CPU Auslastung und erstellt mehr Threads als das Spring Framework. Da

genau diese beiden Kriterien enorm wichtig für eine schnelle Anwendung sind, ist Spring in diesem Punkt die bessere Wahl.

2.1.4 Cloud-Awareness

Es gibt verschiedene Kriterien für eine Anwendung, welche in einer Cloud gehostet werden soll. Ein guter Leitfaden, um eine Anwendung zu beurteilen sind die Kriterien einer IDEAL App [4]. Dieses Akronym steht für:

- **Isolated State:** Die Komponenten einer Cloud Anwendung können verteilt und auf verschiedene Ressourcen der Cloud aufgeteilt werden [4]. Dies beinhaltet sowohl allfällige Sessions, welche für die Kommunikation der Anwendung verwendet wird, sowie auch den Application-State.
Erfüllungskriterium: Das Framework erlaubt es, dass einzelne Komponenten unabhängig voneinander arbeiten können und nicht von einem internen Zustand abhängig sind.
- **Distribution:** Durch Skalierung und Modularisierung wird die Logik der Anwendung aufgeteilt. Die Komponenten können mehrere Cloud Ressourcen verwenden, oder auch in verschiedenen Teilen der Welt verfügbar sein. Eine Anwendung muss so gebaut sein, dass die einzelnen Schichten der Anwendung, ohne die Applikationslogik zu verändern, aufgeteilt werden können. Die Schichten können klassische Businessanwendungsschichten sein, sowie auch verschiedene Prozesse in einem Workflow oder Pipes und Filters.
Erfüllungskriterium Das Framework erlaubt es, die Komponenten auf die idealen Angebote der Cloud zu adaptieren.
- **Elasticity:** Eine Anwendung, welche in der Cloud läuft, kann sowohl vertikal wie auch horizontal skaliert werden. Die Leistung der Anwendung sollte darum durch die Skalierung gleichmässig verbessert werden.
Erfüllungskriterium: Das Framework belastet die Anwendung so wenig, dass eine möglichst lineare Skalierung erreicht werden kann.
- **Automated Management:** Die Anwendung benötigt im Idealfall keine menschliche Intervention. Alle Routine-Aufgaben werden durch die Anwendung selbst übernommen. Die Anwendung muss sich zudem selbst aus einem Fehler erholen können.
Erfüllungskriterium: Das Framework benötigt keine Interaktion im produktiven Betrieb. In Fehlerfällen soll es sich selbständig erholen können und Updates können automatisiert eingespielt werden.
- **Loose Coupling:** Die Kommunikation der Komponente mit anderen Komponenten soll von der Logik der Anwendung getrennt werden. Das Aufteilen in Komponenten kann die Kommunikation der Anwendung erheblich vereinfachen [4]. Die Kommunikation selbst wird über einen Broker geleitet, um die Kommunikation vom Standort und dem Zustand der Komponente zu lösen.
Erfüllungskriterium: Das Framework erlaubt es, die Anwendung in Komponenten zu unterteilen und mittels eines Kommunikationsmechanismus zu kommunizieren.

Play Framework

Das Play Framework ist ein reines Web-Framework, und kann mit beliebigen Java-Libraries und anderen Frameworks kombiniert werden. Die interne Struktur von Play macht es einem Entwickler sehr einfach, das Layers-Pattern mit Persistenz, Logik und Presentation Layer zu verwenden.

Isolated State Wird nach diesem Pattern vorgegangen, muss ein allfälliger Zustand, welcher mittels Session im Browser eines Clients verwendet wird, in einer Datenbank gespeichert werden, welche alle Instanzen für die Zustandsverwaltung verwenden. Ein Application-State, welcher das Verhalten einer Instanz im Vergleich zu anderen verändert, gibt es bei einer korrekten Anwendung des Patterns nicht.

Distribution Wenn das Design der Anwendung es verlangt, können die einzelnen Layer über eine Kommunikationsmiddleware miteinander verbunden werden. So können die einzelnen Layer modularisiert und individuell skaliert werden. Die Flexibilität einer Modularisierung hängt somit von der gewählten Middleware ab. Damit die Endanwendung von allen Vorteilen der gewählten Cloud profitiert,

muss die Applikation jedoch an die Cloud angepasst werden, da viele Cloud-Anbieter ihre Services mittels Libraries, Frameworks oder spezifischen APIs zur Verfügung stellen.

Elasticity Die Elastizität einer Anwendung ist sehr stark von der Implementierung der Logik abhängig. Wie im Performancetest in Unterabschnitt 2.1.3 ermittelt wurde, benötigt Play im allgemeinen mehr Ressourcen. Besonders die Spitzen in der CPU Auslastung deuten auf eine weniger grosse Elastizität hin, im Vergleich zum Spring Boot Framework.

Automated Management Das Framework minimiert die menschliche Interaktion sehr stark. Kann die Anwendung erfolgreich hochgefahren werden, so werden die meisten Fehler abgefangen, in das Log geschrieben und die Anwendung kehrt in einen stabilen Zustand zurück. Bei Fehlern, welche von fehlenden oder fehlerhaften Ressourcen verursacht werden, wie beispielsweise zu wenig Arbeitsspeicher oder Hardware-bedingte Ausfälle, kann nicht garantiert werden, dass sich die Anwendung wieder erholen kann.

Loose Coupling Da Play das MVC Pattern implementiert, wird der Entwickler in der Modularisierung kaum eingeschränkt. Wie auch in Abschnitt 2.1.4 (im Unterabschnitt Play-Framework, Distribution) beschrieben, kann dieses Kriterium mit der richtigen Kommunikationsmiddleware erfüllt werden.

Spring

Spring bietet viele verschiedene Frameworks an, welche in einem Spring Projekt einfach kombiniert werden können [22]. Durch das grosse Ökosystem, welches Spring einem Entwickler bietet, sowie der guten Dokumentation der Projekte und Komponenten, ist es sehr einfach, leichtgewichtige und schnelle Komponenten zu bauen. Wie auch das Play Framework macht es einem das Spring Web Framework sehr einfach, Anwendungen mit einem der Model-View Pattern zu erstellen.

Isolated State Bei einer konsequenten Implementierung des MVC Pattern gibt es keinen Application State, der zwischen Instanzen ausgetauscht werden muss. Einzig der Zustand der Clients muss über alle Instanzen konsistent sein. Spring bietet für das Management der Session zwei verschiedene Frameworks an, Spring Security für Login-Informationen und Spring Session für benutzerdefinierten Inhalt einer Session. Es ist möglich beide zu kombinieren, sowie können auch beide Frameworks die Session Objekte in eine Datenbank auslagern.

Distribution Eine Verteilung der Komponenten auf verschiedene Cloud-Angebote ist mit dem Spring Ökosystem gut möglich. Spring bietet verschiedene Frameworks für Integrationen in bestehende Systeme und der Umsetzung von Cloud Patterns an. Somit wird es für einen Entwickler sehr einfach, eine Anwendung zu modularisieren und deren Komponenten auf verschiedene Angebote zu verteilen. Wenn eine Cloud-spezifische API verwendet werden soll, kann mit Spring sehr einfach über die Dependency Injection ein Adapter geschrieben werden, der die bestehende Infrastruktur an die neue Cloud Umgebung anpasst.

Elasticity Spring Web ist ein sehr leichtes Framework. Wie man im Unterabschnitt 2.1.3 sehen kann, ist die Anwendung sparsam mit Ressourcen und belastet die CPU weniger als das Play Framework. Es liegt die Vermutung nahe, dass komplexe Implementierungen in Spring weniger zulasten der Performance sind.

Automated Management Wie auch beim Play Framework ist nach dem Start der Anwendung kaum Interaktion mehr nötig. Die Anwendung stabilisiert sich nach einem Fehler selbst, Updates können durch einfaches Ersetzen der Anwendung eingespielt werden und benötigen danach keine weiteren Schritte.

Loose Coupling Auch dieses Kriterium erfüllt Spring Web durch das vielfältige Ökosystem von Komponenten. Mit Kommunikationsmiddlewares, wie beispielsweise Spring AMQP oder Spring CloudBus, ist es sehr einfach, Komponenten voneinander zu trennen und miteinander kommunizieren zu lassen. Zudem bietet Spring viele Möglichkeiten etablierte Cloud Patterns in eine bestehende Anwendung zu implementieren, ohne die Logik verändern zu müssen.

Fazit

Beide untersuchte Frameworks können in einer Cloud betrieben werden. Es gibt lediglich zwei Punkte, welche für Spring sprechen. Erstens sind die Unterschiede in der Performance zwar klein, können aber bei grossen Anwendungen eine erhebliche Rolle spielen, besonders wenn die Anwendung unter hoher Last steht. Zweitens bietet das Spring Ökosystem viele, fast fertige Lösungen an, um eine Anwendung zu erweitern. Dies bietet einem Entwickler sehr viel Flexibilität und erhöht die Robustheit einer Anwendung, da sich grosse Teile einer Logik im Framework befinden.

2.1.5 Ergebnis

Die Wahl des Web-Frameworks ist eine richtungsweisende Entscheidung. Besonders bei einer falschen Evaluation des Frameworks können im Verlauf der Arbeiten Schwierigkeiten und unnötige Probleme auftreten. Die aufgestellten Kriterien sind von Spring alle besser erfüllt worden als vom Play-Framework. Spring hat mit Spring-Boot ein modernes und modulares System entwickelt, zu welchem sehr einfach neue Komponenten hinzugefügt und entfernt werden können. Besonders aber die Dokumentation macht bei Spring einen viel besseren Eindruck als beim Play-Framework. Viele Informationen über die Interna von Spring sind in der Dokumentation ersichtlich und es ist selten nötig, die JavaDoc des Frameworks zu benutzen. Eigenschaft, welche sehr oft in den Hintergrund rückt, ist die Rückwärtskompatibilität. In Spring bleiben viele Schnittstellen sogar über Major Versionen stabil[24].

Für die Evaluation mussten die beiden Beispielanwendungen als produktive Instanzen deployed werden. In Spring-Boot wird die komplette Software in ein JAR kompiliert und ist somit ausführbar. In Play war es eher mühsam, eine produktive, lauffähige Version zu erzeugen. Besonders ein Bug im Persistenz-Framework kostete beim Testen viel Zeit.

Aufgrund der aufgestellten Kriterien und den Eindrücken von beiden Frameworks während den Tests, ist die Wahl auf Spring gefallen. Aufgrund der Erkenntnisse der Evaluation, des grossen Ökosystems von Modulen, der ausgezeichneten Dokumentation und der einfachen Integration neuer Komponenten, bietet dieses Framework alles, was für eine schnelle und stabile Anwendung nötig ist.

2.2 Persistenz Framework

Das Persistenz Framework ist ein wichtiger Teil der Anwendung. Allem voran ist es für einen effizienten Zugriff auf die Datenbank verantwortlich.

JDBC Die simpelste Möglichkeit in Java auf eine Datenbank zuzugreifen, ist über JDBC direkt mit Prepared Statements zu operieren. Diese ist zweifelsohne die schnellste Möglichkeit. Alle Mechanismen, wie etwa Lazy-Loading, Abhängigkeitsauflösung oder Caching müssen jedoch selbst entwickelt werden.

Hibernate Ein sehr beliebtes Framework ist Hibernate. Es implementiert den JPA Standard [26], eine Schnittstelle, welches das Konvertieren und Zuordnen von Java Objekten zu Datenbankentitäten vereinfacht. Mit Hibernate greift man über ein Repository auf eine Tabelle in der Datenbank zu. Die Auflösung von Abhängigkeiten, der Zugriff darauf sowie das Generieren der Queries und das Auslesen der Daten übernimmt das Framework. Um die Daten zu transformieren verwendet Hibernate entweder Reflection, um auf Felder einer Klasse zuzugreifen, oder kann über einen sogenannten Persistence-Context konfiguriert werden. Letzterer wird über XML Dateien konfiguriert, welche die Daten jedes Objektes in der Datenbank beschreibt. Beide Mechanismen sind jedoch nicht sehr effizient.

JOOQ JOOQ ist ein sehr fortgeschrittener Query Builder für Java. JOOQ ist ein Database First Tool[10], welches Queries generieren kann und direkt auf der Datenbank ausgeführt werden können. Das Framework beinhaltet einen Code-Generator, welcher komplette Datenbanken analysieren und Entity Objekte generieren kann. Die Queries sind somit typisiert und die empfangenen Daten können automatisch in Java Objekte konvertiert werden. Da JOOQ intern ebenfalls auf JDBC setzt, sind die Einbusen in der Performance minimal.

Kriterien

Alle drei Technologien haben sich in der Praxis bewährt und sind stabil und zuverlässig. Im Anzeiger beschränkt sich die Verwendung einer Persistenz auf das schnelle Lesen aus einer Datenbank. Der Anzeiger ist nicht von spezifischen Features eines bestimmten Datenbankherstellers abhängig und kann gut mittels ORM auf die Datenbank zugreifen. Der Import neuer Daten hat höhere Anforderungen an die Persistenzschicht, denn es müssen einerseits schnell viele Daten in die Datenbank geschrieben werden können, andererseits müssen die Daten bei Bedarf ebenfalls schnell aus der Datenbank in die Anwendung kommen.

2.2.1 Fazit

Die verschiedenen Komponenten haben unterschiedliche Anforderungen an eine Persistenzschicht. Es ist sehr unwahrscheinlich, dass es möglich ist, in allen Komponenten dasselbe Produkt zu verwenden. Zudem ist es sinnvoll im Rahmen des Prototyps die verschiedenen Lösungen zu implementieren um unterschiedliche Lösungsansätze aufzuzeigen.

Anzeige Die Anzeigekomponente ist eine gewöhnliche Web-Anwendung, in welche sich Hibernate sehr gut integrieren lässt. Hibernate bietet eine einfache Möglichkeit um auf Objekte in der Datenbank zu navigieren. Es übernimmt die Fehlerbehandlung und kümmert sich um das ordnungsgemässe Abräumen der Datenbankverbindung. Zudem bietet die Spring Integration von Hibernate, Spring JPA, sehr einfache Schnittstellen für einen Datenzugriff an. Im besten Fall benötigt jede Tabelle lediglich ein Interface, auf dessen Methoden Queries definiert werden. Die Logik, welche für den Zugriff auf die Datenbank nötig ist, ist somit komplett im Framework versteckt.

Import Komponenten Die beiden Import Komponenten müssen schnell auf Daten zugreifen und diese speichern können. Besonders für das Prüfen und Setzen von Constraints in der Datenbank benötigt Hibernate sehr viel Zeit. Ein besonders wichtiger Punkt bei der Evaluation einer Lösung ist, dass die Queries beim Importieren der Daten wiederverwendet werden können. Ein Framework, welches Queries generieren muss, würde dafür unnötig Zeit benötigen. Mit JDBC ist es mit `PreparedStatement`s möglich, dieselben Queries mit verschiedenen Datensätzen wiederverwenden zu können. Das Konstruieren von Objekten aus einer Antwort der Datenbank muss mit JDBC selbst implementiert werden. Um die Importkomponente nicht unnötig an einen Datenbank-Hersteller zu binden, kann JOOQ als Query-Builder verwendet werden, welcher Queries in verschiedenen SQL-Dialekten generieren kann. Für die Importkomponenten ist JOOQ somit eine ideale Lösung, da es sowohl Queries generieren kann, diese aber auch direkt ausführen kann.

2.3 Datenimport

Der Import und die Transformation der Daten wird in Module aufgeteilt. Dabei gibt es verschiedene Anforderungen an die Module. Der Import der Echtzeitdaten muss vor allem schnell sein, während der Import der statischen Daten grosse Mengen effizient verarbeiten können muss. Die statischen Daten können mehrere 100 MB oder einige GB gross sein, eine Anforderung, mit der das gewählte Framework zurechtkommen muss.

2.3.1 Batch Import

Für den Import der statischen Daten wird eine Lösung benötigt, welche effizient und schnell grosse Datensätze einlesen, transformieren und schreiben kann. Dass Batch-Prozesse heute noch von grosser Bedeutung sind, zeigt der JSR 325, welcher von einigen grossen Firmen, darunter Ericsson, im Jahre

2010 erstmals veröffentlicht wurde [1]. In Java gibt es zwei etablierte Implementationen dieses Standards, JBatch und Spring Batch. Aufgrund der Entscheidung für Spring Web im Front-End, wurde bei dieser Evaluation das Augenmerk auf Anforderungen gelegt, welche für das Importieren kritisch sind, aber von Spring Batch nicht erfüllt werden können. Eine einheitliche Landschaft über Frameworks hilft dem Verständnis der Anwendung sehr und minimiert zudem den Wartungsaufwand.

Der Import von CSV Daten ist ein klassischer Anwendungsfall für ein Batch Framework. Spring Batch unterstützt den Entwickler mit vielen vorgefertigten Klassen zum Lesen und Schreiben von Daten in verschiedene Datenquellen.

Fazit

Spring Batch ist ein sehr beliebtes Framework für die Stapelverarbeitung. Es ist sehr gut dokumentiert, bietet für Standardfälle viele konfigurierbare Methoden an um Daten zu verarbeiten und ist zudem noch sehr schnell. Deswegen fiel die Wahl auf Spring Batch.

2.3.2 Realtime Import

Der Import der Echtzeitdaten ist etwas trickreicher. Diese Daten sind hierarchisch und müssen beim Importieren komplett verarbeitet werden um alle benötigten Informationen zu erhalten [6]. Ein Batch Framework ist deshalb nicht für den Import geeignet. Nach intensiver Recherche nach geeigneten Lösungen wurde der Ansatz, ein Framework zu Hilfe zu nehmen, verworfen. Aus diesem Grund wird nicht auf ein Framework für den Datenimport zurückgegriffen. Um die Systemlandschaft einheitlich zu halten, wird aber auch für dieses Modul (wo möglich) die Spring Plattform verwendet. Vor allem Dependency Injection, Anbindung an die Datenbank mittels JDBC und Konfiguration der Anwendung kann mittels Spring sehr einfach umgesetzt werden.

Fazit

Mangels Frameworks wird für dieses Modul eine eigene Lösung entwickelt.

2.4 Kommunikation

In einer Cloud Umgebung müssen Module und Komponenten miteinander kommunizieren können. Diese Anforderung wird vom Trend zu Microservices angetrieben. Kommunikation zwischen Komponenten kann von sehr einfachen Trigger bis hin zu komplexen RPC-Aufrufen alles beinhalten. Für die Evaluation einer Technologie und zu einem Proof-Of-Concept wird ein einfaches Use-Case gewählt und implementiert. Es gibt viele verschiedene Lösungen für unterschiedliche Szenarien, um eine Kommunikation zwischen Komponenten zu realisieren [9]. Eine einfache Form der Kommunikation könnte über die Datenbank implementiert werden. Eine solche Lösung ist für die statischen- und Echtzeittransportdaten verwendet worden. Diese Lösung erlaubt jedoch keine direkte Verarbeitung von Events. Eine in Java sehr beliebte Lösung ist die Verwendung des Java Messaging Service (JMS), eine RPC Infrastruktur für Java [25]. Leider ist diese Lösung auf die Verwendung in Java limitiert, was die Erweiterbarkeit des Produkts unnötig einschränkt. Seit einigen Jahren gewinnt der AMQP Standard an Beliebtheit. Sehr viele Plattformen bieten bereits Client-Implementationen des Standards an [16]. Ausserdem bieten eine wachsende Anzahl an Cloud-Anbietern AMQP kompatible Services an. Aufgrund der Beliebtheit, Portabilität und Stabilität des Standards beschränkt sich die Evaluation auf eine Architektur mittels AMQP.

2.4.1 Architektur

Bei der Wahl eines Kommunikationssystems stellt sich die Frage nach der gewünschten Topologie. Grundsätzlich gibt es zwei Ansätze in AMQP: Der erste Ansatz sieht die Verwendung eines Brokers vor, welcher die Nachrichten annimmt, entsprechend verarbeitet und an den oder die Empfänger zustellt. Ein zweiter Ansatz ist ein System ohne Broker, welches sich selbst organisieren kann und die Komponenten direkt miteinander kommunizieren. Die Wahl dieser Architektur ist für die Skalierbarkeit und vor allem für die Fehlertoleranz des Systems entscheidend. Die Kriterien, nach denen die beiden Konzepte evaluiert wurden, sind:

1. **Komplexität:** Wie einfach ist es, die Struktur einzurichten, zu überwachen und Fehler darin zu finden.
2. **Skalierbarkeit:** Wie skaliert das System bei wachsender Anzahl Teilnehmer
3. **Verwendung:** Wie erfolgreich wird die Architektur in bestehenden Systemen eingesetzt.

Broker System Auf den ersten Blick ist ein Broker-System einfacher, bietet aber weniger Fehlertoleranz. Im einfachsten Fall mag diese Hypothese korrekt sein. Bei einer solchen Behauptung wird an ein monolithisches System gedacht. Jedoch muss ein Broker-System nicht zwingend einen einzelnen Server oder ein Cluster sein, welche alles verwalten. Ein sehr berühmtes Beispiel eines dezentralen, skalierenden Broker-Systems ist SMTP. Ein Mail Transfer Agent verteilt und routet E-Mails, welche an ihn weitergeleitet werden [11], genau wie ein Broker. Viele Broker-Hersteller bieten die Möglichkeit, einen Cluster aus Brokern zu verwenden [20] [5], diese können aber auch dezentral funktionieren. Ein Broker-System hilft zudem, ein komplexes Netzwerk zu abstrahieren und vor allem das Finden von Komponenten zu vereinfachen.

Selbstorganisierendes System Ein System ohne organisierende Komponente muss sich besonders um drei Punkte kümmern: Discovery, Verfügbarkeit und Management. Diese Punkte müssen entweder vom verwendeten Produkt, oder durch den Anwendungscode berücksichtigt werden. Besonders wichtig bei einem solchen System ist, dass die Topologie möglichst simpel bleibt, denn in einem solchen System ist es schwer herauszufinden, ob das Netzwerk gestört oder eine Nachricht nicht angekommen ist [15]. Zudem bietet ein dezentrales System keine einheitliche Verwaltung an. Besonders Identitätsmanagement von verschiedenen Kommunikationskanälen können in solch einem System nur schwer bis gar nicht umgesetzt werden.

Fazit In einfachen Netzwerken und Topologien bietet ein dezentrales, selbstorganisierendes System einige Vorteile gegenüber einem Broker System. Doch in den komplexen Netzwerken eines Cloud-Anbieters kann kaum von solch einem Fall ausgegangen werden. Ausserdem bietet ein Broker-System mehr Flexibilität und Fehler sind einfacher zu finden. Pivotal, der Hersteller eines sehr beliebten Broker Systems, hat vor ein paar Jahren gezeigt, dass mit einem Cluster aus Broker mehr als 1 Mio. Messages pro Sekunde verarbeitet werden können [12]. Da die Kommunikation im Prototyp lediglich zur Demonstration eines möglichen Use-Cases eingebaut wird, ist die Verwendung eines flexibleren Broker-System die bessere Wahl.

2.4.2 Frameworks

Das gesuchte Produkt muss viele Anforderungen erfüllen können. Da der Einsatz der Technologie noch nicht festgelegt ist, muss das Produkt viel Funktionalität besitzen, damit ein Wechsel auf eine andere Technologie in Zukunft vermieden werden kann. Zudem ist es wichtig, dass das Produkt bereits in der Industrie Verwendung findet, denn ein gut etabliertes System wird voraussichtlich auch die nächsten Jahre weiter existieren. Ebenfalls wichtig ist das Framework oder die Library für die Kommunikation. Die Verwendung davon sollte sich gut in die bestehende Architektur integrieren lassen und wenig Anwendungscode erzeugen. Im Idealfall muss die Anwendungslogik lediglich das Konfigurieren einer Queue, das Senden und Empfangen beinhalten. Dementsprechend sind folgende Kriterien aufgestellt worden:

- **Funktionalität:** Was unterstützt die Software, wie ist die Dokumentation und welches Wissen muss ein Entwickler mitbringen, um das Produkt verwenden zu können
- **Verwendung:** Wie viel Konfigurationscode ist nötig, um Nachrichten zu senden und empfangen, wie werden Empfänger registriert und wie gut lässt es sich in die bestehende Architektur mit Spring integrieren.

Spring Cloudbus/Spring AMQP

Spring bietet bereits eine Messaging Lösung basierend auf RabbitMQ an. Da Spring und RabbitMQ von Pivotal produziert werden, kann davon ausgegangen werden, dass Spring auch in Zukunft auf RabbitMQ setzen wird und dieses Framework schnell neue Features von RabbitMQ unterstützen wird.

RabbitMQ ist in Erlang geschrieben [14], einer funktionalen Sprache, welche in den 80er Jahren für eine Form von Messaging entwickelt worden ist.

Funktionalität RabbitMQ bietet bereits in der Standardinstallation sehr viele Features. Äusserst wichtig in einem Messaging System sind Cluster und Federations. Eine Federation ist vergleichbar mit einem Cluster, mit einem wichtigen Unterschied: Jeder Broker in einer Federation ist für andere Queues zuständig, vergleichbar mit den MTA im globalen E-Mail Verkehr [17]. Beide Features werden von RabbitMQ unterstützt. Zudem ist es mit RabbitMQ möglich, High-Availability Queues zu verwenden, womit ein Single Point of Failure (SPOF) eliminiert werden kann [19]. Ausserdem bietet RabbitMQ die Möglichkeit, Nachrichten mittels Tracing zu verfolgen [18] um Fehler einfacher zu finden.

Ein sehr mächtiges und wichtiges Feature von RabbitMQ ist das Plug-In-System. Dies erlaubt es Drittentwicklern Erweiterungen für den Broker zu schreiben. Es können beispielsweise neue Arten von Exchanges eingebunden, neue Protokolle unterstützt, Authentisierung und Autorisierung oder das Management des Brokers erweitert werden. Ein weiterer sehr wichtiger Punkt ist die Dokumentation, welche für RabbitMQ sehr detailliert ist. Für jedes Feature und offizielle Plug-In gibt es eine sehr ausführliche Dokumentation. Ausserdem gibt es ein sechsteiliges Tutorial für 8 Programmiersprachen und Plattformen.

Verwendung Da Spring bereits RabbitMQ als Framework integriert, müssen lediglich die Abhängigkeiten davon konfiguriert werden. Danach können Exchanges und Queues konfiguriert und in der gesamten Anwendung verfügbar gemacht werden. Eine einfache Queue mit einem Consumer und einem Producer kann mittels Dependency Injection konfiguriert werden.

```
@Configuration
public class RabbitConfig {

    @Bean
    public Queue helloQueue() {
        return new Queue("hello");
    }

    @Bean
    public RabbitReceiver receiver() {
        return new RabbitReceiver();
    }

    @Bean
    public RabbitSender sender() {
        return new RabbitSender();
    }
}
```

Listing 5: Basis Konfiguration von Spring AMQP

Die Klasse `RabbitSender` kann mit einem `RabbitTemplate` Nachrichten an eine Queue senden.

```

@Component
public class RabbitSender {

    @Autowired
    private RabbitTemplate template;

    @Autowired
    private Queue queue;

    public void send() {
        String message = "Hello World!";
        this.template.convertAndSend(queue.getName(), message);
        System.out.println(" [x] Sent '" + message + "'");
    }
}

```

Listing 6: Klasse zum Senden einer Message in Spring AMQP

Die Klasse RabbitReceiver kann mittels Annotation konfiguriert werden, um Nachrichten aus einer Queue zu lesen.

```

@RabbitListener(queues = "#{helloQueue.name}")
public class RabbitReceiver {

    @RabbitHandler
    public void receive(String in) {
        System.out.println(" [x] Received '" + in + "'");
    }
}

```

Listing 7: Klasse zum Empfangen einer Message in Spring AMQP

Apache ActiveMQ

ActiveMQ ist ein Message Broker der ursprünglich für JMS entwickelt wurde, aber seit einiger Zeit auch AMQP unterstützt. ActiveMQ wird von der Apache Foundation entwickelt, weshalb man davon ausgehen kann, dass auch dieses Produkt eine Zukunft in der Java Welt hat. Deshalb bietet er alle Features an, welche von AMQP vorgeschrieben sind.

Funktionalität ActiveMQ unterstützt neben AMQP auch noch weitere Protokolle wie MQTT, STOMP und OpenWire. Zudem bietet es eine Infrastruktur für Clustering, aber leider keine Federations. Trotzdem bietet ActiveMQ einen grossen Funktionsumfang. Es ist beispielsweise möglich, via Logging Interceptor den Nachrichtenfluss zu beobachten und protokollieren. Auch ActiveMQ bietet bereits eine Spring Integration an, welche die Konfiguration des Clients grösstenteils übernimmt. Die Dokumentation ist gut, wenn es auch teilweise nötig wäre, etwas ausführlicher zu dokumentieren.

Verwendung ActiveMQ bietet bereits eine Integration in das Spring Framework an. Die Konfiguration der Queues und Exchanges benötigt zusätzlich eine Annotation, welche ActiveMQ aktiviert. Der Name `@EnableJms` ist etwas irreführend, denn JMS wird nicht zwingend verwendet.


```

@Configuration
@EnableJms
public class ActiveMqConfiguration {
    @Bean
    public Queue queue() {
        return new ActiveMQQueue("sample.queue");
    }
}

```

Listing 8: Konfiguration einer Anwendung mit ActiveMQ

Ein Producer kann mittels JmsMessagingTemplate eine Message an den Broker senden.

```

@Component
public class Producer implements CommandLineRunner {

    @Autowired
    private JmsMessagingTemplate jmsMessagingTemplate;

    @Autowired
    private Queue queue;

    @Override
    public void run(String... args) throws Exception {
        send("Sample message");
        System.out.println("Message was sent to the Queue");
    }

    public void send(String msg) {
        this.jmsMessagingTemplate.convertAndSend(this.queue, msg);
    }
}

```

Listing 9: Klasse zum Senden von Messages in ActiveMQ

Um zu empfangen muss, wie schon bei RabbitMQ, eine Methode mittels Java Annotation markiert werden.

```

@Component
public class Consumer {

    @JmsListener(destination = "sample.queue")
    public void receiveQueue(String text) {
        System.out.println(text);
    }
}

```

Listing 10: Klasse zum Empfangen einer Message in ActiveMQ

Qpid

Qpid ist ein sehr neues Produkt und wird, wie ActiveMQ, von Apache entwickelt. Es ist komplett in Java geschrieben und ist im Vergleich zu ActiveMQ ein eher junges Projekt.

Funktionalität Die Features von Qpid sind im Vergleich zu ActiveMQ und vor allem RabbitMQ eher bescheiden. Qpid Proton ist das AMQP Toolkit, wie es die Entwickler nennen, welches in Java verwen-

det wird. Qpid unterstützt Cluster und Federations und bietet ausserdem viele Mechanismen an, um eine Anwendung bei einem Broker zu authentisieren. Qpid bietet leider keine Spring Integration an, wodurch die komplette Konfiguration des Frameworks im Anwendungscode erfolgt.

Verwendung Trotz mehreren Versuchen, ist es nicht gelungen, Qpid in Spring zu integrieren. Der vielversprechendste Versuch wurde mit Zuhilfenahme des offiziellen Repository mit Beispiel Implementationen durchgeführt (<https://github.com/apache/qpid-jms/tree/0.23.0/qpid-jms-examples>). Dieses Scheitern ist nicht zuletzt auf die lückenhafte und teilweise nicht vorhandene Dokumentation des Produktes zurückzuführen.

Fazit

Eine Anforderung wie diese abzudecken ist schwierig. Da es sich um einen Proof-Of-Concept handelt, sind die Anforderungen an ein Produkt vage und unspezifisch. Umso wichtiger ist es, die Lösung zu wählen, welche die beste Rahmenbedingung, die höchste Flexibilität und vor allem den grössten Funktionsumfang hat, um zukünftige Anforderungen erfüllen zu können. Diese Punkte hat RabbitMQ auf ganzer Linie erfüllt. Die Dokumentation und die Anleitungen sind beispielhaft und sehr gut verständlich. Die Integration in Spring ist sehr einfach und die Zukunft der Lösung in Spring ist durch den gemeinsamen Hersteller gesichert.

Kapitel 3

Architektur

Dieses Kapitel beschreibt die entworfene Architektur und erläutert Entscheidungen, welche dazu geführt haben.

3.1 Systemarchitektur

Beim Gestalten der Architektur wurde stark auf die Skalierbarkeit geachtet. Um eine solche Architektur zu erarbeiten, müssen die zu skalierenden Teile ausfindig gemacht werden. In diesem Projekt ist der Abfahrtsanzeiger eine solche Komponente. Zudem müssen mögliche Flaschenhälse detektiert werden. Diese Engpässe sind in den Import-Modulen und in der Datenbank-Komponente. Damit eine Skalierung des Abfahrtsanzeigers und der Datenbank ermöglicht werden kann, ist die gesamte Architektur in drei Komponenten aufgeteilt worden: „Static-Import“, „Realtime-Import“ und „Anzeige-komponente“. Die Datenbank bildet eine Abstraktion zwischen den Import Modulen und der Anzeige-komponente, welche die Skalierung der Applikation ermöglicht.

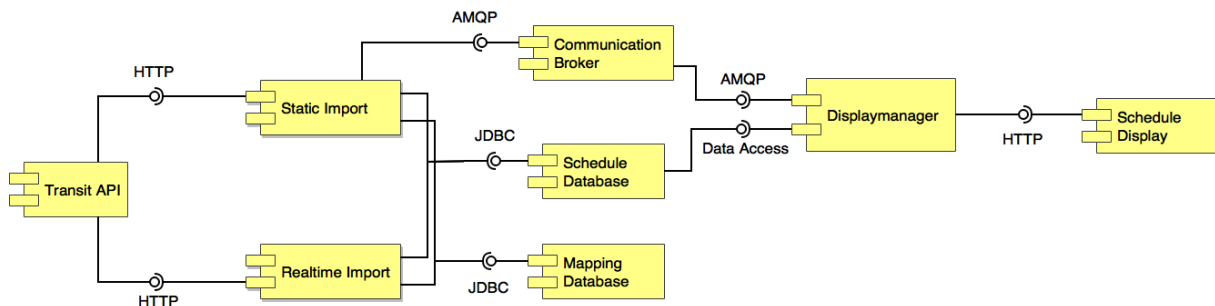


Abbildung 3.1: Überblick der Komponenten

Die beiden Importkomponenten erhalten die zu importierenden Daten durch Zugriffe auf eine HTTP API. Neue Versionen werden in regelmässigen Abständen heruntergeladen und verarbeitet. Diese Daten werden in eine gemeinsame Datenbank geschrieben und von der Anzeigekomponente in einem kleinen Verarbeitungsschritt für die Anzeige aufbereitet.

Die Aufteilung der einzelnen Aufgabenbereiche in unabhängige Komponenten hilft, die Logik, welche für die Anzeige der Daten nötig ist, möglichst simpel zu halten. Somit wird die Anzeige, durch die Zeit, welche für den Import der Daten benötigt wird, nicht beeinflusst. Durch Versionierung, beschrieben in Abschnitt 4.3, der Datenbank können sich die verschiedenen Komponenten nicht gegenseitig stören. Die Import-Module müssen nicht horizontal skalieren, sondern lediglich eine ausreichende Performance mit sich bringen. Was eine ausreichende Performance ist, hängt stark von der Datenquelle und der gewünschten Aktualität der Echtzeitinformationen ab.

3.1.1 Organisation

Um eine Code Duplizierung zwischen den einzelnen Komponenten des Projektes zu vermeiden, ist ein weiteres Projekt („shared“) verwendet worden. Das untenstehende Package-Diagramm zeigt die wichtigsten Komponenten der gesamten Architektur.

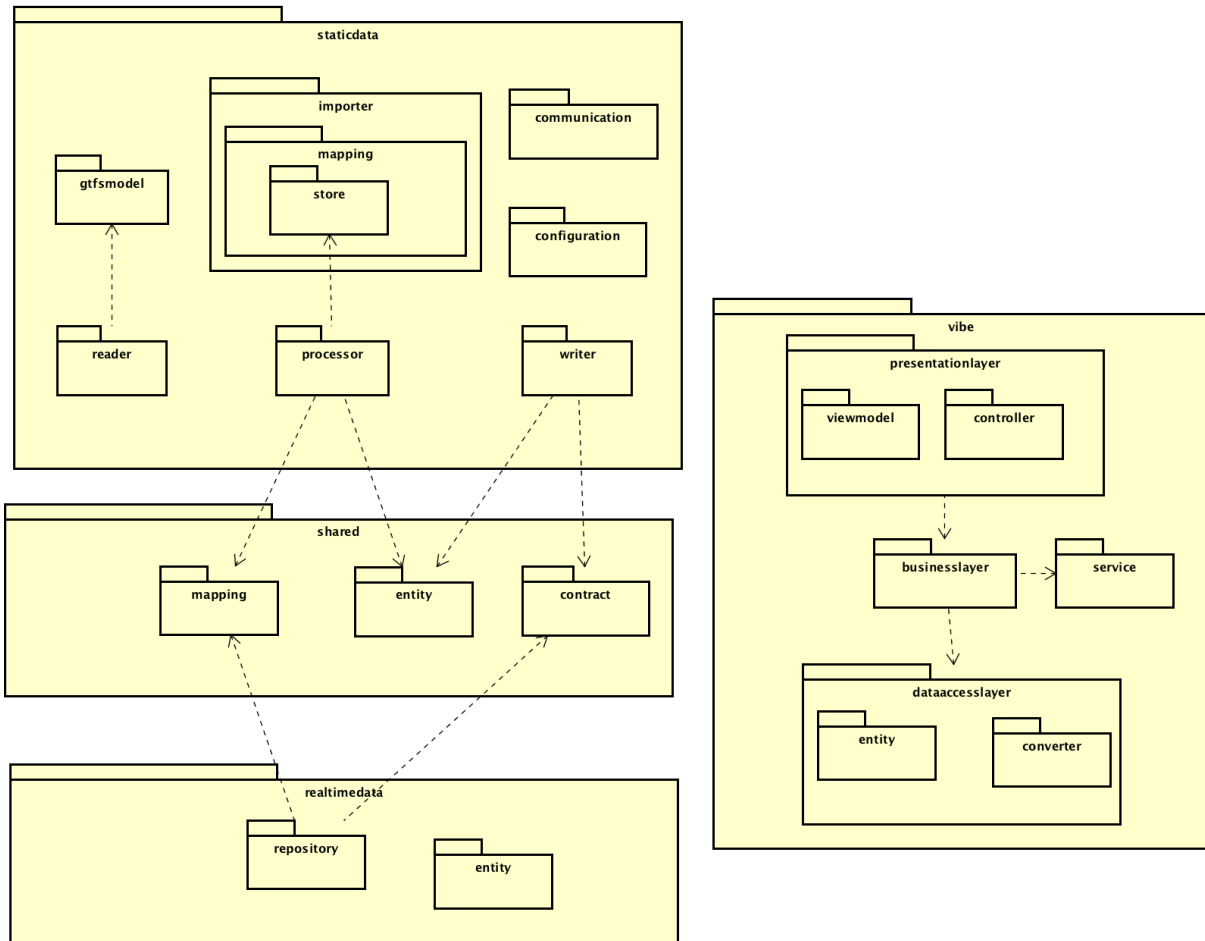


Abbildung 3.2: Package Diagramm

Die in Abbildung 3.2 dargestellten Komponenten zeigen das Zusammenwirken der verschiedenen Komponenten innerhalb der Module. Gemeinsam genutzte Komponenten sind in ein externes Package ausgelagert worden.

3.1.2 Datenquellen

Die Datenquellen sind im Prototyp APIs, welche GTFS Daten anbieten. Durch die Definition des Datenbankschemas ist die Erweiterbarkeit des Systems gewährleistet. Es ist möglich, Daten in einem anderen Format (beispielsweise eine VDV Norm oder HAFAS Daten) in die Datenbank einlesen zu können.

GTFS Static Feed Die statischen Verkehrsdaten werden vom Betreiber im GTFS Format als ZIP Datei angeboten, welche CSV Dateien beinhaltet (siehe Unterabschnitt 1.1.1). Im Prototyp werden die Daten jeden Tag um 2:00 Uhr nachts geladen und in die Datenbank geschrieben. Der gesamte Vorgang dauert bei ca. 33 MB Daten vier Minuten. Das Intervall des Imports hängt vom Anbieter der Daten ab, es kann daher keine Empfehlung abgegeben werden, wie oft die Daten aktualisiert werden sollen.

GTFS Realtime Feed Die Echtzeitdaten werden vom Betreiber im Protocol Buffers Format zur Verfügung gestellt. Für Java gibt es Packages, sodass die Java Klassen nicht aus dem Protobuf Format kompiliert werden müssen. Der Import der Daten dauert, bei etwa 5000 Datensätzen, 10 Sekunden. Im Prototyp werden alle 30 Sekunden neue Daten heruntergeladen. Da der Standard noch keine inkrementelle Updates anbietet [6], werden die vorherigen Updates gelöscht und die neuen Daten eingefügt.

Das Zeitintervall von 30 Sekunden ist für den Prototyp ideal. Ein anderer Wert kann in der Konfigurationsdatei eingestellt werden. Wichtig ist, die Verzahnung der Updateintervalle zu beachten. Wenn ein Server beispielsweise die Daten im Zeitintervall X aktualisiert und die Applikation die Daten im Zeitintervall Y vom Server herunterladet, so ist mit einer Verspätung von maximal $X + Y - 1$ Zeiteinheiten zu rechnen. Folgendes Bild hilft dieses Verhalten zu veranschaulichen.

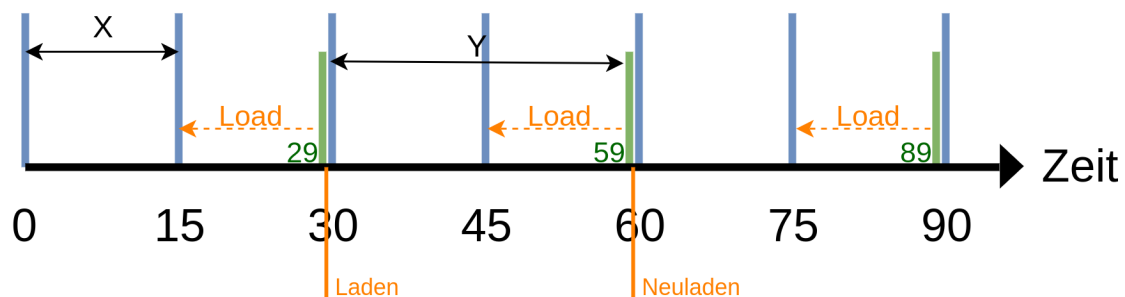


Abbildung 3.3: Zeitverschiebung der Updates

Auf dem Bild erkennt man, dass nach dem ersten Laden die Daten auf den Zeitpunkt 15 Zeiteinheiten aktualisiert sind. Kurz vor dem zweiten Update, bei Zeitpunkt 59, sind die aktuellen Daten immer noch dieselben wie beim ersten Laden. Das Alter der Daten ist somit $15 + 30 - 1 = 44$ Zeiteinheiten. Nach dem Update wurden die Daten auf den Stand der Zeiteinheit 45 aktualisiert und das Ganze wiederholt sich. Beim Konfigurieren des Updates-Intervalls ist es wichtig, diese Zeitverschiebung zu beachten.

3.1.3 Fahrplan Datenbank

Die Datenbank, in welche die Fahrplandaten geschrieben werden, orientiert sich stark an dem in Abschnitt 3.2 definierten Domänenmodell. Es ist eine relationale Datenbank und fasst alle für die Anzeige essentielle Informationen zusammen.

3.1.4 Mapping Datenbank

Damit das Produkt mit verschiedenen Datenquellen und vor allem mit verschiedenen ID-Formaten umgehen kann, verwendet die Fahrplan Datenbank selbst definierte IDs. Damit aber die Importkomponenten die Relationen zwischen beiden ID-Systemen nicht verlieren, müssen diese Daten separat noch abgespeichert werden. Dies ist für das Verknüpfen der Echtzeitdaten, einem Update der statischen Daten oder bei überschneidenden Daten aus verschiedenen Quellen nötig.

3.1.5 Importkomponente

Die beiden Importkomponenten laden Daten von der entsprechenden API und konvertieren diese in das in Unterabschnitt 4.2.1 definierte Datenschema. Diese Komponenten können nicht skalieren und es darf in jedem Fall nur eine Komponente pro API aktiv sein. Das Mapping System und die Beziehungen der Daten, besonders bei den Echtzeitdaten, verhindern eine effiziente, parallele Verarbeitung.

3.1.6 Anzeigekomponente

Die Anzeigekomponente ist sehr leichtgewichtig, um die Daten schnell und effizient an den Browser zu liefern. In dieser Komponente werden die erforderlichen Daten zusammengesetzt und in DTOs konvertiert. Um diese Komponente so wenig wie möglich zu belasten, wird eine Client-seitige Template-Engine verwendet. Die View ändert sich selten, daher ist es sinnvoll, das Zusammensetzen des HTML-DOM dem Browser zu überlassen und die Anzeigekomponente nur nicht unnötig zu belasten.

3.1.7 Skalierbarkeit

Eine Anforderung der Anwendung ist eine gute Skalierung der Komponenten. Die Fahrplan-Datenbank ist eine zentrale Komponente, es muss deshalb sichergestellt sein, dass diese horizontal sowie vertikal skaliert. Ein Teil der Logik wird somit mittels SQL-Abfragen an die Datenbank übertragen, da diese gut mit grossen Datenmengen umgehen kann. Die Anzeigekomponente hat keinen Zustand, jede Abfrage ist nach deren Bearbeitung irrelevant für alle folgenden Abfragen. Somit kann diese Komponente beliebig vertikal und auch horizontal skaliert werden.

3.1.8 Cloudfähigkeit

Die gesamte Anwendung soll in einer Cloud gehostet werden können. Durch die Aufteilung der Anwendung in verschiedene Module, kann jeweils für jedes Modul das passende Cloud Angebot verwendet werden.

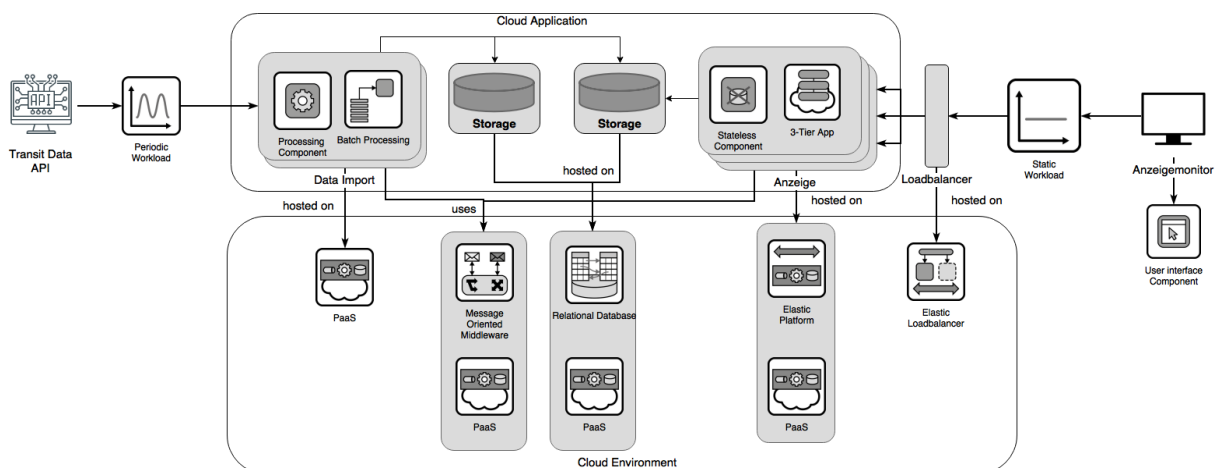


Abbildung 3.4: Cloud Architektur des Prototyps

Alle Module können in einem Docker Container verwendet werden. Das garantiert eine sehr hohe Portabilität, da fast jeder Cloud Anbieter verwendet werden kann, welcher Docker als Execution Environment anbietet. Somit ist es möglich, alle Module in einem PaaS, IaaS oder in einer eigenen Umgebung hosten zu können.

Data Import Der Import der Daten verwendet eine API, welche die Daten liefert. Beide Import Komponenten verursachen eine periodische Workload für die Umgebung. Im Prototyp sind zwei PaaS Instanzen nötig, um statische und Echtzeit Daten importieren zu können.

Storage Beide Datenbanken sind relational, und können in einer Cloud als PaaS gehostet werden.

Messaging Zur Kommunikation zwischen Display und Import wird eine Message Oriented Middleware verwendet. Wenn der Cloud Anbieter eine geeignete Lösung, welche mit RabbitMQ kompatibel ist, anbietet, kann diese Komponente mittels PaaS abgedeckt werden.

Display Die Anzeige der Daten kann ebenfalls mittels Docker Container gehostet werden. Der Container kann in einer elastischen Plattform skaliert und beispielsweise hinter einen Load Balancer geschaltet werden.

3.1.9 Fazit

Die entworfene Architektur lagert viel Arbeit und Logik an die Datenbank und die Importkomponenten aus. Somit skaliert die Anzeigekomponente sehr gut und ist auch mit wenig verfügbaren Ressourcen schnell. Der Datenbank muss aber genügend Leistung zur Verfügung stehen, um die skalierende Anzeigekomponente mit Daten beliefern zu können. Sowie auch die beiden Importkomponenten benötigen genügend Leistung um schnell und effizient Daten in die Datenbank zu schreiben. Da die Datenbank zu einem Single Point of Failure wird, muss sichergestellt sein, dass diese genügend gegen Ausfälle oder Engpässe abgesichert ist. Es ist ausserdem nötig, für jede weitere Datenquelle zusätzliche Instanzen der Importkomponenten zu verwenden. Das aktuelle Design sieht keine Konsolidierung verschiedener Datenquellen, welche im selben Format vorliegen, vor. Zudem benötigt es für jedes weitere Datenformat eine neue Implementation der Importkomponenten.

3.2 Datenmodell

Die Anwendung soll Daten unabhängig von Standards anzeigen können. Die Daten müssen deswegen in einem klar definierten Format vorliegen. Das gewählte Schema beinhaltet alle Daten, welche für die Anzeige von Abfahrten nötig ist und verhindert durch Abstraktion, dass Daten redundant abgespeichert werden müssen.

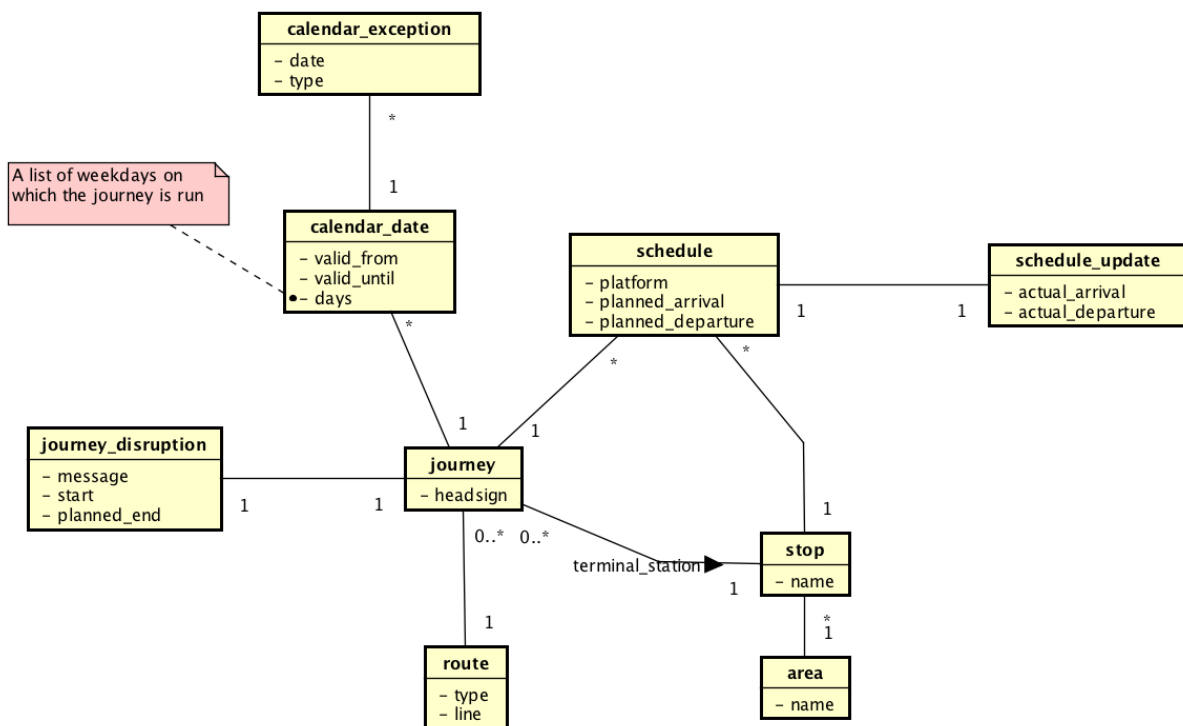


Abbildung 3.5: Datenmodell des Prototyps

Die in Abbildung 3.5 definierte Domäne, wird im folgenden Abschnitt genauer erläutert.

Route Die Route beschreibt eine bestimmte Strecke eines Verkehrsmittels und entspricht einer Linie. Fährt die selbe Linie verschiedene Strecken, wie es oft bei Bussen vorkommt, entspricht jede Strecke einer Route.

- **Type:** Das Verkehrsmittel welches auf der Route verkehrt, wie beispielsweise ein Bus, Zug oder Schiff etc.
- **Line:** Die Linienbezeichnung der Strecke.

Area Eine Area beschreibt einen Bereich, ein sogenannter Meta-Bahnhof. Dieser Bereich beinhaltet mehrere Haltestellen, wie beispielsweise die Perrons verschiedener Zuggesellschaften oder Bushaltestellen. An einem Hauptbahnhof hat man typischerweise mehrere Haltestellen, welche jedoch logisch alle demselben Ort entsprechen.

- **Name:** Der Name des Bereichs, beispielsweise der Bahnhofsname.

Stop Eine kleinere Menge von Haltestellen. An kleinen Bahnhöfen entsprechen die Area und der Stop demselben Bereich. In grossen Bahnhöfen wäre ein Stop beispielsweise die Perrons einer Bahngesellschaft, ein anderer der Busbahnhof.

- **Name:** Der offizielle Name des Bahnhofs oder der Sammlung von Haltestellen.

Journey Entspricht einer Durchführung einer bestimmten Route. Meist ist dies die Fahrt eines bestimmten Fahrzeuges. Eine Journey wäre beispielsweise der 10:32 Uhr IC, am 30. Mai 2017, von Zürich nach Genf.

- **Headsign:** Die Zielanzeige auf dem Fahrzeug, falls vorhanden.

Journey Disruption Eine Störung einer Journey, welche zu Verspätung oder Ausfall der Journey führt.

- **Message:** Die Meldung, welche die Störung genauer beschreibt.
- **Start:** Ab wann die Störung aufgetreten ist.
- **Planned End:** Wann die Störung behoben sein sollte.

Calendar Date Definiert die Durchführung eines Journeys an bestimmten Wochentagen. Wird eine Journey beispielsweise nur an Wochentagen oder während eines bestimmten Ereignisses durchgeführt, kann dies mithilfe dieser Eigenschaft definiert werden.

- **Valid From:** Ab wann eine Definition gültig ist. Wird eine Journey regelmässig durchgeführt, ist dies meist der Beginn des Fahrplans.
- **Valid Until:** Bis wann die Definition gültig ist. Bei Journeys, welche nur an bestimmten Tagen angeboten werden (beispielsweise während der Urlaubssaison), definiert dieser Wert das Ende dieser Periode.
- **Days:** Eine Liste aus Wochentagen, an denen der Journey in dem definierten Zeitfenster durchgeführt wird.

Schedule Ein geplanter Stop an einer bestimmten Haltestelle eines bestimmten Fahrzeugs. Dies könnte beispielsweise der Halt in Bern des 10:32 Uhr IC von Zürich nach Genf an einem bestimmten Datum sein.

- **Platform:** Die Bezeichnung des Haltepunktes, an dem das Fahrzeug hält. Dies sollte dem Gleis, der Kante oder ähnlichem entsprechen.
- **Planned Arrival:** Die geplante Ankunft eines Fahrzeugs an einem Stop.
- **Planned Departure:** Die geplante Abfahrt eines Fahrzeugs an einem Stop.

Schedule Update Die prognostizierte Ankunft und Abfahrt eines Fahrzeuges an einem Stop.

- **Actual Arrival:** Die prognostizierte Ankunft eines Fahrzeuges an einem Stop.
- **Actual Departure:** Die prognostizierte Abfahrt eines Fahrzeuges an einem Stop.

Calendar Exception Definiert geplante Änderungen im Fahrplan. Wenn beispielsweise eine Linie aufgrund von Baustellen nicht verkehrt.

- **Date:** Das Datum der Änderung.
- **Type:** Definiert was passiert. Wird die Linie beispielsweise aufgehoben, werden Journeys hinzugefügt etc.

Teil III

Umsetzung

Kapitel 4

Implementation

In diesem Kapitel werden die Resultate des Prototyps und die Probleme, welche bei der Entwicklung aufgetreten sind beschrieben.

4.1 Infrastruktur

In diesem Abschnitt wird die Infrastruktur beschrieben, welche während des Entwicklungsprozesses für die Demonstration wie auch für die Entwicklung selbst verwendet wurde.

4.1.1 Amazon AWS

Die Cloud von Amazon bietet viele Services, welche das Betreiben einer Applikation vereinfacht. Für eine Datenbank bietet AWS diverse Produkte an, welche das Einrichten und die Verwaltung eines DBMS vereinfacht.

Amazon RDS mit PostgreSQL Amazon RDS sind Datenbankinstanzen eines bestimmten Herstellers. Für diese Arbeit wurde eine PostgreSQL Instanz gewählt. PostgreSQL ist einerseits im Geschäftsumfeld sehr beliebt und implementiert den SQL-Standard andererseits sehr genau. Die Wahl des Datenbankherstellers beeinflusst das Resultat der Fragestellung kaum, da jedes auf Amazon angebotene Datenbanksystem in der Performance vergleichbar ist.

EC2 Registry In der Docker-Terminologie wird die Ablage von Docker-Images „Registry“ genannt. Amazon bietet ein solches Registry an, um die Images der Anwendung dort hochzuladen, wo dann von den Host-Systemen direkt darauf zugegriffen werden kann und der neuste Stand der Anwendung ausgerollt wird.

Elastic Beanstalk Dieser Service ist das PaaS Angebot von Amazon. Es können Anwendungen verschiedener Technologien hochgeladen und ausgeführt werden. Für das Ausrollen einer Docker-Anwendung sind zwei Schritte nötig:

1. Die Anwendung wird nach dem Kompilieren und Testen mit den entsprechenden Einstellungen und Parametern in ein Docker Image geschrieben. Dieses wird in die Registry hochgeladen.
2. Nach dem Hochladen des Docker Images, wird der Elastic Beanstalk Service von Amazon benachrichtigt, worauf dieser das neue Image herunterlädt und startet.

Docker

Jedes der entwickelten Module wird in einem Docker Container verwendet. Als Basis Container wird das offizielle Image von OpenJDK mit der Java Runtime 1.8 verwendet (https://hub.docker.com/_/openjdk/). Für das Anzeigemodul wird die Linux Alpine Version des Images verwendet, da dieses sehr klein ist. Für die Importmodule muss leider das Standardimage verwendet werden, welches auf Debian basiert. Grund dafür ist, dass beide Module SQLite verwenden und es ein bekanntes Problem mit Alpine und der neusten SQLite Version gibt.

4.2 Prototyp

Der Prototyp der Komponenten wird mittels Abfahrtsanzeiger realisiert. Auf diesem Anzeiger sollen folgende Punkte ersichtlich sein:

- Eine Linienbezeichnung: Eine Busliniennummer oder S-Bahn Linie etc.
- Ziel der Abfahrt: Falls ein Fahrzeug eine Zielanzeige hat, sollten diese mit der Anzeige übereinstimmen.
- Kante oder das Gleis der Abfahrt.
- Geplante Abfahrt in Stunden und Minuten.
- Falls eine Abfahrt eine Verspätung aufweist, so soll diese auch in Minuten angezeigt werden.

4.2.1 Fahrplandatenbank

Die primäre Datenbank beinhaltet die gesamten Fahrplan-Informationen, welche für die Anzeige relevant sind.

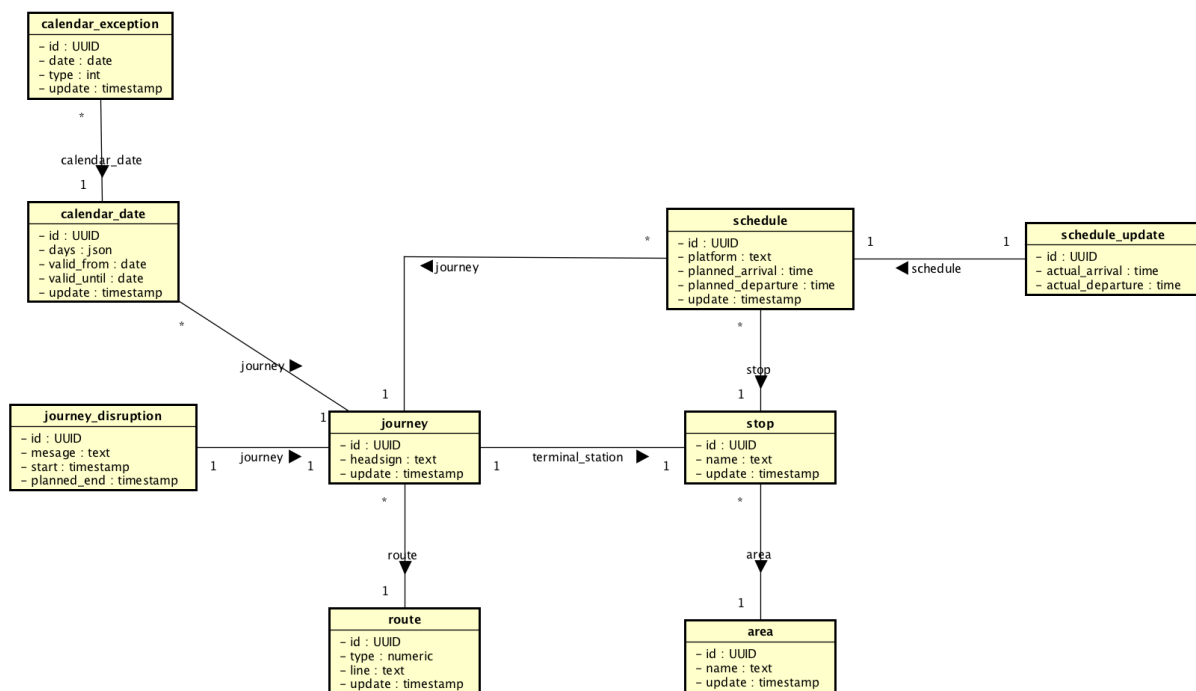


Abbildung 4.1: Schema der Fahrplan Datenbank

Der Prototyp verwendet PostgreSQL als Datenbank, weshalb die Typen denen aus dem Datenbanksystem entsprechen. Sämtliche Beziehungen zwischen Entitäts werden über die ID aufgelöst, für diese werden UUIDs verwendet. Zudem haben alle Entitäts ein Feld des Update-Timestamps, welcher die Version der Daten angibt.

Area

- **name:** Der Name der Area.

Route

- **type:** Der Typ des Verkehrsmittels als numerischer Wert. Diese wurden aus der GTFS Spezifikation übernommen [7], es wurden folgende Werte definiert:

- 0: Tram, Strassenbahn oder ähnliche Fahrzeuge, welche auf der Strasse verkehren.
 - 1: U-Bahn und Züge, welche unter der Erde verkehren.
 - 2: S-Bahn, Regional- oder Intercity-Züge. Nah- und Fernverkehr auf der Schiene.
 - 3: Stadtbuss, Fern Bus oder ähnliche Fahrzeuge.
 - 4: Fähre oder Schifffahrten über kurze oder lange Distanzen.
 - 5: Standseilbahn oder ähnliche Fahrzeuge, welche von einem Seil unter dem Fahrzeug angetrieben werden.
 - 6: Luftseilbahn oder hängende Seilbahnen.
 - 7: Standseilbahn, Zahnradbahn, Fahrzeuge auf Schienen, welche auf grossen Steigungen verkehren.
- **line:** Die Linienbezeichnung der Route. Dieser Text wird im Anzeigemodul als Linie angezeigt.

Stop

- **name:** Der Name des Stopps.

Journey

- **headsign:** Der Text auf der Zielanzeige des Fahrzeugs. Dieser Text wird im Anzeigemodul als Ziel der Abfahrt angezeigt.

Journey Disruption

- **message:** Die Nachricht welche die Störung möglichst genau beschreibt.
- **start:** Seit wann die ungeplante Störung auftritt.
- **planned_end:** Bis wann die ungeplante Störung ungefähr andauert.

Schedule

- **platform:** Die Kante oder das Gleis, an dem das Fahrzeug hält. Falls dieser Schedule keine Kante oder Gleis hat, wird der Wert „0“ erwartet.
- **planned_arrival:** Die geplante Ankunftszeit des Fahrzeugs.
- **planned_departure:** Die geplante Abfahrtszeit des Fahrzeugs. Die Stunde und Minute dieses Werts werden im Anzeigemodul als geplante Abfahrt angezeigt.

Schedule Update

- **actual_arrival:** Die effektive oder prognostizierte Ankunft des Fahrzeugs.
- **actual_departure:** Die effektive oder prognostizierte Abfahrt des Fahrzeugs. Die Differenz zu der geplanten Abfahrtszeit wird in Minuten angezeigt.

Calendar Date

- **days:** Die Tage, an denen die Journey durchgeführt wird, in einem JSON Array. Eine Journey, die beispielsweise von Montag bis Freitag durchgeführt wird, sieht wie folgt aus:

```
{ "service_days": ["MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY"] }
```

- **valid_from:** Der Start der Gültigkeit des Calendar Date.
- **valid_until:** Das Ende der Gültigkeit des Calendar Date.

Calendar Exception

- **date:** An welchem Datum die Ausnahme im Fahrplan zutrifft.
- **type:** Definiert um was für eine Ausnahme es sich handelt. Es sind folgende Werte definiert:
 - 1: Eine Durchführung wurde hinzugefügt.
 - 2: Eine Durchführung wurde entfernt.

4.2.2 Mapping Datenbank

Durch die Verwendung des eigenen ID Systems muss eine konsistente Navigation zwischen den IDs der Datenquelle und der Fahrplandatenbank möglich sein. Diese Daten werden in einer Mapping Datenbank gespeichert. Es ist vorgesehen, dass jede Datenquelle eine eigene Mapping Datenbank verwendet, um das Kollisionsproblem zwischen den IDs auszuschliessen. Das Schema besteht, mit einer kleinen Ausnahme, aus Tabellen mit einer Fremd-ID und einer Prototyp-ID. Auch diese Daten besitzen jeweils einen Update Timestamp, um das Mapping einer Import-Version zuordnen zu können.

Journey Mapping

- **gtfs_trip_id:** Die ID des GTFS Trips.
- **gtfs_service_id:** Die Service ID, welche der entsprechende GTFS Trip besitzt, um das GTFS Calendar Date zu referenzieren.
- **id:** Die ID des Journeys zu welchen die Daten aus GTFS Trip und GTFS Route zusammengesetzt ist.

Route Mapping

- **gtfs_id:** Die ID der GTFS Route.
- **id:** Die ID zu welcher die GTFS Route konvertiert wurde.

Stop Mapping

- **gtfs_id:** Die ID des GTFS Stops welcher ein „parent_stop“ besitzt.
- **id:** Die ID zu welcher der GTFS Stop konvertiert wurde.

Calendar Date Mapping

- **gtfs_id:** Die ID des GTFS Calendar Date.
- **id:** Die ID zu welcher die GTFS Calendar Date konvertiert wurde.

Area Mapping

- **gtfs_id:** Die ID des GTFS Stop welcher kein „parent_stop“ besitzt.
- **id:** Die ID zu welcher die GTFS Route konvertiert wurde.

4.2.3 Static Import

Der Static Import verarbeitet die statischen GTFS Daten. Diese werden von der API als ZIP-Datei angeboten. Der Import lädt die Datei herunter, entpackt sie und verarbeitet jede darin enthaltene CSV-Datei. Das Resultat dieser Verarbeitung wird in die Fahrplan- und Mappingdatenbank geschrieben und kann von den Anzeigemodulen gelesen werden.

Diese Importkomponente verwendet das Spring Batch Framework. Dieses Framework implementiert den JSR 325 Standard [21] und verarbeitet Daten in fest definierten Schritten.

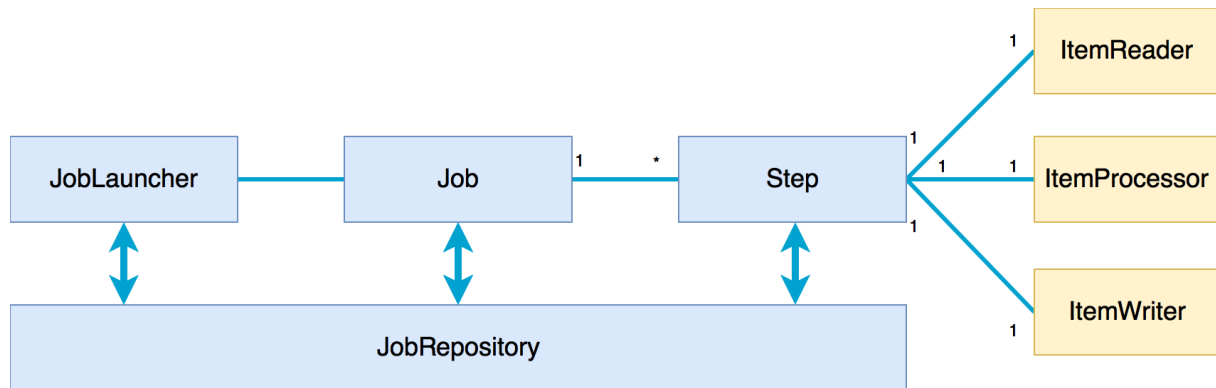


Abbildung 4.2: Referenzmodell von Spring Batch

Im Static Import ist der gesamte Import der statischen Daten ein Job. Jedes Entity in der Fahrplan Datenbank wird durch einen Step verarbeitet. Die einzelnen CSV Dateien werden von Item Readern in Java Objekte gelesen und dann einem Item Processor übergeben, welcher die GTFS Daten in das Schema des Prototyps konvertiert und anschliessend durch einen Item-Writer in die Fahrplan Datenbank geschrieben. Während diesen Steps werden MappingStore Objekte mit den Mapping Informationen gefüllt. Ein MappingStore speichert alle diese Informationen in einer Java Map. Nach der erfolgreichen Verarbeitung und das Persistieren der Fahrplandaten, werden diese Informationen aus der Map gelesen und in die Mapping Datenbank geschrieben.

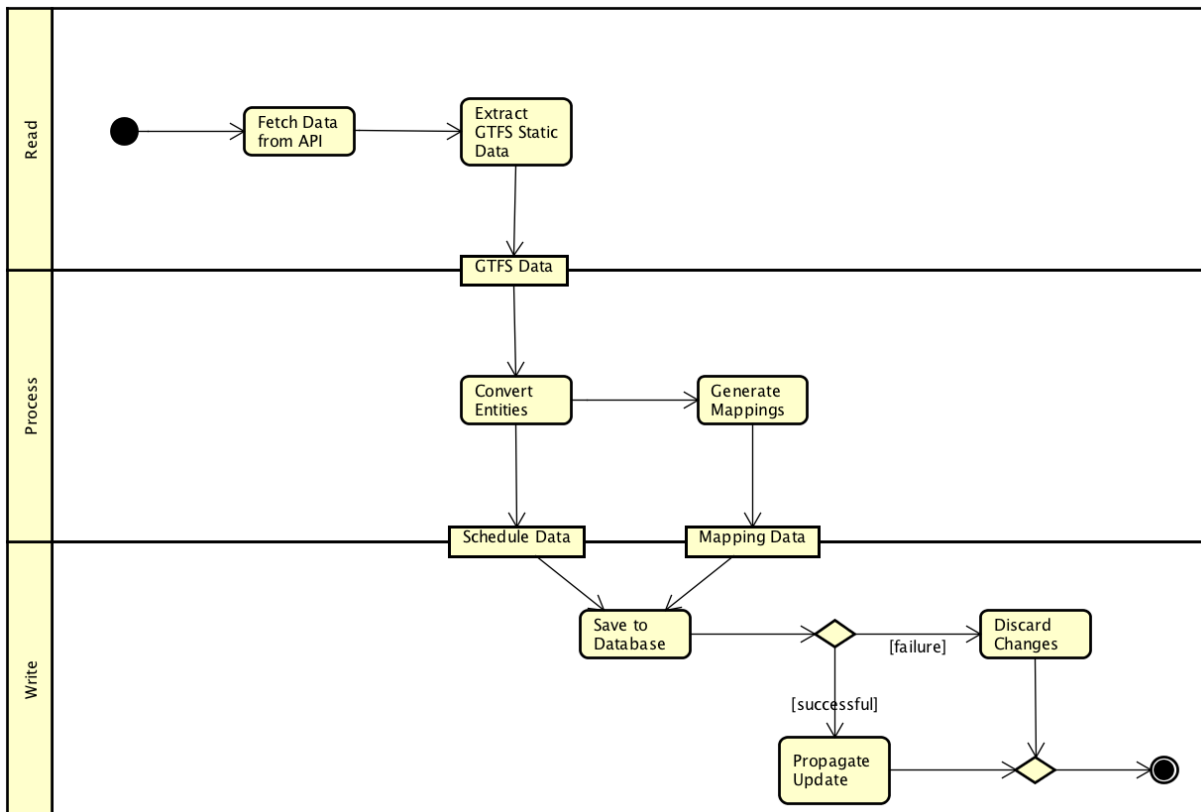


Abbildung 4.3: Aktivitätsdiagramm des Static Import

Die einzelnen Schritte aus Abbildung 4.3 sind in folgenden Abschnitten zusätzlich erklärt.

Fetch Data from API Die Daten werden in einem definierten Intervall heruntergeladen und eingelesen. Die ZIP-Datei wird auf dem Heap abgelegt, die Daten werden dort entpackt und auf die Festplatte geschrieben. Das Framework bietet leider keine Möglichkeit, CSV Dateien direkt aus dem Heap zu lesen, beispielsweise als `InputStream`, weshalb dieser Zwischenschritt nötig ist.

Extract GTFS Static Data Die einzelnen Dateien werden vom Framework eingelesen und in Java Objekte konvertiert. Diese werden den jeweiligen `ItemProcessor` Instanzen übergeben und werden dann in die Objekte aus dem Prototypschema konvertiert. Die API des Auftraggebers beinhaltet folgende Dateien:

- **trips.txt**
- transfers.txt
- **stop_times.txt**
- **stops.txt**
- **routes.txt**
- feed_info.txt
- **calendar_dates.txt**
- agency.txt

Für die Anzeige der Daten sind jedoch nur die fett markierten Dateien interessant. Aufgrund der Relationen in den GTFS Daten muss eine bestimmte Reihenfolge eingehalten werden. Diese Reihenfolge richtet sich nach der Navigation in den GTFS Daten. Als erstes müssen die GTFS Entities eingelesen

werden, welche keine relevanten Fremdschlüsselattribute haben. Daraus ergibt sich folgende Reihenfolge:

1. Areas
2. Stops
3. Routes
4. Journeys
5. Calendar Infos
6. Schedules

Convert Entities Im diesem Schritt werden die GTFS Entitäts an die passenden `ItemProcessor` Instanzen übergeben, welche die nötigen Daten daraus extrahieren und die relevanten Daten in die `MapperStore` Objekte speichern. In diesem Schritt muss besonders auf die Konsistenz der Daten geachtet werden. Es ist deshalb sehr wichtig, dass während der Konvertierung möglichst wenig Fehler auftreten. Für den Fehlerfall müssen definierte Daten oder Abläufe vorhanden sein.

Generate Mappings Während die Entitäts konvertiert werden, werden auch die Mappings generiert. Für jedes verarbeitete Objekt werden diese Informationen abgespeichert.

Save to Database Nach erfolgreichem Konvertieren können die Daten in die Fahrplan Datenbank gespeichert werden. Nach diesem Schritt werden die Mapping Daten in die Datenbank geschrieben und der Import Job ist abgeschlossen.

Propagate Changes Nach erfolgreichem Abschluss des Jobs werden alle laufenden Anzeigekomponenten benachrichtigt und können auf die aktuellste Version der Daten wechseln.

Skalierung

Im Prototyp darf nur eine Instanz, welche statische Daten importiert, laufen. Durch die Versionierung der Daten können bei einem parallelem Import von Daten Inkonsistenzen zwischen gemeinsamen Datensätzen entstehen; so zum Beispiel bei Bahnhöfen, welche von mehreren Datenquellen erfasst werden. Um diese Unstimmigkeit zu vermeiden, werden die Zustände der Instanzen in der Datenbank gespeichert, wie in Abschnitt 4.3 beschrieben.

Probleme

Hibernate Zu Beginn des Import Prototyps ist Hibernate verwendet worden, um die Daten zu persistieren. So konnte schnell evaluiert werden, ob das Framework für diese Komponente verwendet werden könnte. Wie in Abschnitt 2.2 beschrieben ist Hibernate nicht die erste Wahl, wenn die Performance im Vordergrund steht. Durch die Migration der Struktur auf JDBC basierte Verbindungen mit nativen Queries konnte die Zeit des Importierens von 40 auf vier Minuten reduziert werden (ohne Speichern der Mappings).

Verschiedene Verwendungen von Calendar Der GTFS Standard sieht zwei verschiedene Anwendungen der `Calendar` vor, welche beschreiben, an welchen Tagen oder Daten ein Trip durchgeführt wird. Bevorzugte Lösung ist es, eine `Calendar` Entität zwischen `CalendarDate` und `Trip` zu verwenden, welche mittels `Service ID` eine Beziehung zwischen den drei Entitäten herstellt. Diese Entität beinhaltet die Wochentage, an denen ein Trip durchgeführt wird. Mit der `Calendar Date` Entität können Ausnahmen zu diesen Daten eingefügt werden. Eine andere Möglichkeit ist es, die `Calendar` Entität wegzulassen und alle Durchführungen eines Trips als Ausnahme zu modellieren. Vorteil dieser Struktur ist eine geringere Datenmenge, wenn ein Verkehrsbetrieb viele unregelmässige Durchführungen eines Trips anbietet. Leider entsteht bei dieser Anwendung des Standards eine $N:M$ Beziehung

zwischen Trip und CalendarDate. Jeder Trip kann als Folge dessen nur zu bestimmten Daten durchgeführt und nicht mehr direkt einem Wochentakt zugeordnet werden. Es ist somit nicht nur aufwändiger, verschiedene Trips zu verschiedenen Durchführungen zuordnen zu können, es verursacht in einem Taktfahrplan, wie den des Auftraggebers, zu sehr grossen Mengen an Daten, welche über eine Calendar Beziehung deutlich reduziert werden kann.

4.2.4 Realtime Import

Das Realtime Importmodul ist für das Importieren der GTFS-Realtime Informationen verantwortlich, welche im binären Format zur Verfügung gestellt werden. Dabei ist zu beachten, dass die Datenquelle der Realtime-Updates auf die Datenquelle der Static-Updates abgestimmt sein muss, damit Echtzeitinformatoren richtig verknüpft werden können. Die nachstehende Grafik bietet einen Überblick für den Ablauf eines Realtime-Updates.

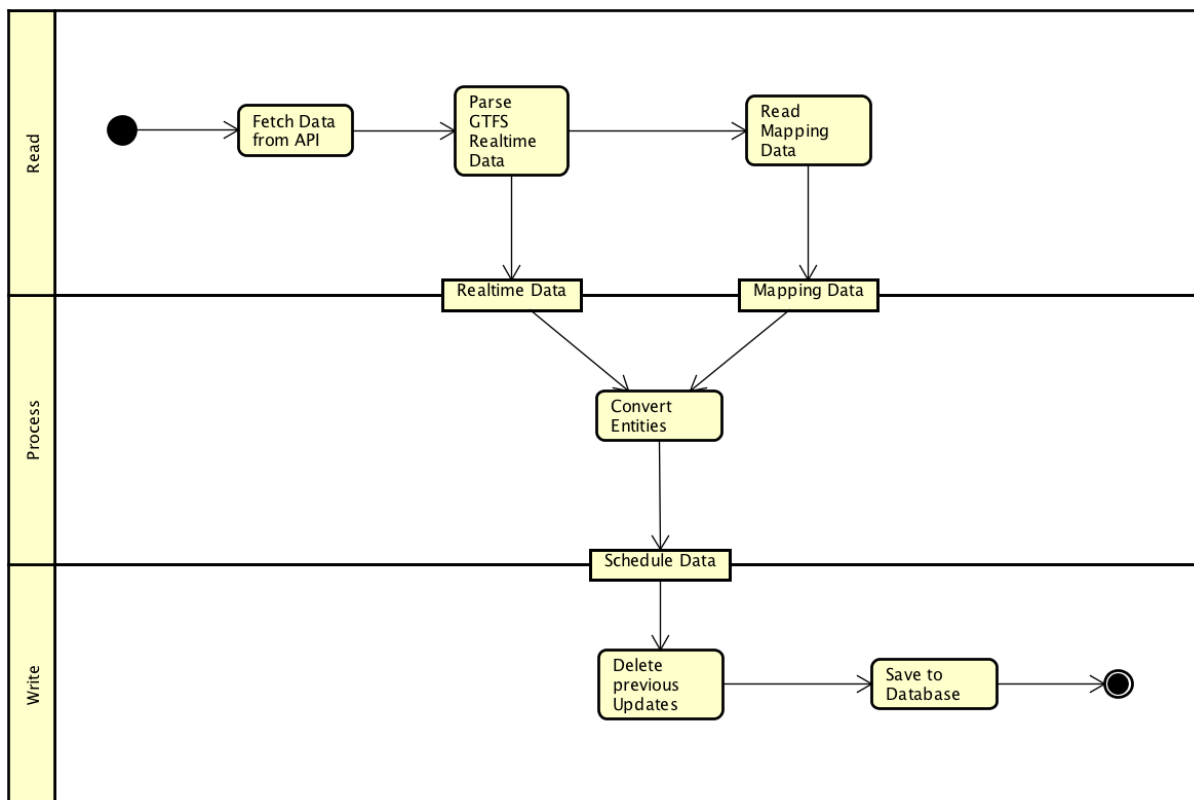


Abbildung 4.4: Aktivitätsdiagramm des Realtime Import

Fetch Data from API Für diesen Schritt gibt es grundsätzlich zwei Techniken welche in Betracht gezogen werden: „Pull“ oder „Push“. Da die „Push“-Methode nicht im GTFS-Standard vorgesehen ist, wurde in dieser Arbeit der „Pull“-Ansatz gewählt.

Parse GTFS Realtime Data Damit die Echtzeitinformatoren verarbeitet werden können, müssen diese aus dem binären Protocol Buffers Format konvertiert werden. Um auf die Daten zuzugreifen, wird eine Java-Library für die Google Transit API verwendet. Nach dem Parsen stehen alle Echtzeitinformatoren als Java Objekte zur Verfügung.

Convert Entities & Read Mapping Data Nachdem die Echtzeitinformatoren heruntergeladen und aus dem binären Format konvertiert wurden, müssen diese in die Fahrplandatenbank eingespielt werden. Dafür müssen die relevanten Informationen extrahiert und in das eigene Datenmodell importiert werden. Die grösste Herausforderung dabei ist es, die Echtzeitinformatoren den Abfahrten

zuzuordnen. Gemäss GTFS-Standard geht das einfach über eine gemeinsame ID. Durch die Verwendung der eigenen IDs sind zur erstellen der Relation die Mapping Informationen nötig, um eine eindeutige Zuordnung zu gewinnen. Leider gibt es bei dieser Lösung einen Haken: Die Performance. Um die Beziehung für ein Update herzustellen, sind mehrere Abfragen auf die Mapping Daten notwendig. Da die Realtime-Updates sehr zeitkritisch und die vielen Datenbankabfragen langsam sind, stellt dies ein Problem dar. Die anfängliche Lösung benötigte für ein Realtime-Update auf den Entwicklungsrechnern ca. drei Minuten. Der Flaschenhals befand sich bei den vielen Datenbankabfragen und den damit zusammenhängenden Datenbank-Verbindungen. Bei der Entwicklungs-Arbeitsumgebung war die Datenbank lokal, weshalb der Aufwand des Verbindungsaufbaus gering war. Bei einem verteilten System, wie das Deployment in der AWS-Cloud, ist dieser aufwändiger und stellt eine Schwachstelle im Prozess dar. Eine mögliche Lösung für dieses Problem besteht darin, alle Mapping Informationen zu Beginn zu laden. Dies verringert die Anzahl Datenbankverbindungen. Der Nachteil dieser Lösung bildet die vergrösserte Payload, da auch nicht verwendete Mapping Informationen zwischengespeichert werden. Eine Datenbankabfrage auf eine gesamte Tabelle ist jedoch sehr schnell und hat sich für dieses Problem als gute Lösung erwiesen. Sind die Mapping-Informationen geladen und die Echtzeitinformationen geparsed, sind alle nötigen Informationen für einen Import der Realtime Updates vorhanden. Für jedes importierte Update wird ein Meta-Objekt erstellt, welches durch den Prozess des Imports mit Informationen angereichert wird. Wenn eine notwendige Information für dieses Objekt nicht zur Verfügung steht ist eine Zuordnung zum statischen Fahrplan nicht möglich. In diesem Fall wird das Update ignoriert. Das Realtime-Update konnte mit der eben beschriebenen Architektur auf ca. zehn Sekunden beschleunigt werden.

Save to Database Nach der erfolgreichen Konvertierung werden alle Updates in die Datenbank gespeichert. Im Gegensatz zum Static Update müssen die Realtime Updates können einzeln eingespielt werden, ohne einen Transaktionsmechanismus. Sie sind dabei ab ihrer Präsenz in der Datenbank sofort gültig und brauchen nicht mit einem Timestamp markiert zu werden. Ein vorhandenes Update ist ein gültiges Update. Beim Speichern der Updates stellte sich die Frage, wie man diese in das Datenbank Modell integrieren kann. Für diese Arbeit wurden zwei Möglichkeiten in Betracht gezogen: Die erste sieht vor einen Fremdschlüssel beim entsprechenden `Schedule`-Eintrag in der `Schedule`-Tabelle zu speichern, welcher das dazugehörige Realtime-Update referenziert. Bei der Umsetzung dieser Lösung gibt es zwei Schwachstellen: Als erstes ist mit dem Speichern des Updates eine zusätzliche Operation auf der Datenbank notwendig, um das Update gültig zu machen. Das zweite Problem tritt beim Aufräumen der Updates auf. Möchte man beispielsweise alle Realtime Updates löschen, so müssen alle Referenzen wieder entfernt werden, was je nach Anzahl `Schedules` aufwändig sein kann. Aufgrund dieser Schwachstellen wird eine zweite Lösung in Betracht gezogen: Die Referenzen werden umgekehrt, ein Realtime Update hat eine Referenz auf eine `Schedule`. Dies vereinfacht sowohl das Speichern als auch das Löschen eines Updates. Das Update ist zudem nach einer Datenbank Operation gültig, was dessen tatsächliche Einspielung nochmals beschleunigt.

4.2.5 Anzeige Modul

Der Abfahrtsanzeiger ist mit Spring Boot und dem Spring Framework implementiert. Dabei verwendet die Web-Anwendung den vom Framework eingebetteten Tomcat-Server. Die Aufgabe des Abfahrtsanzeigers ist es, die Fahrplaninformationen der Fahrplandatenbank dem Endbenutzer zu präsentieren. Die Anzeige wird dabei in einem Intervall von zehn Sekunden aktualisiert. Der Abfahrtsanzeiger ist auf der klassischen 3-Layer Architektur mit dem Data-Access-, Business- und Presentation-Layer aufgebaut.

Data Access Layer Um auf die Fahrplandatenbank zuzugreifen, wird der OR-Mapper Hibernate verwendet. Auch wenn Hibernate nicht in der Performance glänzen kann, ermöglicht es einen einfachen Zugriff auf die Datenbank und kümmert sich um das Nachladen von Referenzen, was für die Implementation des Abfahrtsanzeigers sehr praktisch ist.

Business Layer Für den Business Layer wurden neben der Dependency Injection von Spring keine weiteren Framework-spezifische Technologien eingesetzt, da dessen Funktionalität überschaubar ist. Der Business Layer nimmt die Daten des Data Access Layer entgegen und passt diese je nach Be-

darf minimal an. Aufwändige Operationen sind bewusst vermieden worden, um den Abfahrtsanzeiger leichtgewichtig und somit performant zu gestalten.

Presentation Layer Der Presentation Layer präsentiert dem Endbenutzer die vom Business Layer aufbereiteten Daten. Dieser Layer wurde mit Spring Web Framework und dem bekannten MVC-Pattern aufgebaut. Die Controller bilden dabei die externe Schnittstelle, in dem sie Requests entgegennehmen, diese verarbeiten und die zugehörige Response generieren. Die generierte Response kann entweder im HTML- oder im JSON-Format strukturiert sein. Die HTML-Response wird durch die Template Engine „Thymeleaf“ generiert. Die JSON-Response wird hingegen aus den DTOs erzeugt.

Wie in Abbildung 4.5 zu erkennen ist, verwendet der Abfahrtsanzeiger eine Kombination von beiden Verfahren, in dem es anfänglich ein durch „Thymeleaf“ generiertes HTML ausliefert, welches anschliessend durch AJAX-Requests bei der JSON-Schnittstelle die anzuzeigenden Daten herunterlädt.

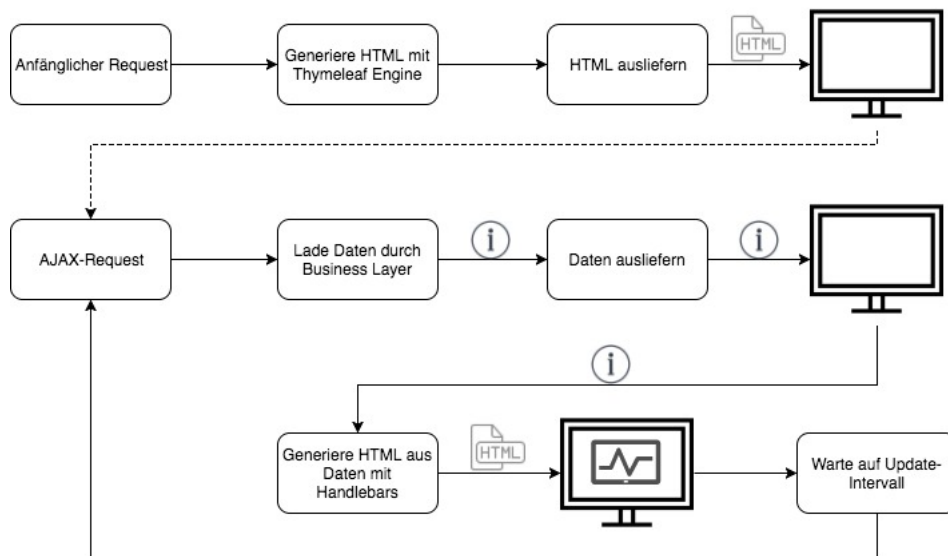


Abbildung 4.5: Arbeitsablauf einer Anzeige

Bei einem erfolgreichen AJAX-Request werden die Daten durch die Frontend Template Engine „Handlebars“ in HTML kompiliert und angezeigt. Diese Methodik die Daten anzuzeigen bietet einige Vorteile. Zum einen wird der Abfahrtsanzeiger durch eine DOM-Manipulation aktualisiert, wodurch ein „flackern“, welches beim Neu laden einer Website entsteht, verhindert wird. Zum anderen kann der Server ausfallen, ohne dass dies im ersten Moment dem Endbenutzer ersichtlich wird, indem fehlgeschlagene AJAX-Requests ignoriert werden. Ausserdem kann die JSON-Schnittstelle auch für weitere Zwecke als dessen Anzeige verwendet werden.

Der Abfahrtsanzeiger zeigt die nächsten Verbindungen ab einem Stopp an. Dabei werden nur gültige und aktuelle Verbindungen angezeigt. Wenn zu den ausgegebenen Verbindungen Echtzeitinformationen vorhanden sind, werden diese ebenfalls angezeigt. Eine Abfahrtsanzeige sieht wie folgt aus:

Abfahrten von Bern Wankdorf, Bahnhof

Linie	Ziel	Kante / Perron	Abfahrt	Hinweis
20	Bern, Bahnhof		16:37	OK
9	Wabern, Tram-Endstation		16:39	OK
20	Bern, Bahnhof		16:41	OK
20	Bern, Bahnhof		16:45	OK
9	Wabern, Tram-Endstation		16:45	OK
28	Bern, Eigerplatz		16:48	OK
20	Bern, Bahnhof		16:49	OK
9	Wabern, Tram-Endstation		16:51	OK
20	Bern, Bahnhof		16:53	OK
20	Bern, Bahnhof		16:57	OK

Abbildung 4.6: Abfahrtsanzeiger

API

Der Abfahrtsanzeiger bietet eine RESTful API an. Für den in diesem Kapitel beschriebenen Abfahrtsanzeiger wurden folgende Schnittstellen implementiert:

- GET `/api/departures/{stopId}`
- GET `/api/departures/{stopId}/at/{time}`
- GET `/departures/{stopId}`

Die ersten zwei Schnittstellen liefern die nächsten zehn Abfahrten ab der Haltestelle mit der entsprechenden `stopId`. Die Anzahl Verbindungen kann durch einen GET-Parameter `size` angepasst werden. Die Daten werden dabei im JSON Format ausgeliefert und beinhalten alle für den Anzeiger notwendigen Informationen. Dies umfasst auch die Verspätungen der Realtime Updates. Die beiden Schnittstellen unterscheiden sich lediglich bei den Parametern. Während bei der zweiten Schnittstelle die Abfahrtszeiten ab einem gewünschten Zeitpunkt angezeigt werden, verwendet die erste Schnittstelle immer die aktuelle Uhrzeit. Die dritte Schnittstelle liefert keine direkten Abfahrtsdaten, sondern eine HTML-Seite, welche die Daten, wie in der oberen Grafik dargestellt, nachlädt und anzeigt.

Um die Schnittstellen nutzen zu können, muss die `stopId` einmalig ausfindig gemacht werden. Um diesen Prozess zu vereinfachen, wird eine weitere Schnittstelle angeboten welche eine Haltestelle nach dessen Bezeichnung sucht und eine Liste von Suchresultate anbietet:

- GET `/search/{stopName}`

Die Suchresultate werden als HTML-Seite ausgeliefert, welche eine Liste von Verlinkungen im Format der dritten Schnittstellendefinition beinhaltet.

Update

Wird ein Static-Update durchgeführt, ändern sich die `stopIds`. Ein Abfahrtsanzeiger, welcher im vordefinierten Intervall die Daten via AJAX-Request nachlädt, verwendet nach einem Update eine alte `stopId`. Für diesen Fall ist ein Update-Mechanismus eingebaut worden. Der Abfahrtsanzeiger kennt die `stopId`'s mehrerer Update-Versionen und kann durch die Update-History sowohl eine alte `stopId` erkennen, als auch die entsprechende neue Version ausfindig machen. Wird eine alte `stopId` erkannt, wird zuerst die neue `stopId` ausfindig gemacht und anschliessend mit dieser weitergearbeitet. In der JSON-Response werden neben den Fahrplandaten auch die verwendete `stopId` mitgeliefert. Diese wird bei jedem AJAX-Request überprüft und die URL des Abfahrtsanzeigers bei Bedarf aktualisiert. Dabei wird die Seite nicht neu geladen, um ein allfälliges „Flackern“ der Anzeige zu vermeiden. Die neue `stopId` wird für zukünftige AJAX-Requests gespeichert und verwendet.

Skalierung

Die Skalierung der Anzeige Komponente ist für die Prototypen eine sehr wichtige Eigenschaft. Spring Boot ist ein sehr leichtes Framework, bietet aber trotzdem alle nötigen Komponenten, um über HTTP kommunizieren zu können. Um dieses Modul optimal skalieren zu können, wurden alle Teile, welche das Framework übernehmen kann, an dieses ausgelagert.

Für das Anzeigen von Abfahrten ist es nicht nötig, Zustände von verschiedenen Anzeigern im Web-Server zu speichern. Jeder Request kann von der Komponente nach dessen Bearbeitung vergessen werden.

4.2.6 Ausfallszenarien

Für die Cloud-Fähigkeit eines Systems sind Ausfallszenarien ein essenzieller Punkt. Für die Elastizität ist es elementar, dass beim Ausfall einer Instanz die Stabilität des Systems nicht gefährdet ist. In dieser Arbeit ist diese Eigenschaft durch die zustandslose Architektur sichergestellt. Die Cloud-Fähigkeit wurde durch den Einsatz der Container-Technologie Docker sichergestellt. Jedes Teilprojekt ist in einem Container gekapselt und wird so als Einheit vom Cloud-Anbieter Amazon verwaltet.

Für eine saubere Handhabung der Ausfallszenarien ist es wichtig den Term „Ausfall“ zu definieren. Mit einem Ausfall wird in diesem Kontext sowohl ein Versagen der (virtuellen) Arbeitsmaschine als auch eine fehlerhafte Ausführung durch einen Logikfehler verstanden.

Amazon AWS

Die PaaS Infrastruktur von Amazon bietet verschiedene Mechanismen, wie fehlgeschlagene Komponenten wieder in einen stabilen Zustand gebracht werden können. Wird Docker verwendet, erkennt die Cloud, wenn die Anwendung abgestürzt ist und nicht mehr antwortet. In diesem Fall stoppt es die aktuelle Instanz und startet automatisch eine neue. Ausserdem erkennt Elastic Beanstalk anhand des HTTP Status Code, wenn die Anwendung fehlerhafte Antworten zurückgibt. Das Verhalten in diesem Fall kann individuell konfiguriert werden. Es können Personen alarmiert werden, oder, ab einem gewissen Schwellwert, die Instanz ausgetauscht werden.

Die IaaS Infrastruktur bietet ähnliche Dienstleistungen auf Basis der Infrastruktur an. In diesem Angebot wird jedoch der Zustand der Infrastruktur, anstelle der Anwendung beobachtet. Es ist leider nicht möglich, Anwendungen innerhalb des IaaS Angebotes zu beobachten und Aktionen bei Fehlerfällen auszuführen.

Static-Import

Der Static-Import sollte nicht mehrfach gestartet werden. Wenn die einzige Instanz des Static-Imports ausfällt, werden die statischen Fahrplaninformationen nicht aktualisiert. Es ist daher wichtig sicherzustellen, dass diese Komponente immer läuft. Der Static-Import ist fehlertolerant aufgebaut. Fehlerhafte Input-Daten werden ignoriert oder führen zu einem Überspringen des kompletten Updates. Die Komponente wird allerdings nie komplett gestoppt, sondern startet gemäss dem voreingestellten Zeitplan im nächsten Intervall einen neuen Import-Versuch. Die gefundenen Fehler werden in einem Log gesammelt und der Abbruch eines Imports zusätzlich in der Datenbank vermerkt. So kann ein

Problem, wie z.B. fehlerhafte Input-Daten (von der Datenquelle) nachvollzogen und für den nächsten Import behoben werden.

Realtime-Import

Der Realtime-Import sollte gleich wie der Static-Import ebenfalls nur einmal ausgeführt werden. Eine mehrfache Ausführung ist aber nicht problematisch und könnte für ein Verzahnen der Update-Intervalle oder aus Gründen der Redundanz interessant sein. Gleich wie der Static Import ist der Realtime Import ebenfalls fehlertolerant gebaut. Bei einem logischen Fehler wird das Update unterbrochen und beim nächsten voreingestellten Intervall ein erneuter Versuch gestartet. Auch hier werden Fehlermeldungen in einem Log gesammelt und Updates ohne Zuordnung zu den statischen Fahrplänen speziell vermerkt, was für die Fehlerbehandlung hilft.

Abfahrtsanzeiger

Der Abfahrtsanzeiger hat keine Abhängigkeit auf die Import-Module und ist somit von dessen Ausfällen nicht betroffen. Wenn es keine Daten zum Anzeigen gibt, so ist der Abfahrtsanzeiger leer.

Kommunikation

Ein Ausfall des Kommunikationsbrokers stört das System nicht. Änderungen in der Fahrplandatenbank werden von dem Anzeiger danach aber nicht mehr erkannt. Die Kommunikation ist im Prototyp lediglich als Proof-of-Concept eingebaut. In einer produktiven Umgebung wäre der Broker hochverfügbar und ausfallsicher.

4.3 Versionierung der Fahrplandaten

Die Fahrplandaten müssen konsistent in die Datenbank gespeichert werden können. Im Fehlerfall würden der Anzeigekomponente Daten fehlen oder es würden falsche Daten angezeigt. Um dies zu verhindern, werden mehrere Versionen aus derselben Datenquelle gespeichert und erst beim erfolgreichen Importieren der Daten alte Versionen gelöscht.

4.3.1 Static-Update

Das Static-Update ist das umfangreichste, da es am meisten Daten enthält. Aus Sicht der Anwendung ist es wichtig alle neuen Daten auf einen Schlag zur Verfügung zu haben. Ist das nicht der Fall, so gibt es falsche Referenzen was zu Applikationslogik-Fehler führt. Die einfachste Lösung für dieses Problem sind Transaktionen. Die Datenbank-Transaktionen haben jedoch eine Limitierung an Datensätzen, was den Spielraum einschränkt. Aus diesem Grund wurde eine eigene, logische Transaktion umgesetzt. Die Daten werden dabei nacheinander eingefügt und mit einem Zeitstempel markiert, welcher zu Beginn des Updates erstellt wird. So können alle Daten einer Transaktion zugewiesen werden.

Vorbereitungsphase Eine Transaktion ermöglicht es die neuen Daten erst bei erfolgreichem Import verfügbar zu machen. Da aber keine echte Datenbanktransaktion verwendet werden konnte, muss sichergestellt werden, dass nur eine Update-Instanz zur selben Zeit arbeitet. Um eine isolierte Ausführung sicherzustellen, mussten insbesondere zwei Faktoren beachtet werden. Als erstes muss es irgendwie möglich sein zu erkennen, ob aktuell ein Update durchgeführt wird. Als zweites muss man dann noch beurteilen können ob das laufende Update kein fehlerhaftes Update ist, welches das System blockiert. Um diese Probleme zu lösen wird ein Update-Status (IN_PROGRESS, SUCCESS, FAILURE) mit einem Zeitstempel verwendet. Ein Update startet immer im Status IN_PROGRESS und endet entweder im Status SUCCESS oder FAILURE. Der Zeitstempel definiert den Start des Updates. Befindet sich ein Update für eine ungewöhnlich lange Zeit im Status IN_PROGRESS, ist es höchstwahrscheinlich korrupt. Die Definition von „ungewöhnlich lange Zeit“ ist je nach Arbeitsumgebung unterschiedlich und kann durch eine Zeitangabe in Minuten in einer Konfigurationsdatei festgehalten werden.

In dieser Arbeit wird davon ausgegangen, dass es nur eine Import-Instanz gibt, weshalb grundsätzlich auf eine Synchronisierung von nebenläufigen Instanzen verzichtet werden kann. Für den speziellen

Fall, dass dennoch mehrere Instanzen gestartet werden und diese beinahe zum gleichen Zeitpunkt ein Update starten möchten, wurde ein sogenanntes „Two-Phase-Locking“ implementiert, welches im Notfall beide Updates unterbindet. Die Updates werden dann gemäss dem definierten Zeitplan, zu einem späteren Zeitpunkt neu gestartet. Eine weitere Kollision der exakten Update-Zeitpunkten ist dabei sehr unwahrscheinlich. Das untenstehende Flussdiagramm zeigt die Vorbereitungsphase eines Updates. In der rechten Spalte ist die Implementierung des „Two-Phase-Locking“ modelliert.

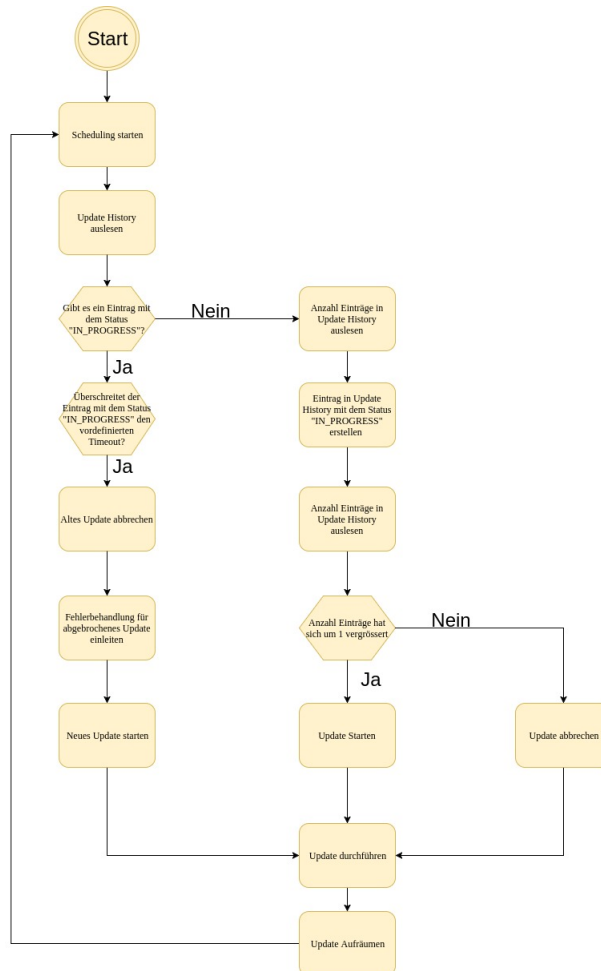


Abbildung 4.7: Flussdiagramm Static-Update

Erfolg Bei einem erfolgreichen Update wird der neue Zeitstempel als gültig markiert. Die Applikationslogik kann nach einem erfolgreichen Update entweder auf den alten Daten weiterarbeiten oder durch Aktualisieren des eigenen Daten-Zeitstempels auf die neuen Daten zugreifen. Die aktuelle Einstellung hält Daten zweier Generationen: Die Daten vor dem Update und die Daten des Updates selbst. Die Anzahl zu speichernde Generationen kann in einer Konfigurationsdatei festgelegt werden.

Fehlerbehandlung Für den Fall, dass es während des Updates ein Fehler in Form einer Exception gibt, wurde eine Fehlerbehandlungsroutine eingebaut. Die Fehlerbehandlungsroutine ist das Pendant zu den sogenannten Transaktions-Rollbacks. Da jeder neue Eintrag durch einen Zeitstempel identifizierbar ist, sucht die Fehlerbehandlung bei einem Fehlerfall alle zu dem fehlgeschlagenen Update zugehörigen Daten und entfernt diese aus den Datenbanken. Der Fehler wird registriert und in der Datenbank festgehalten.

4.3.2 Realtime-Update

Das Realtime-Update zeichnet sich durch andere Eigenschaften als das Static-Update aus. Das Realtime-Update muss nicht im Rahmen einer Transaktion durchgeführt werden, da jeder Update Eintrag für

sich steht und dadurch schon alleine gültig ist. Das Update ist jedoch zeitkritisch, da die Echtzeitinformationen mit jeder Verspätung an Wichtigkeit verlieren. Im GTFS-Format sind zwei Arten von Updates vorgesehen: FULL_DATASET und DIFFERENTIAL. Wobei letzteres für ein inkrementelles Update-Verfahren steht, welches sich noch im Entwicklungsstatus befindet und im Prototyp nicht verwendet wird. Das Update-Verfahren welches verwendet und auch umgesetzt wurde ist das FULL_DATASET. In dieser Variante steht jeder Update-Feed für sich. In anderen Worten: Bei jedem neuen Realtime-Update verlieren die vorherigen Updates ihre Gültigkeit.

Erfolg Im Good-Case sind alle relevanten Realtime-Updates in der Datenbank erfasst. Ein Update wird dann als relevant eingestuft, wenn es einen Eintrag der Fahrplandatenbank referenziert. Fehlt diese Referenz, wird das Realtime-Update ignoriert.

Fehlerbehandlung Bei einem Fehler während des Parsens oder Einfügens eines Realtime-Update wird dieses ignoriert und die Verarbeitung der weiteren Updates fortgeführt.

4.4 Amazon Web Services

Zur Überprüfung und Demonstration des Prototyps ist die gesamte Infrastruktur auf AWS installiert worden. Damit sich die Komponenten optimal in die Umgebung der Amazon Cloud einfügen, sind die Module regelmässig in die Cloud deployed und die Angebote auf ihre Tauglichkeit überprüft.

4.4.1 Anzeigemodul

Das Anzeigemodul ist eine klassische Web Anwendung, welche Daten zur Verfügung stellt. Ein PaaS ist die ideale Wahl für eine solche Applikation. Das PaaS Angebot von Amazon heisst Elastic Beanstalk, welches es unter anderem erlaubt, eine Konfiguration für ein Docker Container hochzuladen, welcher danach automatisch ausgeführt wird. Mit Elastic Beanstalk kann eine Anwendung mittels Umgebungsvariablen im Web-Interface konfiguriert werden. Zudem kann eine Anwendung bei Bedarf automatisch skalieren.

Installation

Um in Elastic Beanstalk ein Docker Container deployen zu können, muss sich dieser in einer Docker Registry befinden. Für den Prototyp ist die in Abschnitt 4.1.1 erwähnte EC2 Registry (kurz ECR) verwendet worden. Ist das Angebot korrekt konfiguriert worden, lädt sich die Umgebung automatisch den Container herunter und startet ihn.

Skalierung

Der Service von Amazon bietet einfache Möglichkeiten zur vertikalen und horizontalen Skalierung an.

Vertikale Skalierung Elastic Beanstalk basiert auf EC2, dem IaaS Angebot von Amazon. Vertikal lässt sich das Angebot mittels dem EC2 „Instanz Typ“ konfigurieren. Diese sind in verschiedene Gruppen, wie Speicher-optimierte Angebote, RAM-Optimierte Angebote oder eine Gruppe für die allgemeine Verwendung aufgeteilt. Die Gruppe „Allgemeine Verwendung“ oder „t2“ eignet sich sehr gut für Web-Services. Im Falle des Prototyps wurde der Instanz Typ „t2.micro“ gewählt. Somit steht einer Instanz 1 CPU und 1 GB Arbeitsspeicher zu, was für Demonstrationszwecke vollkommen ausreicht.

Horizontale Skalierung Elastic Beanstalk bietet aber auch eine horizontale Skalierung an. Diese ist sehr einfach zu konfigurieren, bietet aber gleichzeitig sehr viele Möglichkeiten die Skalierung zu steuern. Die Auslöser der Skalierung sind sehr stark von der Software abhängig, welche in die Cloud deployed wird. Da Amazon sehr viele Metriken anbietet, welche als Auslöser dienen, kann auch die Anwendung ideal skaliert werden, vorausgesetzt die Leistungsgrenzen der Software sind bekannt.

Die Antwortzeit des Anzeigemoduls lässt bei einer Belastung der CPU von 65% etwas nach und bei 90% beginnen Anfragen, aufgrund fehlender Ressourcen, fehlzuschlagen. Somit gilt es zu vermeiden, dass die Auslastung des Prozessors einer Instanz an diese Grenze gerät. Die Anwendung startet ab

einer maximalen CPU Last von über 65% jeweils eine neue Instanz, und einer Auslastung von 30% wird jeweils eine heruntergefahren. Diese Grösse wird mittels Durchschnitt der Belastung aller Verfügbaren Instanzen berechnet. Der Load-Balancer von Amazon, welcher bei der Aktivierung der horizontalen Skalierung automatisch verwendet wird, teilt die Last auf alle Verfügbaren Instanzen auf. Damit die Last auf alle Instanzen gleich verteilt wird, kann der Load Balancer alle verfügbaren Metriken der jeweiligen Instanzen abfragen und passt die Weiterleitungsstrategie an.

4.4.2 Datenbanken

Amazon bietet in seiner Cloud viele Services zur Speicherung von Daten an. Für die Persistierung von relationalen Daten können die in Unterabschnitt 4.1.1 beschriebenen Services verwendet werden.

Installation

Das Aufsetzen einer solchen Datenbank ist sehr einfach. Beim Erstellen der Instanz wählt man den Datenbank-Anbieter, Namen, Login-Daten etc. Nach einem kurzen Setup ist die Datenbank bereit und kann verwendet werden.

Skalierung

Das RDS Angebot von Amazon bietet Möglichkeiten zur Skalierung. Sie sind aber auf die Möglichkeiten der Datenbank-Anbieter begrenzt.

Vertikale Skalierung Die RDS Instanzen von Amazon basieren, wie auch Elastic Beanstalk auf EC2. Die vertikale Skalierung unterliegt somit denselben Kriterien. Beide verwendeten Datenbanken haben 1 CPU, 2 GB Arbeitsspeicher für die Fahrplan Datenbank bzw. 1 GB Arbeitsspeicher für die Mapping Datenbank. Grund für diese Unterscheidung ist die unterschiedliche Belastung der Datenbanken, welche im Betrieb beobachtet wurde.

Horizontale Skalierung Datenbanken können beschränkt horizontal skalieren. Schlussendlich basieren alle Lösungen auf Replikationen und Synchronisierung der verschiedenen Instanzen. Mit RDS können sogenannte „Read-Only Replicas“ erstellt werden. Dies sind eigenständige Datenbank Instanzen, welche lesenden Zugriff anbieten, und sich mit der Master Datenbank abgleichen. Da diese Datenbanken aber andere Hostnamen haben, muss zusätzlich eine Load Balancing Mechanismus verwendet werden, beispielsweise über ein gemeinsamen DNS-Eintrag.

4.4.3 Import Module

Die beiden Import Module verarbeiten regelmässig eine grosse Menge an Daten. Dies ist eine Charakteristik einer Batch-Anwendung. Amazon bietet mit AWS Batch eine Lösung an, mit der solche Abläufe einfach und kostengünstig abgewickelt werden können. Leider basiert dieses Angebot sehr stark auf den Infrastrukturen von Amazon. Somit wäre die gesamte Anwendung nicht mehr auf eine andere Cloud portierbar. Die Verwendung von Elastic Beanstalk musste leider auch schnell verworfen werden. Grund dafür ist, dass das PaaS nur mit Programmen funktioniert, welche über HTTP erreichbar sind oder Jobs über den Simple Message Service von Amazon empfangen können. Da beide Import Module mittels Docker gestartet werden können, sind die IaaS Instanzen momentan noch die beste Wahl. Denn beide Komponenten müssen nicht skalieren und können unabhängig voneinander ihre Aufgaben ausführen.

Installation

Die EC2 Instanzen verwenden das Linux Image von Amazon, das sich AMI nennt. Alle EC2 Instanzen welche im t2 Tier sind, sind an eine sogenannte CPU-Balance gebunden. Diese definiert, wie viel CPU-Leistung einer Instanz während einer Periode zur Verfügung steht. Ein Credit in dieser Balance bedeutet, dass die CPU in dieser Periode während einer Minute zu 100 % Ausgelastet sein kann, oder zu 50 % in zwei Minuten etc. [3]. Da das Realtime Import Modul die CPU viel öfter belastet als das Static Import Modul, verwendet das Realtime Modul „t2.small“, das Static Modul „t2.micro“ als Instanz-Typ.

Zusätzlich muss auf diesen Instanzen Docker konfiguriert, das Image der entsprechenden Komponente heruntergeladen und einen Container daraus konstruiert werden. Die Umgebungsvariablen, welche die Anwendung konfigurieren, können beim konstruieren des Containers angegeben werden. Dieser Prozess kann mit Docker-Compose jedoch vereinfacht werden.

Skalierung

Da beide Import Module nicht für eine horizontale Skalierung ausgelegt sind, bleibt dafür lediglich die vertikale Skalierung. Wie in RDS und Elastic Beanstalk, ist die verfügbare Leistung durch den Instanz-Typ definiert.

4.4.4 Deployment Architektur

Die aus den verschiedenen Komponenten entstandene Struktur ist mit einer Pipeline zu vergleichen.

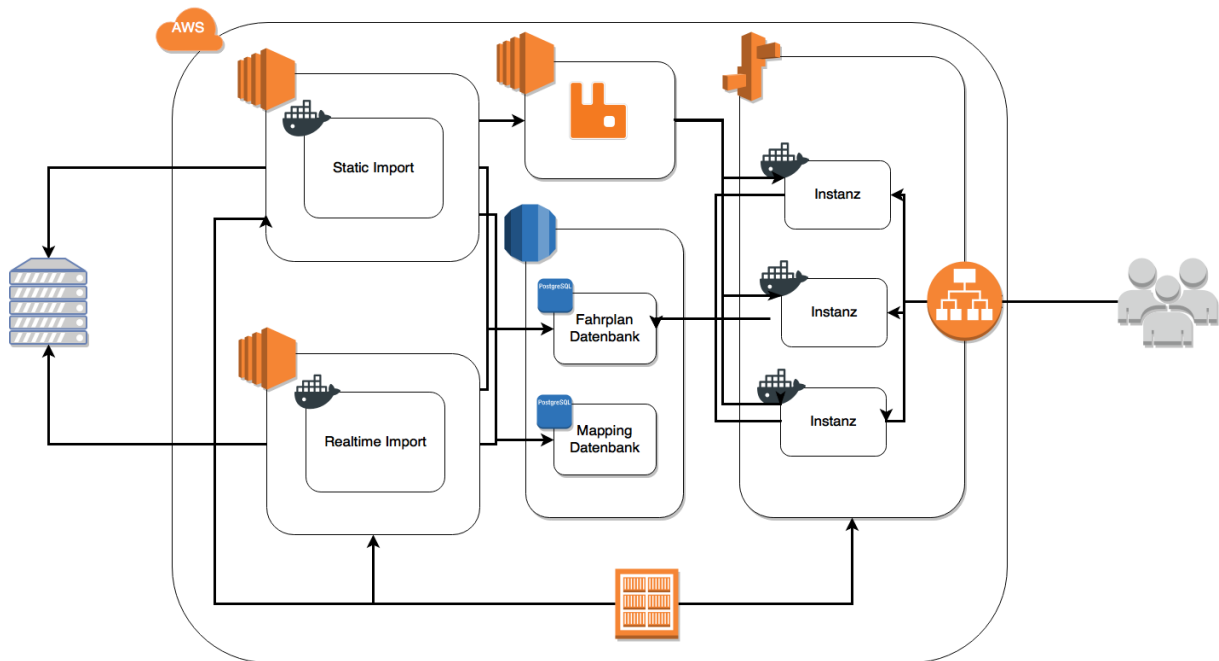


Abbildung 4.8: Struktur in AWS

EC2

Die Daten werden von der API des Auftraggebers geladen, in die jeweilige Import Instanz geladen, und die Resultate der Konvertierung in die Datenbank gespeichert. Die Import Komponente sind in EC2 Instanzen deployed.



RDS

Die Datenbank wird mit RDS gehostet und beinhaltet die Fahrplan- und Mapping Datenbank. Der Datenbank muss, als zentrale Komponente, genügend Leistung zur Verfügung gestellt werden, so dass sie nicht zu einem Engpass wird.



Kommunikation

Beim erfolgreichen Import der statischen Daten benachrichtigt die Import Instanz alle Anzeigekomponenten, damit diese auf die neusten Versionen der statischen Daten umschalten.



Elastic Beanstalk

Alle Instanzen des Anzeigemoduls werden durch Elastic Beanstalk verwaltet. Die Skalierung wird vom Elastic Loadbalancer, der zwischen Browser und Serverinstanz geschaltet wird, verwaltet. Hier sind beispielhaft drei Instanzen verwendet worden.



ECR

Die Containerregistry verwaltet alle Docker Images und speichert deren Versionsverlauf. Alle Komponenten des Systems werden von diesem Service bereitgestellt.



Kapitel 5

Testing

Der Prototyp ist während des ganzen Entwicklungsprozess regelmässig durch Tests geprüft worden. In den Tests wurden die Anforderungen verifiziert und sichergestellt, dass die Performance in einem akzeptablen Rahmen ist. Da die Komponenten unabhängig voneinander funktionieren, wurde auf Integrations- und Systemtests verzichtet. Allem voran ist das korrekte Verhalten des Systems anhand von Vergleichen der angezeigten Daten mit einer bestehenden Lösung des Auftraggebers überprüfbar.

5.1 Cloud

Um die Anforderung der Cloud-fähigkeit zu testen, wurde die Anwendung in regelmässigen Abständen in die Cloud deployed. Der Continuous Integration Prozess, beschrieben in Abschnitt 5.6.3, deployed jede neue Version automatisch in die entsprechende Cloud Instanz. Durch manuelle, situationsbezogene Tests konnten der Prototyp schnell verifiziert werden. Die Performance der einzelnen Komponenten wurden regelmässig getestet und mit vorherigen Resultaten verglichen. Wichtig bei diesen Tests war sowohl, dass die Anwendung gut in das Cloud Produkt passt, aber auch, dass das richtige Angebot ausgewählt wurde.

5.1.1 Ergebnis

Zu Beginn der Arbeit ist ein PaaS Angebot für alle Komponenten vorgesehen gewesen. Mit der Verwendung von Docker für jede Komponente wäre das Deployment dieser ein sehr einfacher, standardisierter Prozess. Leider ist Elastic Beanstalk nicht für Anwendungen gemacht, welche keine Anbindung per HTTP oder das Messaging System von Amazon anbieten. Eine Installation mit Docker auf Elastic Beanstalk muss entweder über HTTP erreichbar sein, oder Jobs über den Simple Queue Service annehmen. Aufgrund dessen wurde im Verlauf der Arbeit für die Import Module auf EC2 Instanzen gewechselt. Für die Anzeigeinstanz ist das Elastic Beanstalk Angebot ideal. Es bietet einfache Skalierung an, kann schnell neue Versionen deployen und erlaubt einen guten Einblick in den Zustand der Anwendung.

5.2 Lasttests

Damit die Anforderung der Skalierung überprüft und erfüllt werden kann, sind regelmässig Lasttests der Anwendung durchgeführt worden. Diese Tests versuchen stationäre Anzeiger zu simulieren, welche pro Minute je sechs Anfragen auf zufällig ausgewählte Haltestellen auslösen. Die Lasttests simulieren somit in der Spitze 400 Anfragen pro Sekunde, was etwa 4000 Instanzen entspricht. Die Tests bauen die Last über etwa fünf Minuten hinweg auf, wobei pro Minute die Anzahl Requests pro Sekunde um 40 zunimmt. Diese Last wird über fünf Minuten gehalten und dann, in denselben Schritten wie beim Aufbau, wieder abgebaut. Die entstandene Last ist in Abbildung 5.1 zu erkennen. Für die Durchführung dieser Tests ist die in ?? beschriebene Infrastruktur verwendet worden. Der Load-Balancer der Anwendung wurde auf ein Maximum von 4 Instanzen eingestellt. Der Test startet mit einer verfügbaren Instanz.

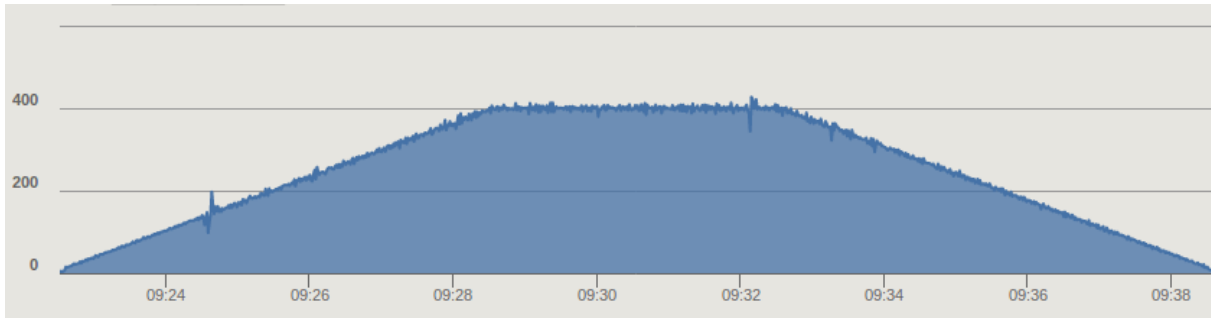


Abbildung 5.1: Anzahl Anfragen pro Sekunde im Lasttest

5.2.1 Ergebnis

Im Verlauf des gesamten Tests musste die Anwendung 250'000 Anfragen während 20 Minuten verarbeiten. Die in Tabelle 5.1 ausgewerteten Antwortzeiten sind sehr gut und haben die Anforderungen mehr als erfüllt.

Durchschnitt	60 ms
Std. Abweichung	28 ms
Minimum	22 ms
50. Perzentil	56 ms
75. Perzentil	74 ms
95. Perzentil	98 ms
99. Perzentil	114 ms

Tabelle 5.1: Statistische Verteilung der Antwortzeiten im Lasttest

Trotz der grossen Belastung der Anwendung sind die Antwortzeiten sehr tief. Im Schnitt dauerte es 60 ms, bis die Anzeigeanstanz eine Antwort des Webservers erhielt und 99 % der Anfragen sind in 114 ms oder weniger beantwortet worden. Wenn man diese Daten im Zeitstrahl betrachtet, kann man gut erkennen, wie die PaaS Umgebung neue Instanzen hochfährt, wenn die Last einer Instanz zu gross wird. Abbildung 5.2 visualisiert die Ergebnisse aus Tabelle 5.1.

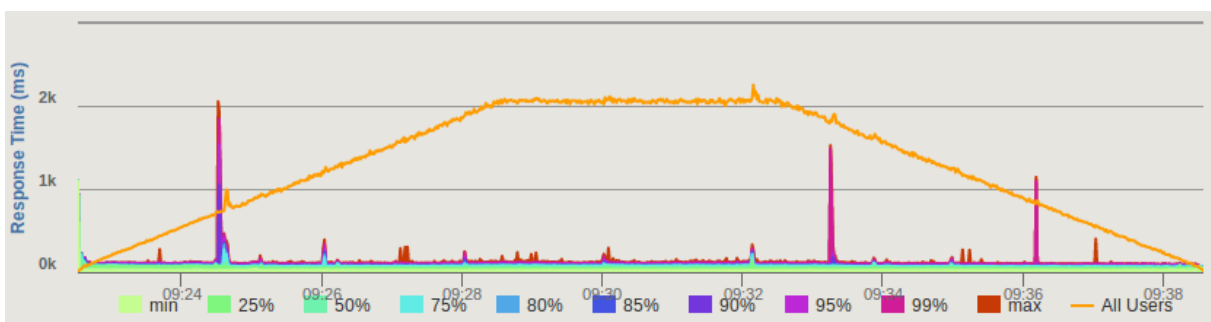


Abbildung 5.2: Perzentile der Antwortzeiten im Lasttest

Zum besseren Verständnis von Abbildung 5.2 ist die Lastkurve in orange eingefügt worden. Somit fallen auf dieser Grafik zwei Punkte auf: Als erstes stechen die Zacken der Antwortzeit ins Auge. Diese entstehen, wenn eine Instanz an ihre Leistungsgrenzen gelangt und die Cloud eine neue hochfährt. Eine einzelne Instanz erreicht ab 160 Anfragen pro Sekunde die Leistungsgrenzen und wird langsamer. Dies entspricht etwa 1500 Monitoren, welche alle zehn Sekunden neue Daten anfordern. Der zweite Punkt welcher Auffällt, sind, dass die Antwortzeiten, abgesehen von den Spitzen, sehr konstant sind.

Drei Viertel der Anfragen werden fast in derselben konstanten Zeit beantwortet, was anhand der türkisenen Linie (75. Perzentil) ersichtlich ist, welche beinahe flach ist.

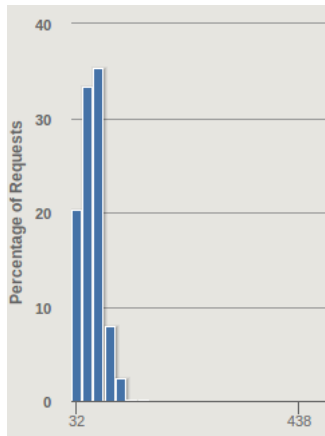


Abbildung 5.3: Verteilung der Antwortzeiten im Lasttest

Wie in Abbildung 5.3 sichtbar, sind die Antwortzeiten über ein kleines Intervall verteilt und somit sehr konstant. Dies deckt sich mit den Erkenntnissen aus Abbildung 5.2.

Teil IV

Ergebnisse

5.3 Funktionalität

Die Import Module unterstützen den GTFS-Standard. Es ist wichtig zu erwähnen, dass es innerhalb des Standards Spielraum für die Aufbereitung der Daten gibt. Die im Prototypen entwickelten Import Module unterstützen das GTFS-Format der BERNMOBIL mit den in der Aufgabenstellung gewünschter Funktionalität. Tauscht man die Datenquellen mit einer ebenfalls gültigen GTFS-Implementation der Daten aus, ist es möglich, dass einzelne Informationen nicht korrekt eingelesen werden können. Dieses Verhalten ist nicht vermeidbar und verlangt zur Korrektur eine Anpassung der Import-Komponenten.

Soll- und Ist-Daten Der resultierende Abfahrtsanzeiger zeigt alle Soll-Daten als Abfahrten einer Haltestelle an. Diese stimmen mit den Informationen, welche vom GTFS-Format aus der Datenquelle von BERNMOBIL extrahiert wurden überein. Die Ist-Daten werden in einer separaten Spalte in Form von Verspätungen angezeigt. Um Verspätungen anzeigen zu können, werden diese aus einer separaten Datenquelle der BERNMOBIL heruntergeladen. Die Korrektheit der Daten sind in Stichproben mit dem Referenzprodukt¹ von BERNMOBIL überprüft worden.

Metabahnhöfe Das Konzept der Metabahnhöfe ist im Prototyp umgesetzt. Metabahnhöfe werden als sogenannte Areas in der Datenbank gespeichert. Eine rekursive Beziehung von Areas wurde nicht umgesetzt, da keine solche Daten vorhanden waren und dies in der Aufbereitung der Daten von BERNMOBIL kein Thema war.

5.4 Erweiterbarkeit

Die Architektur des Prototyps ist auf die Erweiterbarkeit ausgelegt und vereinfacht diese. Eine Erweiterung könnte beispielsweise durch Hinzufügen von weiteren Datenquellen, sowohl im GTFS- aber auch in einem fremden Daten-Format, stattfinden. Um weitere Datenquellen anzubinden können entweder die aktuellen Import-Module angepasst oder neue Import-Module geschrieben werden. So lange das Datenbankschema nicht angepasst wird, funktioniert der Abfahrtsanzeiger wie gewohnt, da dieser eine eigenständige Komponente darstellt.

Möchte man den Prototypen durch ein weiteres Modul ergänzen, kann dieses von der normalisierten Datenbasis der Fahrplandatenbank profitieren. Wenn ein weiteres Modul hinzugefügt wird sollte auf die Auslastung der Fahrplandatenbank geachtet werden. Denn sie ist die zentrale Komponente für die Import-, aber auch weitere Front-End Module.

5.5 Skalierbarkeit

Die Skalierbarkeit, wie sie in den Anforderungen festgehalten wurde, ist erfolgreich umgesetzt. Auch hier unterstützt die gewählte Architektur die Anforderung. Es ist wichtig zu erwähnen, dass nicht alle Komponente für eine beliebige Skalierung geeignet sind. Gemäss Aufgabenstellung und auch Architektur, sind die Module, welche von der Datenbank lesen, in unserem Fall der Abfahrtsanzeiger, skalierbar modelliert.

¹<https://www.oev-plus.ch>

5.6 Ausblick

In diesem Kapitel sind Aspekte und Weiterentwicklungsmöglichkeiten beschrieben, welche im Rahmen dieser Arbeit nicht umgesetzt wurden. Grund dafür war entweder die limitierte Zeit, Ressourcen oder noch nicht verfügbare Funktionalität.

5.6.1 Anzeigemodul

Web-Sockets

Die Aktualisierung der Fahrplaninformationen geschieht im Front-End in einem vordefinierten Intervall von zehn Sekunden. Dabei werden via AJAX-Request die neusten Abfahrtsinformationen nachgeladen. Dieses Intervall könnte man durch Web-Sockets ersetzen, so dass der Webserver das Front-End des Anzeigemoduls über neue Daten informiert. Es ist zu beachten, dass die Anzeige in jedem Fall in einem höheren Takt aktualisiert werden muss, da Abfahrten, welche in der Vergangenheit liegen, nicht mehr angezeigt werden. Dies kann mithilfe von einfachen DOM-Manipulationen im Frontend gelöst werden.

5.6.2 Cloud-Environment

Amazon Aurora

Es ist sinnvoll, die Datenbank auf ein Angebot des Cloud-Anbieters zu deployen, welches eine automatische Skalierung dessen ermöglicht. Im Amazon-Umfeld wäre dies das „Amazon Aurora“-Angebot: Eine relationale Datenbank-Engine, welche die Geschwindigkeit und Zuverlässigkeit einer kommerziellen Datenbank mit der Wirtschaftlichkeit einer Open-Source Datenbank vereint[2]. Es skaliert Speicher und Instanzen automatisch und überwacht die gesamte Infrastruktur vollautomatisch. Jedoch ist zum Zeitpunkt der Arbeit keine Version für PostgreSQL verfügbar, weshalb dieses Angebot für diese Arbeit nicht verwendet wurde. Eine PostgreSQL-Version wird bereits erwartet.

Import auf PaaS

Die Importkomponenten können mit dem aktuellen Angebot von Amazon nicht auf ein PaaS deployed werden. Wie in Unterabschnitt 4.4.3 beschrieben, müsste die Anwendung stark angepasst werden. Zudem müssten, um die Produkte von Amazon verwenden zu können, deren Frameworks und Libraries verwendet werden, was die Kopplung an den Cloud Provider erhöht. Jedoch wäre das Amazon Batch Angebot günstiger und einfacher zu verwalten, als die IaaS Instanzen.

5.6.3 Erweiterung

Mehr Datenformate

Möchte man den Abfahrtsanzeiger durch weitere Datenquellen mit unterschiedlichen Datenformate wie beispielsweise VDV-453/4 ergänzen, so muss dafür eine neue Import Komponente geschrieben werden. Werden die Fahrplandatenbank nicht angepasst, so wird der Abfahrtsanzeiger wie gewünscht weiter funktionieren, da dieser von den Importkomponenten durch die Fahrplandatenbank entkoppelt ist.

Mehrere Realtime-Import Module

Auch wenn in dieser Arbeit keine Verwendung von mehreren Realtime-Import Modulen vorgesehen ist, ist es dennoch möglich und je nach Anwendungsfall sogar sinnvoll. Es ist zu beachten, dass die aktuelle Implementation nur die letzten importierten Updates berücksichtigt. Soll der Prototyp durch weitere Realtime-Import Module (zum Beispiel von einem anderen Daten-Standard oder Datenquelle) ergänzt werden, so muss das aktuelle Realtime-Import Modul minimal angepasst werden. Es muss möglich sein, die Updates einer Datenquelle zuordnen zu können, um beim Einspielen von neuen Updates die zugehörigen alten Updates aus der Fahrplandatenbank entfernen zu können.

Inkrementelle Updates

Wie im Unterabschnitt 4.3.2 beschrieben, sieht der GTFS-Standard zwei Arten von Realtime-Updates vor: `FULL_DATASET` und `DIFFERENTIAL`. Der `DIFFERENTIAL`-Modus wird noch nicht unterstützt und wurde deshalb in dieser Arbeit nicht umgesetzt. Wenn dieser in Zukunft unterstützt wird, können auch `DIFFERENTIAL`-Updates durch anpassen des Realtime-Import Moduls verarbeitet werden.

Glossar

CI Continous Integration.

Docker Eine Containertechnologie, welche das Bereitstellen von Anwendungen vereinfacht.

EC2 Skalierbare Rechenkapazität in der Amazon Cloud.

Elastic Beanstalk Service zur Bereitstellung von Webanwendungen und -Services in verschiedenen Sprachen und Plattformen.

GTFS General Transport Feed Specification.

IaaS Infrastructure as a Service, eine Dienstleistung in der Cloud, welche Infrastruktur zur Verfügung stellt. Typisch um Betriebssystem zur Verfügung zu stellen, ohne die darunter liegende Hardware direkt kontrollieren zu können.

PaaS Platform as a Service, eine Dienstleistung in der Cloud welche, Laufzeitplattformen zur Verfügung stellt. Typisch um Anwendungen zu betreiben ohne das darunterliegende Betriebssystem kennen zu müssen.

Protocol Buffers Sprach- und Plattformneutraler, erweiterbarer Mechanismus um strukturierte Daten zu serialisieren.

RDS Relationale Datenbankplattform in der Amazon Cloud.

RESTful Representational State Transfer. Eine Abstraktion der Struktur und des Verhaltens von Web-Services.

SPOF Single Point of Failure.

Spring Framework Ein Java Framework dessen Ziel es ist, die Entwicklung mit Java/Java EE zu vereinfachen und gute Programmierpraktiken zu fördern.

VDV Verband Deutscher Verkehrsunternehmen. Unternehmen des öffentlichen Personal- und Güterverkehrs in Deutschland.

Literatur

- [1] Ericsson AB. *JSR 325: IMS Communication Enablers (ICE)*. Mai 2017. <https://jcp.org/en/jsr/detail?id=325>.
- [2] Amazon. *Produktdetails zu Amazon Aurora*. Apr. 2017. <https://aws.amazon.com/de/rds/aurora/details/>.
- [3] Amazon AWS. *T2 Instances*. Mai 2017. <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html>.
- [4] Christoph Fehling u. a. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.
- [5] Apache Software Foundation. *Clustering*. Apr. 2017. <http://activemq.apache.org/clustering.html>.
- [6] Google. *GTFS Realtime Referenz*. Apr. 2017. <https://github.com/google/transit/blob/master/gtfs-realtime/spec/en/reference.md>.
- [7] Google. *GTFS Static Referenz*. Apr. 2017. <https://developers.google.com/transit/gtfs/reference/>.
- [8] Todd Hoff. *How PayPal Scaled To Billions Of Transactions Daily Using Just 8VMs*. März 2017. <http://highscalability.com/blog/2016/8/15/how-paypal-scaled-to-billions-of-transactions-daily-using-ju.html>.
- [9] Gregor Hohpe und Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., 2003.
- [10] JOOQ. *Different use cases for JOOQ*. Mai 2017. <https://www.jooq.org/doc/3.9/manual/getting-started/use-cases/>.
- [11] J. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. Internet Engineering Taskforce, Okt. 2008. <https://tools.ietf.org/html/rfc5321>.
- [12] Jerry Kuch. *RabbitMQ Hits One Million Messages Per Second on Google Compute Engine*. Apr. 2017. <https://content.pivotal.io/blog/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine>.
- [13] Lightbend. *2.2 Billion Clicks And Beyond: How Hootsuite Modernized Its URL Shortener To Scala/Akka Using The Lightbend Reactive Platform*. März 2017. <https://www.lightbend.com/resources/case-studies-and-stories/how-hootsuite-modernized-its-url-shortener>.
- [14] Tony Garnock-Jones Matthias Radestock. *RabbitMQ, Open-Standard Business Messaging in 5000 lines of Erlang*. Mai 2017. <http://www.rabbitmq.com/resources/erlang-exchange-talk-final/ex.html>.
- [15] Pivotal. *Broker vs Brokerless*. Mai 2017. <https://www.rabbitmq.com/blog/2010/09/22/broker-vs-brokerless/>.
- [16] Pivotal. *Clients & Developer Tools*. Apr. 2017. <https://www.rabbitmq.com/devtools.html>.
- [17] Pivotal. *Federation Plugin*. Mai 2017. <https://www.rabbitmq.com/federation.html>.
- [18] Pivotal. *Firehose Tracer*. Mai 2017. <https://www.rabbitmq.com/firehose.html>.
- [19] Pivotal. *Highly Available (Mirrored) Queues*. Mai 2017. <https://www.rabbitmq.com/ha.html>.
- [20] Pivotal Software. *Clustering Guide*. Apr. 2017. <https://www.rabbitmq.com/clustering.html>.
- [21] Pivotal Software. *JSR-352 Support*. Mai 2017. <https://docs.spring.io/spring-batch/reference/html/jsr-352.html>.

- [22] Pivotal Software. *spring.io - Projects*. Mai 2017. <https://spring.io/projects>.
- [23] Spring. *Getting Started - Creating Asynchronous Methods*. März 2017. <https://spring.io/guides/gs/async-method/>.
- [24] Phil Webb. *How not to hate Spring in 2016*. Mai 2017. <https://spring.io/blog/2015/11/29/how-not-to-hate-spring-in-2016>.
- [25] Wikipedia. *Java Message Service* — *Wikipedia, Die freie Enzyklopädie*. Apr. 2016. https://de.wikipedia.org/w/index.php?title=Java_Message_Service&oldid=15346465.
- [26] Wikipedia. *Java Persistence API* — *Wikipedia, Die freie Enzyklopädie*. Apr. 2016. https://de.wikipedia.org/w/index.php?title=Java_Persistence_API&oldid=159789607.

Abbildungsverzeichnis

1.1 Übersicht GTFS Static Datenmodell	5
2.1 Diagramm der Antwortzeiten von Play im Skalierungstest	14
2.2 Diagramm der Antwortzeiten von Spring im Skalierungstest	15
2.3 Heap Grösse während Lasttests im Play Framework	16
2.4 CPU Auslastung während Lasttests im Play Framework	17
2.5 Anzahl Threads während Lasttests im Play Framework	17
2.6 Heap Grösse während Lasttests mit Spring Boot	18
2.7 CPU Auslastung während Lasttests mit Spring Boot	18
2.8 Anzahl Threads während Lasttests mit Spring Boot	18
3.1 Überblick der Komponenten	29
3.2 Package Diagramm	30
3.3 Zeitverschiebung der Updates	31
3.4 Cloud Architektur des Prototyps	32
3.5 Datenmodell des Prototyps	33
4.1 Schema der Fahrplan Datenbank	38
4.2 Referenzmodell von Spring Batch	41
4.3 Aktivitätsdiagramm des Static Import	42
4.4 Aktivitätsdiagramm des Realtime Import	44
4.5 Arbeitsablauf einer Anzeige	46
4.6 Abfahrtsanzeiger	47
4.7 Flussdiagramm Static-Update	50
4.8 Struktur in AWS	53
5.1 Anzahl Anfragen pro Sekunde im Lasttest	56
5.2 Perzentile der Antwortzeiten im Lasttest	56
5.3 Verteilung der Antwortzeiten im Lasttest	57

Anhang

Qualitätssicherung

Entwicklungsprozess

Damit die Resultate des Prototyps schnell und transparent zur Verfügung stehen, wird in dieser Arbeit Continuous Integration verwendet. Für die Entwicklung bedeutet das, dass der Workflow penibel eingehalten werden muss, damit einerseits keine fehlerhafte Version ausgerollt wird und andererseits regelmässig neue Versionen zur Verfügung stehen. Zur Unterstützung sind hauptsächlich Jenkins und Sonarqube verwendet worden.

Jenkins

Jenkins ist ein Continuous Integration (CI) Service. Es ermöglicht, dass beinahe jede erdenkliche Aufgabe, welche normalerweise manuell ausgeführt wird, automatisiert und nachvollziehbar wird. Sobald eine Änderung in das GitLab Repository gepusht wird, wird Jenkins benachrichtigt und startet den Build-Vorgang. Dieser beinhaltet das Kompilieren, Testen und die statische Analyse des Source Codes. Wurde der Master Branch verändert, wird die Anwendung zusätzlich in ein Docker-Image kopiert und in die EC2 Registry hochgeladen.

Sonarqube

Sonarqube analysiert den Source-Code und meldet bekannte Sicherheitsrisiken, schlechten Stil und prüft den Code gemäss Style-Guidelines. Es ist ein Hilfsmittel, um die Code Qualität zu erhöhen und bekannte Sicherheitsprobleme schnell zu erkennen.

Sonarqube

Um die Codequalität hoch zu halten, sind zwei verschiedene Werkzeuge verwendet worden. Der Aufbau des Source-Codes hält sich an die Google Java Style-Guidelines, welche in der IDE überprüft und angewendet werden. Für die statische Code Analyse wurde Sonarqube verwendet. Dieses statische Analysetool bietet eine grosse Menge an Regeln, welche in vielen Fällen sinnvoll sind. Bei jedem Deployment, wurden die Resultate der Module überprüft und die als sinnvoll erachtete Vorschläge wurden im Code verbessert. In diesem Abschnitt werden die Ergebnisse aus Sonarqube des fertigen Prototypen erklärt.

Für Java existieren in Sonarqube 245 Regeln für die Qualität des Codes. Davon sind 75 für Fehler (Bugs), 20 für Schwachstellen (Vulnerabilities) und 159 für den Stil (Code Smells).

Anzeigemodul

Dieses Modul hat mit ca. 2600 Zeilen Java Code und 250 Zeilen JavaScript Code die kleinste und übersichtlichste Code Basis.



Abbildung 4: Grösse der Codebasis des Anzeigemoduls

Die wenigen Code Zeilen lassen sich durch die bewusst umgesetzte schlanke Architektur erklären. Von diesen 2600 Zeilen ist lediglich 3% als Duplikation erkannt worden. Ein grosser Teil der Duplikationen wurde durch die sehr ähnlichen Entitäten detektiert. Weiter wurden in den Tests ähnliche Assert-Statements als Duplikation registriert.

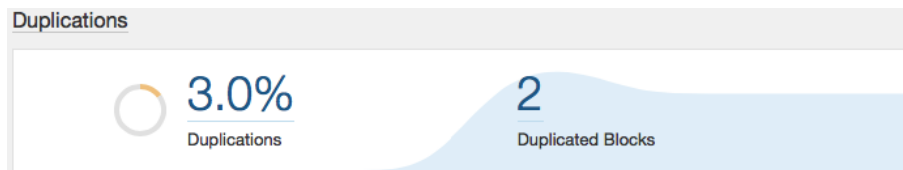


Abbildung 5: Duplikation im Code des Anzeigemoduls

Alle von Sonarqube gefundenen Schwachstellen und Fehler wurden behoben.



Abbildung 6: Auswertung Fehler und Schwachstellen im Anzeigemodul

Darüber hinaus existieren auch keine Code-Smells im Anzeigemodul.



Abbildung 7: Qualität des Codes im Anzeigemodul

Static Import Modul

Das Modul für den Import der statischen Daten ist das Grösste Projekt des gesamten Prototyps. Insgesamt gibt es über 10'000 Zeilen Code. Die 150 Zeilen JavaScript Code stammen von einem Gradle Plugin, welches eine HTML-Seite der Testresultate generiert. Da Jenkins das gesamte Artefakt an die Sonarqube Instanz weiterleitet, wird dieses ebenfalls analysiert.



Abbildung 8: Grösse der Codebasis im Importmodul für statische Daten

Sonarqube hat 5.6% der Zeilen als doppelt erkannt. Wie im Anzeigemodul sind auch hier Entities, welche ähnliche Felder haben dafür verantwortlich. Die Entities fallen in diesem Modul mehr ins Gewicht, da es aufgrund der Abbildung vom GTFS-Format mehr davon gibt. Ausserdem sind für die Test, Java Klassen mit JOOQ aus der Datenbank generiert worden. Diese befinden sich im Source-Code Repository und wirken sich ebenfalls auf das Resultat der Code Duplizierung aus.

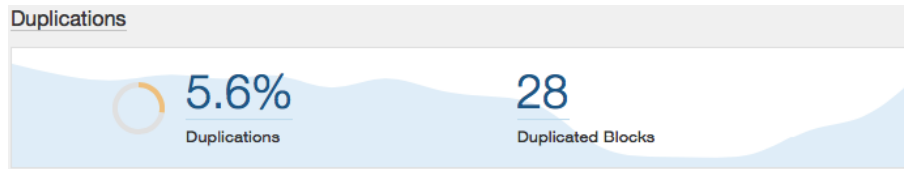


Abbildung 9: Duplikation im Code des Importmodul für statische Daten

Auch in diesem Modul erkennt Sonarqube keine Fehler und Schwachstellen.



Abbildung 10: Auswertung Fehler und Schwachstellen im Importmodul für statische Daten

Ausserdem wurden in diesem Modul ebenfalls keine Code-Smells gefunden.

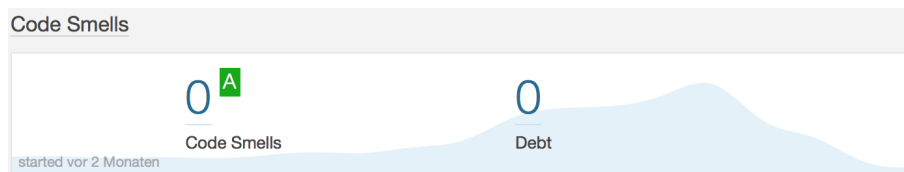


Abbildung 11: Qualität des Codes im Anzeigemodul

Realtime Import Modul

Das Importmodul für Echtzeitdaten umfasst 7400 Java Zeilen Code.

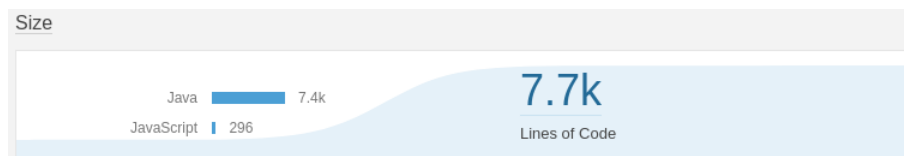


Abbildung 12: Qualität des Codes im Importmodul der Echtzeitdaten

In diesem Modul ist die prozentuale Duplikation am höchsten. Die Gründe dafür sind dieselben wie beim Importmodul der statischen Daten.



Abbildung 13: Qualität des Codes im Importmodul der Echtzeitdaten

Wie in den beiden anderen Modulen, existieren in diesem Modul keine erkennbaren Fehler und Schwachstellen.



Abbildung 14: Auswertung Fehler und Schwachstellen im Importmodul der Echtzeitdaten

Es sind ebenfalls keine Code-Smells erkannt worden.

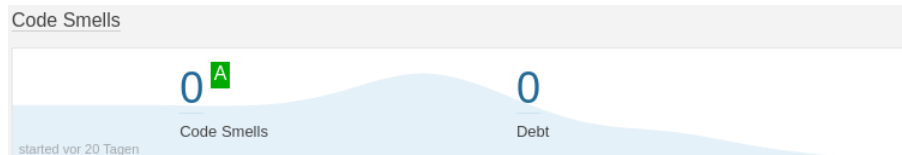


Abbildung 15: Qualität des Codes im Importmodul der Echtzeitdaten

Unit Tests

Um die Funktionalität einzelner Klassen und deren Zusammenspiel zu testen, ist es unerlässlich Unit-Tests zu schreiben. Das Ziel war möglichst schnell einen funktionierenden Prototyp präsentieren zu können. Dadurch sind automatisierte Tests in den Hintergrund gerückt, da es viel Zeit kostet diese zu schreiben. Trotzdem wurde bei jeder Komponente mindestens ein Test geschrieben um sicher zu stellen, dass die Test Umgebung funktioniert. In Spring ist es besonders wichtig, einen kompletten `ApplicationContext` für den Dependency Injection Mechanismus bereitzustellen. Beim Kennenlernen des Frameworks ist es schwer, die Fehlermeldungen, welche bei einer falschen Konfiguration auftreten, zu verstehen. Gemessen werden die Unit-Tests in der prozentualen Coverage der wichtigen Komponenten.

Anzeige Modul

In diesem Modul müssen in jedem Layer andere Testumgebungen konfiguriert werden. Da Spring Data bei dem Erstellen von Repositories viel Arbeit durch Heuristik übernimmt, muss der Data-Access Layer kaum getestet werden. Als einziges muss das `ScheduleRepository` getestet werden, da die Abfrage nach den Abfahrten eine komplexe Query ist, welche spezifische Logik fordert. Der Business Layer ist mit einer einzigen Klasse sehr klein gehalten. Dieser ist durch Mocks der Repositories getestet. Auf Integrationstests werden verzichtet, weshalb die HTTP Endpunkte nicht getestet werden. Jedoch werden die Controller Klassen getestet, welche HTML oder JSON zurückgeben. Die angestrebte Coverage der zu testenden Komponenten liegt zwischen 90 und 100%.

Ergebnis

Durch die Konsequente Anwendung von Best Practices des Frameworks, konnte dieses auch für Tests einfach verwendet werden. Es sind fast alle Teile der Anzeigekomponente getestet, die Coverage beträgt fast 100%. Ein kleiner Wehrmutstropfen ist jedoch, dass das `ScheduleRepository` nicht getestet werden kann. Grund dafür ist die Verwendung der neuen „Date and Time API“ aus Java 8 in den Entitys. JPA, und somit auch Hibernate, bieten keine native Unterstützung für die neuen Datentypen, weshalb eigene Konverter Klassen geschrieben werden mussten. Diese werden mit einer Java Annotation markiert und das Framework erkennt beim Konstruieren der Repositories automatisch die Konverter. Die Testumgebung kann nicht automatisch konstruiert werden, wodurch diese Klassen manuell registriert werden müssten. Hibernate bietet aber keine Möglichkeit dies zu tun, weshalb diesen Teil der Anwendung ungetestet bleibt.

Static Import Modul

Dieses Modul hat viel Logik, die getestet werden muss, jedoch auch viel Konfiguration des Frameworks. Ein besonders wichtiger Bereich sind die „Processor“. Diese sind das Herzstück des Imports der statischen Daten und sind beinahe der wichtigste Teil des gesamten Systems. In einem Batch Ablauf sind Fehler schwer zu finden und ziehen meist Folgefehler mit sich. Deshalb ist es besonders wichtig, dass Fehlerfälle eliminiert werden. Die Fehlerfälle, welche nicht eliminiert werden können müssen ins Log geschrieben und deren Behandlung durch Tests überprüft werden. Die Konvertierung der GTFS Daten muss für jeden möglichen Fall definiert sein. Fehler und Exceptions müssen abgefangen und behandelt werden. Es ist äusserst wichtig, dass Fehler komplett nachvollzogen werden können. Eine weitere wichtige Komponente des Moduls sind die `MappingStore`, welche ebenfalls gut getestet und auf plausible Fehlerfälle untersucht werden müssen. Die Konfiguration des Frameworks und der Jobs und Steps werden nicht getestet. Da diese sehr einfach gehalten sind, kann immer durch Methoden-namen erkannt werden, was in diesen konfiguriert wird. Zudem werden diese Teile des Moduls mit JavaDoc dokumentiert. Die angestrebte Coverage ist in diesem Modul entspricht etwa 80 %.

Ergebnis

Die gesamte Coverage in diesem Modul ist knapp über 50%. Wenn aber die oben erwähnte Konfiguration davon abgezogen wird, dann ist die erreichte Coverage bei 82%. Alle `ItemProcessor` sind zu 100% getestet sowie alle Teile der `ItemReader` und `ItemWriter`, welche nicht von externen Ressourcen abhängen (wie beispielsweise der GTFS API oder der Datenbank).

Realtime Import Modul

Dieses Modul ist ähnlich wie das Static Import Modul. Es unterscheidet sich aber darin, dass in diesem Modul fast keine Konfiguration zu finden ist. Ein wichtiger Bereich ist die Konvertierung der GTFS Entitäts in das eigene Schema. Wie auch im Static Import Modul ist es auch hier wichtig, dass Fehlerfälle, wenn möglich eliminiert werden können, oder abgefangen, behandelt und aufgezeichnet werden. Das Laden und Zwischenspeichern der Daten wird nicht getestet, das Speichern der konvertierten Objekte wird nur in kleinem Masse getestet. In diesem Modul ist die angestrebte Test Coverage etwa 70%.

Ergebnis

Da in diesem Modul viel selbst entwickelt worden ist, konnte für die Tests auch kein Framework verwendet werden. Wie auch im Static-Import Modul sind alle Komponenten, welche nicht von externen Ressourcen abhängig sind getestet worden. Dies beinhaltet vier der sechs Repositories, sowie die `ImportRunner` Klasse, welche den ganzen Import koordiniert. Somit ist die Test Coverage bei etwa 80%.

Shared Module

Diese Modul beinhaltet primär Speicherobjekte, welche lediglich Felder, Getter und Setter haben. Diese werden nicht getestet. Wichtig jedoch ist das korrekte Verhalten des `UpdateManager` und dessen Repositories. Da diese Komponente in mehreren anderen Komponenten verwendet wird, ist es umso wichtiger, gute Test zu schreiben, um bei Änderungen Fehler zu vermeiden. Die angestrebte Test Coverage liegt bei 80%.

Ergebnis

Dieses Modul wird sehr ausführlich getestet. Die Coverage über alle Klassen, welche Logik beinhalten liegt bei 97%. Es werden, wie bereits erwähnt, der `UpdateManager` sowie die drei Repositories, welcher dieser verwendet getestet.

Konfiguration

Alle Komponenten des Prototyps können mit verschiedenen Parametern konfiguriert werden. Dafür ist der „Configuration File Processor“ von Spring verwendet worden, der standardmässig Konfigurationen aus definierten Dateien liest und in der Anwendung verfügbar macht. Für eine Cloud-Anwendung sind Umgebungsvariablen eine weitere Abstraktion zwischen Konfiguration und Ausführung. Denn mit solchen Variablen kann die Anwendung aus dem User Interface der Cloud konfiguriert werden. In den folgenden Unterkapiteln werden die Möglichkeiten der Konfiguration der Anwendung in der Konfigurationsdatei sowie mit Umgebungsvariablen erläutert.

Applikations-Konfiguration

Wird die automatische Konfiguration des Spring Frameworks aktiviert, werden alle Dateien im ‚resource‘ Ordner, welche die Endung `.properties`, `.yml` oder `.xml` haben, als Konfigurationsdateien interpretiert. In Abschnitt 5.6.3 ist der Schlüssel der Konfiguration im Format der `.properties` Datei. Im Prototyp wird jedoch das YAML Format verwendet, da es übersichtlicher ist.

Konfigurations-Schlüssel	Beschreibung
spring.datasource.url	URL im JDBC Format für Datenbankverbindung der Fahrplandatenbank
spring.datasource.username	Benutzername für die Datenbank
spring.datasource.password	Passwort für die Datenbank
spring.datasource.driver-class-name	Fully Qualified Name der Klasse der als Treiber verwendet wird
spring.datasource.jpa.database-platform	Zu verwendender SQL Dialect für JPA (wird nicht benötigt, ist aber für Spring Data Meta Package nötig)
bernmobil.mappingrepository.datasource.url	URL im JDBC Format für die Datenbankverbindung der Mappingdatenbank
bernmobil.mappingrepository.datasource.username	Benutzername für die Mapping Datenbank
bernmobil.mappingrepository.datasource.password	Passwort für die Mapping Datenbank
bernmobil.jobrepository.datasource	Schema und Autorität der Datenbankverbindung für Spring Batch Daten
bernmobil.jobrepository.name	Pfad der Datenbankverbindung
bernmobil.jobrepository.driver-class-name	Full Qualified Name der Klasse der als Treiber die Spring Batch Datenbank verwendet wiederverwendet wird
bernmobil.staticsource.url	HTTP URL der statischen GTFS Daten
bernmobil.staticsource.folder	Ordner im Filesystem in welchem statische GTFS Daten entpackt werden können
bernmobil.amqp.fanout-queue	Name der Fanout Queue für Kommunikation zwischen Static Import und Front-End
bernmobil.history.size	Anzahl Versionen, welche in der Fahrplandatenbank gespeichert werden
bernmobil.history.timeout-duration	Länge in Minuten, ab wann eine laufende Instanz als nicht ordnungsgemäss abgeschlossen erkannt wird
bernmobil.batch.chunk-size	Grösse der Prozessschritte innerhalb der Spring Batch Steps
bernmobil.realtime-source.url	HTTP URL der Echtzeit GTFS Daten
bernmobil.locale.timezone	Zeitzone für die Umrechnung und Darstellung der Abfahrtszeiten
bernmobil.ruleset.delay	Differenz zwischen geplanter Abfahrt und tatsächlicher Abfahrt, in Minuten, ab der ein Verkehrsmittel als verspätet gilt

Tabelle 2: Konfigurationsmöglichkeiten der Anwendung

Umgebungsvariablen-Konfiguration

Die Konfiguration der Anwendung mittels Umgebungsvariablen wird durch das Betriebssystem oder die Laufzeitumgebung gemacht. Beim Erstellen des Docker Container können diese Werte mitgegeben werden, so dass sie beim Starten des Containers zur Verfügung stehen. Die Variablen mit dem Prefix „RDS“ sind von Elastic Beanstalk reserviert, und werden bei der Anbindung einer Datenbank an das Angebot automatisch gesetzt.

Variable	Beschreibung
RDS_HOSTNAME	Hostname des Datenbankservers der Fahrplandatenbank (db.example.com)
RDS_PORT	Portnummer der Datenbankanwendung
RDS_DB_NAME	Name der Datenbank
RDS_USERNAME	Benutzername für Datenbankverbindung
RDS_PASSWORD	Passwort für die Datenbankverbindung
RDS_MAPPING_HOSTNAME	Hostname des Datenbankservers mit Mappingdatenbank
RDS_MAPPING_PORT	Portnummer der Datenbankanwendung
RDS_MAPPING_DB_NAME	Name der Mapping Datenbank
RDS_MAPPING_USERNAME	Benutzername für die Datenbankverbindung
RDS_MAPPING_PASSWORD	Passwort für die Datenbankverbindung
RABBITMQ_HOSTNAME	Hostname des RabbitMQ Server
RABBITMQ_USERNAME	Benutzername für die Verbindung mit RabbitMQ Server
RABBITMQ_PASSWORD	Passwort für die Verbindung mit RabbitMQ Server

Tabelle 3: Einstellbare Umgebungsvariablen

Zeit Auswertung

In dieser Arbeit standen jeder Person 360 Arbeitsstunden zur Verfügung, um die geforderte Leistung von 12 ECTS Credits zu erfüllen.

Zweck	Oliviero Chiodo	Matteo Patisso
Evaluation	33 h	28 h
Architektur	62 h	72 h
Implementation	116 h	122 h
Cloud & Testing	85 h	75 h
Dokumentation	82 h	78 h
Meetings	28 h	28 h
Total	406 h	403 h

Persönliche Berichte

Matteo Patisso

Die Bachelorarbeit ist aus Engineering-Sicht die kompletteste und anspruchsvollste Arbeit, die ich bis heute geschrieben habe. Im Gegensatz zu meinen Vorarbeiten war diese Arbeit ein klassisches Projekt, in dem ich sehr viel im Studium Gelerntes umsetzen konnte.

Es gab reichlich viele Herausforderungen zu meistern, weshalb die Kommunikation zwischen allen Beteiligten, neben den technischen Skills, ein wichtiges Element der Arbeit war. Trotz gut formulierter Aufgabenstellung mussten viele Anforderungen anfänglich abgeklärt aber auch während der Entwicklung der Arbeit verfeinert werden. Bei diesen Herausforderungen wurden wir stetig sowohl von unserem BA-Betreuer als auch Industriepartner unterstützt. Die Zusammenarbeit war in diesem Sinne hervorragend.

Mit gewissen Bezeichnungen wie „Skalierbarkeit“ oder „Cloud-Fähigkeit“ konnte die Aufgabenstellung auf unterschiedliche Art und Weise interpretiert werden. Einer der grössten Herausforderungen in dieser Arbeit war es einen guten Mittelweg zwischen einfacher Erweiterbarkeit und Over-Engineering zu finden. Diese Herausforderung stellte mich und mein BA-Partner Oliviero vor viele Diskussionen, welche wir immer konstruktiv ausdiskutierten. Ich bin in dieser Hinsicht sehr dankbar, da ich der Meinung bin, dass dieses Verhalten nicht selbstverständlich ist und ich dabei sehr viel lernen konnte.

Mit dem Resultat der Arbeit bin ich sehr zufrieden. Sie rundet mein Studium positiv ab und war eine gute Rekapitulation des gelernten Stoffes. Ich fühle mich auch dank dieser Arbeit bereit für die Arbeitswelt und freue mich positiv auf dieses Projekt zurückschauen zu können.

Oliviero Chiodo

Als Pendler und langjähriger Wochenaufenthalter halte ich mich für einen erfahrenen ÖV-Benutzer. Das Thema dieser Arbeit hat mich deshalb sehr fasziniert und mich angespornt. Zudem war es mir ein grosses Anliegen, in der Bachelorarbeit einen kompletten Prototypen designen und entwickeln zu können, um Erfahrung in jedem Bereich des Software-Engineering zu sammeln.

Die Aufgabenstellung ist sehr offen formuliert und hat uns einen grossen Spielraum für eigene Ideen und Lösungen gelassen. Dies empfand ich Anfangs als etwas belastend, da es das Hauptziel der ersten Phase der Arbeit war, so schnell wie möglich einen funktionierenden Prototypen zu entwickeln. Zur gleichen Zeit Technologien und Frameworks zu evaluieren fiel mir sehr schwer. Ich fand es aber trotzdem sehr interessant und bereichernd, mich mit der Evaluation auseinander zu setzen und bin sehr zufrieden mit dem Resultat der Evaluation.

Besonders ansprechend fand ich ebenfalls, dass wir die Anwendung selbst in die Cloud bringen durften. Ich konnte dadurch nicht nur lernen, was für Tücken in einem solchen Ablauf auftreten können, ich konnte auch noch das erlangte Wissen im parallel besuchten Modul „Cloud Solutions“ anwenden. Ich finde diese Erfahrung und das gesammelte Wissen sehr wertvoll und bin überzeugt, dass mir diese Thematik auch in meiner weiteren Zukunft noch begleiten wird.

Mit dem entwickelten Prototyp bin ich sehr zufrieden. Wir konnten mit einer engen Zusammenarbeit mit dem Auftraggeber und der Betreuungsperson Probleme schnell identifizieren und beheben.

Ausserdem konnte ich die im Studium erlernten Techniken und Werkzeuge anwenden und sehr wertvolle Erfahrung sammeln. Nicht zuletzt durch diese Arbeit kann ich sehr viel Wissen über den Aufbau und die Verwendung von Frameworks, das Einlesen in Standards und die Verwendung einer Cloud, in meine zukünftige Anstellung mitnehmen.

Einverständniserklärung Publikation auf eprints.hsr.ch

SA

BA

Titel der Arbeit: _____

Team: _____

Betreuer: _____

Wir sind mit der Publikation unserer Arbeit auf eprints.hsr.ch einverstanden, sofern für diese Arbeit keine Geheimhaltungsvereinbarung unterzeichnet wurde.

Nach Bekanntgabe der Note haben wir die Möglichkeit innert 14 Tagen Einsprache zu erheben und das Einverständnis zur Publikation der Arbeit auf eprints.hsr.ch zurückzuziehen. In diesem Falle wird nur der Abstract publiziert.

Rapperswil,

Name(n)

Unterschrift(en)

Eigenständigkeitserklärung

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben

Rapperswil, den: _____

Der Student, Olivero Chiodo

Rapperswil, den: _____

Der Student, Matteo Patisso



HSR

HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Vereinbarung

1. Gegenstand der Vereinbarung

Mit dieser Vereinbarung werden die Rechte über die Verwendung und die Weiterentwicklung der Ergebnisse der Bachelorarbeit Plattform für Verkehrsinformationen bei Bernmobil von den Studierenden Oliviero Chiodo und Matteo Patisso unter der Betreuung von Prof. Dr. Luc Bläser in Zusammenarbeit mit dem Auftraggeber BERNMOBIL geregelt.


2. Urheberrecht

Die Urheberrechte stehen den Studierenden zu.

3. Verwendung

Die Ergebnisse der Arbeit dürfen sowohl von den Studierenden, von der HSR wie von BERNMOBIL nach Abschluss der Arbeit verwendet und weiterentwickelt werden.


Rapperswil, den 15.2.2017


.....
Die Studentin/der Student

Rapperswil, den 15.2.2017


.....
Die Studentin/der Student

Rapperswil, den 21.2.2017


.....
Der Betreuer / die Betreuerin der Bachelorarbeit

Bern, den 14.2.2017


.....
Im Namen des Auftraggebers BERNMOBIL

Sitzungsprotokolle

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	14. Februar 2017
Uhrzeit	15:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Einführung in Thematik
2. Definieren erster Schritte
3. Informationen zu Modulen (GTFS Static und Protocol Buffer)

Aufgaben

- Module und Architektur mit UML modellieren
- GTFS und SBB OpenTransportData analysieren

Bemerkungen

- Eine Komponente besteht aus einem Input Modul, einem Logikmodul und einem Anzeigemodul. Bei GTFS gäbe es zwei Inputmodule (für Static und Protocol Buffer), ein Logikmodul, welches die beiden Informationsquellen zusammenführt und ein Anzeigemodul welches die Daten an den Browser ausliefert
- Es dürfen auch Abhängigkeiten zwischen Modulen definiert werden. Gängige Architekturen bei Abhängigkeitsbeziehungen wären Peer und Core Modell.

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	21. Februar 2017
Uhrzeit	10:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Aktueller Stand der Arbeit (primär nach JIRA Dashboard)
2. Weiteres vorgehen für die folgende Woche

Aufgaben

- Architektur in UML definieren
- Einlesen in GTFS Static und Realtime

Bemerkungen

- Das Architektordiagramm hat klar Vorrang, und sollte so schnell wie möglich fertig sein. Damit kann ein erster Prototyp erstellt werden.

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	21. Februar 2017
Uhrzeit	16:00 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Server Infrastruktur
2. Organisation der Dokumentation
3. Besprechung Kickoff-Meeting mit Christan Loosli
4. Wie soll das Projektmanagement gemacht werden

Aufgaben

- Wöchentliche oder Zwei-Wöchentliche Reviews der Dokumentation
- Es soll schnell ein konkreter Prototyp entwickelt werden, ohne der Modularität und der Skalierung grosse Beachtung zu schenken
- Genauere Definition von Skalierung erarbeiten

Bemerkungen

- Ein möglicher Prototyp wäre die Anzeige an einem Informationsboard beispielsweise am Bahnhof oder an einer Bushaltestelle
- Für die Skalierung sollten die Regeln der verwendeten Frameworks eingehalten werden (beispielsweise Spring oder Play)

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	28. Februar 2017
Uhrzeit	10:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Zugriff auf GTFS Realtime Daten
2. Architektur und Domainmodell
3. Prototyp besprechen
4. Meilensteine anschauen
5. Eventuelle Verwendung von Scala
6. Präferenz einer Datenbank

Aufgaben

- Prototyp mit GTFS Static und Realtime ohne Module entwickeln
- Domainmodell der verschiedenen Module anhand ÖV-Plus Anzeiger erstellen
- Einlesen in Container(Docker) und Verteilungstechnologie(Akka, RabbitMQ)

Bemerkungen

- Meilensteine:
 - Prototyp Spring
 - Verteiler Prototyp mit Docker
 - Service zu Broker ausbauen

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	01. März 2017
Uhrzeit	12:30 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Prototyp besprechen
2. Weiteres Vorgehen (Prototyp)
3. Informieren Meeting mit Christian Loosli

Aufgaben

- Prototyp fertigstellen
- Projektplan erstellen
- Informieren Amazon (und andere Anbieter) Services
- Architekturdokumentation
- Organisieren Testumgebung
- Anforderungen definieren

Bemerkungen

- Anstelle von Messaging normales HTTP

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	8. März 2017
Uhrzeit	12:45 Uhr
Teilnehmer	Luc Bläser Oliviero Chiodo

Traktanden

1. Fortschritt letzter Woche
2. Unvollständige Daten von ZVB
3. Anfragen Projektplan-, Dokumentation- und Source-Code Review

Aufgaben

- Prototyp fertigstellen um weiteres Vorgehen zu bestimmen
- Deployment auf Amazon
- Projekt kontinuierlich dokumentieren

Bemerkungen

- Vorherige Analyse von Amazon und Technologien (Load Balancing etc.)
- Die Konfiguration des Anzeigemanagers wurde erwähnt, wie man das konsistent halten soll.

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	15. März 2017
Uhrzeit	12:30 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Fortschritt des Prototypen
2. Diskussion der Architektur

Aufgaben

- Prototyp unverteilt fertig schreiben
- Datenbankschema anpassen, von GTFS lösen
- Architektur dokumentieren

Bemerkungen

- Die Datenbank soll bereits eine aufbereitete Form der Daten enthalten, sodass die Anzeige der Daten so wenig Aufwand wie möglich ist.

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	15. März 2017
Uhrzeit	13:15 Uhr
Teilnehmer	Christian Loosli Thomas Bodenmann Matteo Patisso Oliviero Chiodo

Traktanden

1. Fortschritt der Arbeit besprechen
2. Modularität diskutieren
3. Calendar und Calendar Dates

Aufgaben

- Recherche über Datenbus zur Kommunikation der Module (IPC als Beispiel)
- Kommunikation zwischen den Containern
- Überlegen, wie Daten fließen können, die nicht in die Datenbank gehören
- Recherche, ob Hibernate eine gute Lösung ist
- Ausnahmefälle überlegen und testen
- Tickets in JIRA zu Modularität und Docker erstellen

Bemerkungen

- Wie greift der Container auf die Datenbank zu (spezifisch oder generisch)
- Der Kommunikationsfluss und der Datenfluss soll getrennt werden
- Szenarien des Ausfalls müssen in die Evaluation einer Kommunikationstechnologie einfließen

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	5. April 2017
Uhrzeit	13:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Aktueller Stand
2. Modelle besprechen
3. Update- und Fehlerstrategie

Aufgaben

- Evaluation der Produkte
- Tickets für Messaging
- Messaging und Komponentenmanagement
- Update der Daten aus Standard entnehmen

Bemerkungen

- Kleines Dashboard für registrierte Komponenten
- Technologievergleiche

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	5. April 2017
Uhrzeit	9:30 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Aktueller Stand
2. Update der bestehenden Daten
3. Error Handling auf Container Ebene

Aufgaben

- Bericht fortführen
- Error Handling Idee entwickeln
- Skalierbarkeit recherchieren

Bemerkungen

- Updates können inkrementell sein. Wenn der Import aber nicht lange dauert, dann können die Daten auch komplett gelöscht werden
- Testplan für die Skalierung erstellen und umsetzen

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	11. April 2017
Uhrzeit	10:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Import Datenbank Trennung
2. Realtime Import Performance Verbesserung
3. Amazon Konto
4. Load-Tests / Profiling
5. AMQP Evaluation begonnen
6. Realtime ID und Static ID stimmen nicht überein

Aufgaben

- In Arbeit hervorheben: Skalierung und andere Datenquellen
- Unit- und Integrationstest
- Dokumentation bis Ende Woche an Christian senden

Bemerkungen

- Evaluation in Ticket schreiben

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	18. April 2017
Uhrzeit	13:00 Uhr
Teilnehmer	Christian Loosli Oliviero Chiodo

Traktanden

1. Bericht besprechen
2. Amazon deployment anschauen
3. Aktueller Stand Arbeit

Aufgaben

- Deployment auf Amazon abschliessen
- Lasttest Resultate vorbereiten evtl. mit Demo
- Bericht weiter schreiben

Bemerkungen

- Struktur des Berichtes ist gut
- Korrekturleser suchen
- Kapitel abschliessen und Korrekturlesen lassen
- Bei 1:1 Zitaten MUSS Quelle angegeben werden
- Literatur in Bericht verweisen
- In Grafiken bessere Beschreibungen
- Analyse und Umsetzung besser trennen

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	19. April 2017
Uhrzeit	12:30 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Aktueller Fortschritt
2. Deployment auf Amazon
3. Problem mit Datenbankverbindung und Hibernate
4. Skalierungsplanung

Aufgaben

- Skalierungskonzepte evaluieren
- Probleme mit der Skalierung evaluieren
- Backlog sammeln

Bemerkungen

- Beim Sammeln des Backlogs können Features, welche noch implementiert werden können zusammengetragen werden
- Konzept der Skalierung ermitteln um zu sehen, wie der Prototyp skalieren kann

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	28. April 2017
Uhrzeit	9:30 Uhr
Teilnehmer	Luc Bläser Ivan Bütler Matteo Patisso Oliviero Chiodo

Traktanden

1. Demo der Anwendung
2. Erklärung der Struktur
3. Demo Skalierung

Aufgaben

- Testpläne für stationäre Abfahrtsanzeiger
- Konzept für Update ausfeilen
- Kommunikation zwischen Komponenten überdenken

Bemerkungen

- Implementierung der Kommunikation würde System eventuell verkomplizieren
- Wenn Kommunikation eingesetzt, beweisen, dass System noch skaliert

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	2. Mai 2017
Uhrzeit	13:00 Uhr
Teilnehmer	Christian Loosli Oliviero Chiodo

Traktanden

1. Rückmeldung von Demo bei Herr Bläser
2. Aktueller Stand
3. Dokumentation und Lasttests

Aufgaben

- Kommunikation mit Herr Bläser besprechen
- Bericht fertig schreiben

Bemerkungen

- Kommunikation ist für Bernmobil teil der Anforderung. Es wäre sehr schade, wenn es diesen Teil nicht in die Arbeit schaffen würde

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	10. Mai 2017
Uhrzeit	12:30 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Stand der Arbeit
2. Struktur der Dokumentation
3. Synchronisation zwischen verschiedenen Import Instanzen

Aufgaben

- Beide Importer entweder in ein Prozess oder mit Locking in DB
- Sichtbarkeit in der Datenbank, Atomarität der Lese- und Schreiboperationen

Bemerkungen

- Beim Locking muss eine geeignete Lösung gefunden und abgeklärt werden, ob Locking überhaupt nötig ist.

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	16. Mai 2017
Uhrzeit	13:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Aktueller Stand der Arbeit
2. Bericht

Aufgaben

- Test-Coverage erhöhen
- JavaDoc zu Methoden schreiben
- Deployment für Demo bei Bernmobil
- Bericht weiterschreiben
- Installationsanleitung mit abgeben

Bemerkungen

- Test-Coverage der Logik auf ein akzeptables Mass bringen (70 - 90 %)
- Christian möchte bei Bernmobil eine Demo durchführen. Bis Mittwoch eine funktionierende Instanz bereitstellen
- JavaDoc für alle Methoden und Klassen (nahe 100%)
- Eine Installationsanleitung schreiben, in der das Verfahren dokumentiert wird, wie auf Amazon deployed werden kann.

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	17. Mai 2017
Uhrzeit	12:30 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Stand der Arbeit
2. Wichtige Punkte in der Dokumentation

Aufgaben

- Bericht schreiben
- Zustandsdiagramm für Synchronisation
- Zustand während Update (Isolation Level)
- Fehlerfälle dokumentieren

Bemerkungen

- Zustandsdiagramm der Kommunikation erstellen, zwischen Static Import und Anzeige
- Mutual Exclusion zwischen Import Instanzen evaluieren (ist es überhaupt nötig)
- Der Zustand in der Update History muss atomar sein

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	23. Mai 2017
Uhrzeit	13:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Stand der Arbeit
2. Refactorings
3. Dokumentation

Aufgaben

- JavaDoc
- TestCoverage

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	24. Mai 2017
Uhrzeit	09:30 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Code Freeze
2. Dokumentation

Aufgaben

- Dokumentation fortführen

Bemerkungen

- Erste Fassung der Dokumentation schnell schreiben, um Feedback einholen zu können

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	30. Mai 2017
Uhrzeit	13:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Dokumentation

Aufgaben

- Dokumentation beenden
- JavaDoc und Test Coverage

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	31. Mai 2017
Uhrzeit	09:30 Uhr
Teilnehmer	Luc Bläser Matteo Patisso Oliviero Chiodo

Traktanden

1. Wichtige Punkte in Dokumentation
2. Demo des fertigen Prototyps
3. Code Review

Aufgaben

- Dokumentation beenden
- Code Review umsetzen

Bemerkungen

- In der Dokumentation sollten bestimmte Punkte welche nicht auf der Hand liegen erläutert werden: Wieso werden drei Komponenten verwendet, Wieso werden die Updates in einem 30 Sekunden Intervall geladen, Schnittstellen und Komponenten, Architektur und Packages, vier Projekte beschreiben.

Sitzungsprotokoll ViBe

Ort	Hochschule für Technik Rapperswil
Datum	6. Juni 2017
Uhrzeit	13:00 Uhr
Teilnehmer	Christian Loosli Matteo Patisso Oliviero Chiodo

Traktanden

1. Korrektur technischer Bericht
2. Stand JavaDoc/Testing

Aufgaben

- Testcoverage
- JavaDoc

Bemerkungen

- Stilfehler in Bericht verbessern