

Facettierte Suche

als Tool für die Explorative Datenanalyse im Web

Andreas Berchtold und Karin Duss

Bachelorthesis

Windisch, 28. August 2020

Studiengang

iCompetence

Betreuer

Professor Dierk König

Dr. Dieter Holz

Experten

Marco Sanfratello

François Martin

Abstract

Die facettierte Suche ist ein wichtiges Werkzeug für die explorative Datenanalyse im Bereich Data Science. In der vorliegenden Bachelorarbeit wird eine Machbarkeitsstudie durchgeführt, in welcher untersucht wird, wie die Facettensuche mit einer Webapplikation für strukturierte Datensätze umgesetzt werden kann. Das Thema wird anhand der Entwicklung eines Prototyps erforscht, der für grosse Datensätze effizient Resultate liefert. Der Fokus der Entwicklung liegt auf der Definition einer flexiblen Datenstruktur, die es ermöglicht, den Prototyp um weitere Facettenarten und alternative Visualisierungen zu erweitern. Die Arbeit dient als Grundlage für komplexere Anwendungen der facettierten Suche. Die Erkenntnisse aus der Entwicklung des Prototyps sind in diesem Bericht zusammengefasst.

Inhaltsverzeichnis

Zusammenfassung	5
1 Einführung	6
1.1 Ausgangslage	6
1.2 Problemstellung	6
1.3 Ziel	6
1.4 Umfang	7
1.5 Methodik	7
1.6 Konzepterläuterung	8
I Konzeptergebnis	9
2 Facetten	10
2.1 Facettentypen	10
2.2 Suche	12
3 Datensatzspeicherung	14
3.1 JavaScript Datenstruktur	14
3.2 Client	14
3.3 Datenbanken	15
4 Client-Server	19
4.1 Optimierungen der Serverabfragen	19
4.2 Datenbankoptimierungen	23
II Umsetzungsergebnis	25
5 Requirements	26
5.1 Funktionale Anforderungen	26
5.2 Nicht funktionale Anforderungen	27
5.3 Out of Scope	28
6 Designentscheide	29
6.1 Innere Datenstruktur	29
6.2 Datenspeicher	34
6.3 Modulsystem	35
6.4 Verhalten bei Selektion	36

INHALTSVERZEICHNIS

7	Architektur	40
7.1	Client-Server	40
7.2	Client	40
8	Client	43
8.1	MainController	44
8.2	Module	45
8.3	Facet	46
8.4	Asynchrone Aktualisierung	47
9	Server	49
9.1	Datenbankeinbindung	49
9.2	Optimierung der Datenbankperformance	51
9.3	Probleme	52
10	User Interface	54
10.1	Interaktionen	54
10.2	Usability	54
11	Testing	56
11.1	Testumgebung	56
11.2	Unit	56
11.3	End-to-End	57
11.4	API	58
12	Endprodukt	59
12.1	Nutzung	59
12.2	Erweiterbarkeit	59
III	Schluss	60
13	Fazit	61
14	Ausblick	62
14.1	Usability des Prototyps	62
14.2	Weitere Facettenimplementierungen	63
14.3	Performance	64

INHALTSVERZEICHNIS

IV Appendix	66
Ehrlichkeitserklärung	67
Literaturverzeichnis	70
Abbildungsverzeichnis	71
Tabellenverzeichnis	72
Glossar	73

Zusammenfassung

Die Entwicklung des Internets und der Webtechnologie ermöglicht nicht nur bessere Suchmaschinenenergebnisse, sondern eröffnet verschiedene Alternativen zur klassischen schlüsselwortbasierten Suche. Datensätze ab einer gewissen Grösse können nicht mehr mittels einer Tabelle oder Textsuche untersucht und analysiert werden. Für grosse Datensätze eignet sich der Ansatz der explorativen Datenanalyse, die immer mehr Beliebtheit findet. Eine Form davon ist die facettierte Suche, die es dem Datenanalysten ermöglicht, die Ergebnismenge schrittweise einzugrenzen.

Obwohl diese Art der Suche auch im kommerziellen Webbereich häufig eingesetzt wird, existiert noch keine generalisierte Open Source Lösung. Das Problem ist, dass die clientseitige Logik (innere Datenstruktur) für jede Webapplikation, die eine Facettensuche integrieren möchte, von Neuem aufgebaut werden muss. In dieser Bachelorarbeit haben wir ausgehend von einem konkreten Beispieldatensatz eine Variante einer solchen inneren Datenstruktur erarbeitet. Schrittweise haben wir die erforderliche Logik erarbeitet, Limitationen ergründet und anschliessend eine verallgemeinerte Lösung entwickelt.

Ein Ziel bei der Entwicklung einer effizienten facettierten Suche ist es, eine optimale Verteilung zwischen clientseitiger und serverseitiger Datenverarbeitung zu finden. Um in der Kommunikation mit der Datenbank zu lange Latenzzeiten zu vermeiden, braucht es eine clientseitige Zustandsverwaltung. Dazu gehören der logische Zustand der Facetten, die aktuellen Filter Queries und die Beschreibung der Facetten.

Operationen wie Filtern, Zählen und Sortieren sind clientseitig effizient implementierbar, bei grossen Datenmengen ist der clientseitige Speicher jedoch limitiert. Ab einer gewissen Anzahl Daten ist eine externe Speicherung der Daten daher unabdingbar. Die Wahl der Datenbank ist für die Effizienz entscheidend, wobei SQL- und NoSQL-Datenbanken unterschiedliche Vor- und Nachteile aufweisen.

Das Auslagern der aufwändigen Operationen auf die Datenbank kann zu ungünstigen Zuständen auf dem Client führen, wenn die Kommunikation nicht synchronisiert wird. Ebenfalls zu beachten ist, dass die Wahl der Datenbank durch die Clientlogik nicht eingeschränkt werden sollte.

1 | Einführung

Die Facettensuche ermöglicht eine explorative Datenanalyse, bei welcher der Benutzer das Resultat nach mehreren Dimensionen eingrenzen kann. Diese Art von Suche ist flexibel und ermöglicht dem Benutzer eine interaktive Exploration komplexer Datensätze. Die Implementierung ist im Gegensatz zur klassischen Textsuche jedoch auch zeitintensiver. Aus diesem Grund wäre ein Open Source Konzept nützlich, das die Logik der Facettensuche für beliebige Datensätze abstrahiert und generell einsetzbar macht. In dieser Arbeit erarbeiten wir anhand der Entwicklung eines Prototyps, wie eine solche Lösung umgesetzt werden kann und welche Schwierigkeiten auftreten.

1.1 Ausgangslage

Data Science ist in der heutigen Zeit ein wichtiger Bestandteil der Informatik und kann in vielen Bereichen angewendet werden. Dabei möchten Datenanalysten oft Informationen und neue Erkenntnisse aus vorliegenden Daten gewinnen können. Insbesondere sollen Zusammenhänge zwischen einzelnen Daten gefunden werden. Teilweise möchte man aber auch Datenbestände auf ihre Korrektheit prüfen.

Um grosse Datenmengen zu durchsuchen eignet sich die explorative Datenanalyse in Form der Facettensuche. Dabei werden die Daten nach Dimensionen gegliedert und anhand gegebener Ontologien strukturiert.

1.2 Problemstellung

Für die explorative Datenanalyse mittels Facettensuche im Web gibt es noch keine Open Source Lösung, die allgemein einsetzbar ist. Dies führt zu einem zeitlichen Mehraufwand und wiederholtem Lösen wiederkehrender Probleme, was die Ressourcen für Weiterentwicklungen einschränkt. Im Gegensatz zur textbasierte Suche muss nicht nur die Gesamtzahl der Treffer aktualisiert werden, sondern auch die Anzahl für jeden Facettenwert. Zudem ist die Filterfrequenz höher im Gegensatz zur klassischen Suche.

1.3 Ziel

Das Ziel ist ein webbasiertes Werkzeug, das Entwickler als Grundlage für ihre Implementierung der Facettensuche verwenden können. Auf dieser Basis kann ein Entwickler sich auf Spezialitäten der Facettensuche konzentrieren (z. B. eine Facette, die kartographisch dargestellt wird).

1.4 Umfang

Im Rahmen des Projekts soll ein Prototyp erstellt werden, der die abstrahierte clientseitige Logik der Facettensuche implementiert. Dieser soll es ermöglichen, Datensätze von 0.5 bis 1.5 Millionen Einträge effizient zu untersuchen. Der Prototyp soll erweiterbar sein, so dass zusätzliche Facettentypen implementiert werden können und die Datenbank austauschbar ist.

Der Prototyp sollte Folgendes enthalten:

- Ein User Interface um den Datensatz zu untersuchen
- Eine generalisierte Clientlogik
- Eine Schnittstelle zum Backend mit API

1.4.1 User Interface

Der Benutzer kann einen Datensatz mithilfe von regulären Facetten untersuchen. Die Suchresultate werden dem Benutzer in Form einer tabellarischen Darstellung angezeigt.

1.4.2 Schnittstelle

Die Schnittstelle vom Frontend zum Backend (Datenbank) ist mit einer API definiert, damit zukünftige Entwickler das Backend ergänzen können.

1.4.3 Fokus

Der Fokus liegt auf der inneren Datenstruktur des Clients und nicht auf dem Backend. Datenbank- und Servertechnologie können in einem weiteren Projekt angepasst werden.

1.5 Methodik

Um das definierte Ziel zu erreichen, haben wir ein schrittweises Vorgehen gewählt. In jedem Schritt wurde das nächste wichtige Teilproblem definiert, das zu lösen ist. Ein solches Teilproblem wurde dann in drei Phasen bearbeitet:

1. Recherche-Phase: Welche Varianten gibt es, um das Problem zu lösen?
2. Praktische Umsetzung: Implementation einer Variante mit einem konkreten Beispieldatensatz
3. Generalisierung: Verallgemeinern der Umsetzung, um möglichst datenunabhängig zu sein.

Ebenfalls war es uns wichtig, noch im Laufe der Arbeit interessante Schwerpunkte setzen zu können. Um der Agilität dieses Ansatzes gerecht zu werden, wurde mit Sprints und User Stories gearbeitet. Am Anfang jedes Sprints wurde der Schwerpunkt bestimmt und die nächsten zu lösenden Probleme definiert.

1.6 Konzepterläuterung

Der Hauptteil der Dokumentation besteht aus zwei Teilen. Der erste Teil bezieht sich auf die Definitionen rund um das Thema Facettensuche und die Rechercheergebnisse. Im zweiten Teil wird auf die konkrete Implementation des Prototyps eingegangen, welche Konzepte gewählt wurden und welche Probleme dabei aufgetreten sind. Im Schlussteil wird der aktuelle Stand analysiert und ein Ausblick gegeben, welche Herausforderungen in der Weiterentwicklung auftreten und welche weiteren Schritte vorzunehmen wären.

Teil I

Konzeptergebnis

2 | Facetten

Das erste Konzept der Facette stammt von S.R. Ranganathan. Sein Ziel war, multidimensionale Eigenschaften von Dokumenten zu beschreiben und eine Bibliotheksklassifikation zu entwickeln. Je nach verwendeter Literatur kann sich die genaue Definition einer Facette unterscheiden [1].

Für diesen Bericht gilt die folgende Definition: Eine Facette ist eine Gruppe von Facettenwerten, die sich alle auf eine spezifische Dimension eines Subjekts beziehen. Jeder Facettenwert repräsentiert entweder ein Attribut oder eine Kategorie, wobei eine Kategorie eine Sammlung von Attributen ist.

2.1 Facettentypen

Im Rahmen dieser Thesis wurden anhand von Beispieldatensätzen verschiedene Facettentypen identifiziert und analysiert. Das Ziel war es, eine einheitliche Facettenbeschreibung für verschiedene Typen zu extrahieren. Nachfolgend werden die zwei extrahierten Typen beschrieben.

2.1.1 Flache Facette

Beim ersten Typ handelt es sich um eine einfache Auflistung der einzelnen Attribute einer Dimension. Auf einen konkreten Datensatz bezogen, in dem eine Spalte eine Dimension repräsentiert und die Zeilen jeweils die dazugehörigen Attribute enthalten, ist dies die direkte Zuweisung eines Spaltenwertes zu einem Facettenwert.

id	Surface	Court
17172	Hard	Outdoor
6543	Grass	Outdoor
7784	Hard	Indoor

Surface	
<input type="checkbox"/> Hard	2
<input type="checkbox"/> Grass	1

Abbildung 2.1: Beispiel für eine flache Facette

Die zweite Form von flachen Facetten tritt auf, wenn für ein Attribut eine **polydimensionale** Klassifikation vorliegt. Auf einen tabellarischen Datensatz bezogen, bedeutet dies, dass eine Zeile in einer einzelnen Zelle mehrere Facettenwerte enthalten kann oder die Facette auf mehrere Spalten verteilt ist. Zur Veranschaulichung: Ein Film kann mehreren

2.1. FACETTENTYPEN

Genres zugeordnet werden. Die Genres sind entweder durch Trennzeichen voneinander unterscheidbar oder sind in mehreren Spalten gespeichert (z. B. „Genre 1“, „Genre 2“). Werden die Anzahl Einträge pro Genre aufsummiert, sind die jeweiligen Totale kleiner oder gleich der Anzahl Zeilen des Datensatzes. Summiert man die jeweiligen Totale, muss dieser Wert grösser oder gleich der Anzahl Zeilen sein, vorausgesetzt jeder Film gehört mindestens einem Genre an.

Movie id	Title	Genre
17172	The Shawshank Redemption	Drama
56384	Black Swan	Drama, Thriller
12244	Parasite	Comedy, Drama, Thriller

Genre	
<input type="checkbox"/> Drama	3
<input type="checkbox"/> Thriller	2
<input type="checkbox"/> Comedy	1

Abbildung 2.2: Beispiel für eine flache polydimensionale Facette

2.1.2 Hierarchische Facette

Facetten, deren Facettenwerte in Kategorien eingeordnet werden können, werden als hierarchische Facetten bezeichnet. Hierarchien treten dabei auf verschiedene Arten auf. In den nachfolgenden Abschnitten werden die beiden unterschiedlichen Arten beschrieben. Analog zum Abschnitt 2.1.1 beziehen sich die Erläuterungen auf tabellarische Datensätze.

Gruppierungen

Eine Hierarchie kann definiert werden, indem ein Attribut als Oberkategorie und ein zweites Attribut als Unterkategorie definiert wird.

id	Surface	Location
1717	Hard	Sydney
6543	Grass	London
7784	Hard	Basel
8897	Hard	Paris
3345	Grass	Lyon

Location	
▶ <input type="checkbox"/> Australia	1
▼ <input type="checkbox"/> Europe	3
▶ <input type="checkbox"/> England	1
▶ <input type="checkbox"/> France	2
▼ <input type="checkbox"/> Switzerland	1
<input type="checkbox"/> Basel	1

Abbildung 2.3: Beispiel für eine hierarchisch gruppierte Facette

In den Unterkategorien entsteht dabei nicht zwingend eine Partition, denn Unterkategorien können verschiedenen Oberkategorien zugeordnet sein. Dieser Typ funktioniert analog für Hierarchien, die aus mehr als zwei Spalten zusammengesetzt sind. Die zwei Spalten „Land“ und „Ortschaft“ können zu einer Hierarchie zusammengefasst werden, bei der man die Ortschaften gruppiert nach Land betrachten kann.

2.2. SUCHE

Das Extrahieren von Informationen zum Aufbau einer Hierarchie aus einer einzelnen Spalte ist eine weitere Methode. Möglich ist dies, wenn eine Spalte Daten enthält, die durch Trennzeichen strukturiert sind. Dies erlaubt es, die Teilinformationen einzeln auszulesen und als eigene Spalten zu interpretieren. Die Hierarchie entsteht damit wieder aus mehreren Spalten, wie im Abschnitt 2.1.1 beschrieben. Aus einem Datum im Format „YYYY-MM-DD“ können das Jahr, der Monat und der Tag ausgelesen und auf unterschiedliche Arten kombiniert werden.

Bereiche

Neben den gruppierten hierarchischen Facetten gibt es noch eine zweite Art. Bei dieser müssen nicht zwingend Ober- und Unterkategorien vorhanden sein.

id	Rank	Location
1756	1	Adelaide
6543	2	London
7784	3	Basel
1717	4	Paris

Rank	
<input type="checkbox"/> 1-3	3
<input type="checkbox"/> 4-10	6

Abbildung 2.4: Beispiel für eine hierarchisch Bereichs-Facette

Die Facettenwerte entsprechen Kategorien und nicht Attributen. Jede Kategorie wird mithilfe einer Bereichsangabe und nicht der Auflistung der einzelnen Werte aus dem Datensatz angegeben. Ein typisches Beispiel hierzu ist die Zusammenfassung von Platzierungen. Anstatt jeden Rang einzeln auf einer flachen Facette aufzulisten, werden diese zu interessanten Bereichen zusammengefasst. Man möchte beispielsweise nur das Podium (1-3) oder die besten Zehn (1-10) untersuchen.

2.2 Suche

Die Facettensuche ist eine Form der explorativen Datenanalyse. Deshalb wird nachfolgend zuerst die explorative Datenanalyse und danach die Facettensuche erklärt.

2.2.1 Explorative Datenanalyse

Suchaktivitäten können in drei Kategorien eingeteilt werden: Lookup, Learn und Investigate [2]. Lookup beschreibt die Aufgabe, eine konkrete, vordefinierte Information zu finden. Die explorative Suche findet in erster Linie in den Bereichen Learn und Investigate Anwendung. Bei Learn geht es um Wissensaneignung, Interpretation und Vergleichen. Die Aktivität Investigate umfasst die Analyse, Evaluation und Entdeckung von Ergebnissen.

Die Stärke der explorativen Suche liegt darin, dass nicht von Beginn an alle Parameter bekannt sein müssen. Der Ablauf einer Suche folgt üblicherweise dem folgenden Schema, wobei die Anzahl Iterationen variiert. Zu Beginn wird eine Suchanfrage formuliert. Die

2.2. SUCHE

dadurch erhaltenen Ergebnisse werden untersucht und verglichen. Mit den gewonnenen Erkenntnissen kann die Suchanfrage angepasst und eine bessere Resultatmenge gefunden werden [2].

Die explorative Suche eignet sich, um einen Datensatz anhand bestimmter Kriterien zu untersuchen. Durch die Veranschaulichung der Trefferanzahl können Abhängigkeiten und Häufungen erkannt werden. Diese Art von Analyse eignet sich insbesondere, um Ausreisser in Datensätzen zu finden und somit Mess- oder Tippfehler zu erkennen.

2.2.2 Facettensuche

Die Facettensuche, auch facettierte Suche, facettierte Navigation oder facettierte Browsing genannt, erlaubt es **Suchenden**, schrittweise Facetten und Facettenwerte auszuwählen, mit dem Ziel die Ergebnismenge einzuschränken und Einträge zu entfernen, die nicht relevant sind (Noise). Daher ist sie eine interaktive Suchart. Vergleicht man die Facettensuche mit der Volltextsuche, besteht ein klarer Vorteil im niedrigeren erfordernten Vorwissen über das Datenschema. Zudem kann ein Benutzer leere Suchergebnisse (Silence) mittels der Trefferanzeige für Facettenwerte umgehen [1]. Um die Facettensuche anwenden zu können, müssen die Daten strukturiert sein.

Anwendung findet die facettierte Navigation vor allem im E-Commerce-Bereich und in der professionellen Datenanalyse. In beiden Situationen gilt das gleiche Suchprinzip, jedoch unterscheidet sich das User Interface. Während die Usability für einen Webshop möglichst schnell verständlich sein muss und die Trefferanzahl pro Merkmal meist weniger wichtig ist als die Gesamtzahl der verbleibenden Artikel, sind die Treffer pro Attribut für einen Datenanalysten relevant. Bei einem Datenanalysten kann man eine gewisse Erfahrung mit dem Interface voraussetzen. Die Abbildung 2.5 zeigt ein Beispiel einer Benutzeroberfläche für die Facettensuche im Datenanalysebereich.

Position	Zip	City	Type	Nomina
1213169	55129	Mainz	Windkraft	600.0
1213172	55129	Mainz	Windkraft	600.0
1213173	55129	Mainz	Windkraft	600.0
1213175	55129	Mainz	Windkraft	600.0
1213226	55129	Mainz	Windkraft	1.1
1213398	55129	Mainz	Windkraft	2000.0
1213404	55129	Mainz	Windkraft	2000.0
1213453	55129	Mainz	Windkraft	800.0
1213682	55129	Mainz	Windkraft	2000.0
Aggregate Fil...				Total
Total:				Different Cities: 1 Different Types: 1 10/1377475

Abbildung 2.5: Beispiel UI für Facettensuche im Datenanalysebereich

Aufgrund der unterschiedlichen Anforderungen ist es wichtig festzulegen, für welche Benutzergruppe die Suche entwickelt wird. Diese Arbeit richtet sich an Datenanalysten.

3 | Datensatzspeicherung

Ein Ziel ist es, dass der Benutzer des Endproduktes einen beliebigen Datensatz für die Facettensuche auswählen kann. Der zu untersuchende Datensatz soll von der Webapplikation eingelesen werden können. Obwohl diese Anforderung nicht im Umfang dieser Arbeit liegt (Abschnitt 1.4), muss ein Konzept für die Speicherung der Daten definiert werden. In diesem Kapitel wird aufgezeigt, wo man den Datensatz abspeichern kann und welche Vor- und Nachteile die unterschiedlichen Möglichkeiten aufweisen. Schlussendlich wird dargelegt, welche Variante in diesem Projekt verwendet wurde und warum.

3.1 JavaScript Datenstruktur

Eine einfache Variante ist es, die Daten in einer JavaScript-Datenstruktur, also einem Objekt oder Array, abzuspeichern. Aufgrund der erwarteten tabellarischen Struktur der Datensätze wäre ein Array passender. Es wäre auch möglich, eine eigens definierte Datenstruktur zu verwenden. Dies ist nicht nötig, weil JavaScript ein Array bereits effizient durchsucht. Ausserdem werden Funktionen wie Filtern, Mapping und Sortieren zur Verfügung gestellt, welche ebenfalls effizient implementiert sind.

Vorteile

- Einfache Implementierung
- Effiziente Sortier- und Filterfunktionen

Nachteile

- Datensatz ist bei einem Reload nicht mehr verfügbar
- Ist nur bis zu einer gewissen Datensatzgrösse möglich

3.2 Client

MDN Web Docs beschreibt drei weitere Optionen, wie man Daten clientseitig speichern kann.

Die erste Option ist die **Web Storage API**. Diese ist auf das Speichern von kleinen Datenmengen ausgelegt. Daher eignet sie sich für das Speichern von Benutzernamen, Anmeldestatus oder benutzerspezifischen Einstellungen [3].

3.3. DATENBANKEN

Die zweite clientseitige Speichermöglichkeit ist die **Cache API**. Diese ist darauf ausgelegt, HTTP-Responses zu spezifischen Requests abzuspeichern und ist sehr hilfreich, um Websitedaten offline zu speichern [3].

Die **IndexedDB API** integriert im Browser ein komplettes Datenbanksystem, ausgerichtet auf die Speicherung von komplexen Daten. Es ist eine JavaScript-basierte objektorientierte Datenbank. Verwendungsbeispiele für die IndexedDB sind das Deponieren von Kundendaten. Es ist aber auch möglich Audio- oder Videodateien zu speichern [3]. Der maximale Speicherplatz ist dynamisch und vom Datenspeicher des Clients abhängig. Das globale Limit entspricht 50 % des verfügbaren Speicherplatzes. Der Nachteil der IndexedDB API liegt darin, dass der Entwickler den Datenerhalt nicht garantieren kann. Es liegt in der Macht des Benutzers, Daten einer Website über die Browsereinstellungen zu entfernen. Des Weiteren werden Daten im Private Browsing Modus am Ende jeder Session gelöscht [4].

Vorteile

- Keine Serveranbindung nötig
- Offline Verwendung möglich, wenn richtig implementiert

Nachteile

- Implementierung je nach Variante kompliziert
- Browserabhängig
- Entwickler kann nicht kontrollieren, wann Daten gelöscht werden

3.3 Datenbanken

Als dritte und letzte Option ist eine Datenbank für die Speicherung des Datensatzes in Betracht zu ziehen. Datenbanken sind darauf ausgelegt, grössere Datenmengen zu speichern und sie bringen eine Struktur in die Daten. Daten können für eine unbeschränkte Zeit gespeichert werden. Je nach Datenbankart werden verschiedene Operationen unterstützt: CRUD, Filtern, Sortieren, etc. Gewisse Operationen sind abhängig vom Ansatz effizienter. Die Datenbankarten weisen auch Vor- oder Nachteile bei der Skalierung auf, da der Datenbestand jeweils unterschiedlich gespeichert wird und der Zugriff auf die Daten ebenfalls verschieden ist. Daher ist es wichtig, die Datenbank den Anforderungen der Applikation entsprechend auszuwählen.

3.3.1 Relationale Datenbanken

In relationalen Datenbanken werden die Daten in verschiedenen Tabellen gespeichert, die untereinander in Beziehung stehen. Jede Tabelle besteht aus Spalten und Zeilen, wobei jede Zeile einem Tupel entspricht und jede Spalte die Werte eines Attributs darstellt. Dies führt zu einer intuitiven Struktur [5].

Vorteile

- Die **tabellarische Datenstruktur** kann sich ein Mensch gut vorstellen.
- Durch Normalformen wird eine **geringere Redundanz** erreicht.
- SQL ist eine weitgehend **standardisierte Datenbanksprache**.
- Durch die Struktur und Organisation der Daten sind **Abfragen effizient**.

Nachteile

- Nicht für die Handhabung **unstrukturierter Daten** geeignet, da sich nicht alle Datentypen in einer Tabellenstruktur abbilden lassen.
- Die Daten werden durch die Normalformen **segmentiert** und Informationen auf separate Tabellen aufgeteilt.

3.3.2 NoSQL Datenbanken

„Not only SQL“-Datenbanken weisen im Gegensatz zu relationalen keine Tabellenstruktur zur Datenspeicherung auf. Stattdessen werden die Daten mithilfe von flexiblen Techniken wie Dokumenten, Graphen, Wertepaaren und Spalten abgebildet. Damit eignen sich NoSQL-Systeme optimal für Anwendungen, bei denen grosse Datenmengen verarbeitet werden müssen und die flexible Strukturen erfordern. Durch die Nutzung von Clustern und Cloud-Servern können NoSQL-Datenbanken auch bei grossen Datenmengen die Daten gleichmässig aufteilen und eine gute Performance erreichen. Bei NoSQL-Datenbanken kann man vier Hauptkategorien unterscheiden [6].

Dokumentenorientierte Datenbank

Eine dokumentenorientierte Datenbank ermöglicht es, Daten in Form von Dokumenten unterschiedlicher Länge abzulegen. Diese Dokumente werden in einer Baumstruktur dargestellt. Die Daten, die eingelesen werden, müssen nicht strukturiert sein. Als Datenformat wird heute vor allem JSON verwendet, das den schnellen Datenaustausch zwischen Anwendungen ermöglicht.

Vorteile

- Die Daten müssen **nicht strukturiert** sein.
- Es ist einfach, neue **Informationen hinzufügen**, da sie nicht in alle Dokumente eingebunden werden müssen.
- Eine **horizontale Skalierung** ist einfach möglich.

Nachteile

- Bei stark vernetzten Datenmengen nimmt die **Performance** ab.

Diese Informationen wurden teilweise vom Digitalen Guide von IONOS [7] übernommen.

Graphdatenbank

Graphdatenbanken bestehen aus Knoten und Kanten, um die Beziehung zwischen den Daten abzubilden. Die Knoten repräsentieren dabei Entitäten und die Kanten die Beziehungen. Knoten können auch noch dazugehörige Eigenschaften haben. Dies eignet sich gut für Daten die eine hohe Vernetzung aufweisen, da die graphische Verknüpfung eine gute Performance ermöglicht. Da der Graph persistente Beziehungen hat und diese nicht berechnet werden müssen, ermöglicht dies, den Graph effizient zu traversieren.

Vorteile

- Die **Performance** ist abhängig von der Anzahl konkreter Beziehungen und nicht der Datenmenge.
- **Flexible und agile Struktur**
- **Übersichtliche Veranschaulichung** von Beziehungen

Nachteile

- Die Auslegung auf eine Ein-Server-Architektur bewirkt eine **schlechte Skalierbarkeit**.
- Die **Abfragesyntax** ist nicht einheitlich.

Diese Informationen wurden vom Digitalen Guide von IONOS [8] übernommen.

Key-Value-Datenbank

Bei der dritten Kategorie, der Key-Value-Datenbank, werden Daten mittels Schlüssel-Werte-Paaren gespeichert. Einfache Formen einer solchen Datenbank sind in objektorientierten Programmiersprachen Arrays, Dictionaries oder Maps. Wie in diesen Datenstrukturen sind auch bei einer Key-Value-Datenbank die Keys immer eindeutig [6].

Vorteile

- **Hohe Performance**
- **Flexible Skalierbarkeit**
- Key-Value Datenbanken weisen eine **hohe Portabilität** auf, da sie auf ein anderes System transferiert werden können, ohne dass dabei der Code geändert werden muss.

Nachteile

- Das **Abbilden als Key-Value-Pair** ist nicht für alle Objekte einfach.
- Es können keine **Suchqueries** erstellt werden.

Diese Informationen wurden teilweise vom Digitalen Guide von IONOS [9] übernommen.

Spaltenorientierte Datenbank

In spaltenorientierten Datenbanken werden die Datensätze in Spalten gespeichert. Beim Datensatz in der Tabelle 3.1 würden die Speicherblöcke wie folgt aussehen: | 1, 2, 3 | Max, Paul, Lea | 50, 30, 20 |.

ID	Name	Alter
1	Max	50
2	Paul	30
3	Lea	20

Tabelle 3.1: Beispieldatensatz für eine spaltenorientierte Datenbank

Dies reduziert die Festplattenzugriffe gegenüber einer Speicherung in Zeilen. Bei Berechnungen von Werten ist dies ein wesentlicher Vorteil. Die Informationen liegen bestenfalls in einem Block oder zumindest in nahe zusammen liegenden Blöcken [10].

Vorteile

- Der Zugriff auf eine Spalte ist schnell, was eine **effiziente Auswertung von grossen Datenmengen** ermöglicht.
- Es besteht die Möglichkeit der **Datenkompression**, da die Daten einer Spalte vom gleichen Typ sind und Einträge des gleichen Typs nahe beieinander liegen.

Nachteile

- **Transaktionen** sind langsamer durch die Spaltenorientierung.
- **Updates** für einzelne Einträge sind schwierig.

Diese Informationen wurden teilweise vom Digitalen Guide von IONOS [11] übernommen.

3.3.3 Suchmaschinen

Obwohl eine Suchmaschine eigentlich keine Datenbank ist, wird deren Verwendung im Zusammenhang mit einer Datenbank und einer Suche immer populärer. Suchmaschinen bieten zusätzliche Funktionen an, wie die Volltextsuche, Vorschläge und komplexe Queries. Ergänzend zu einer Datenbank können diese also die Filtereffizienz für die Facetensuche steigern. Der Nachteil von Suchmaschinen liegt in der zusätzlichen Komplexität und Abhängigkeit, die eine Integration in ein System mit sich bringt. Ein Beispiel für eine Suchmaschine ist Elasticsearch.

Elasticsearch könnte man auch als NoSQL-Datenbank verwenden ohne zusätzliche Datenbank. Jedoch hat Elasticsearch gegenüber einer regulären Datenbank Limitationen. Das grösste Problem für dieses Projekt wäre der mögliche Datenverlust. Bei grossen Datenmengen kann mit Elasticsearch nicht garantiert werden, dass keine Daten verloren gehen. Die Entwickler sind aber daran, das System robuster zu gestalten [12]. Trotzdem wird Elasticsearch vielfach als Teil einer Pipeline mit einer zusätzlichen Datenbank verwendet.



4 | Client-Server

Eine Client-Server-Architektur ermöglicht es, Daten serverseitig beispielsweise mit einer Datenbank zu verwalten, um dynamischen Inhalt auf einer Website zu integrieren. Bei einer Webapplikation fungiert der Browser als Client und kommuniziert mit dem Webserver via HTTP. Der Browser kann einen HTTP-Request absetzen, um Daten auf dem Server abzufragen, zu ändern, hinzuzufügen oder zu löschen. Der Request besteht aus einer URL, die den Endpunkt definiert, der angesprochen werden soll und einem Verb, das die erwünschte Aktion beschreibt (GET, POST, PUT oder DELETE).

In Abbildung 4.1 ist eine simple Client-Webserver-Architektur schematisch dargestellt. Der Browser schickt eine Anfrage an den Server. Der Server gibt die angeforderten Daten an den Client zurück mit einer entsprechenden Statusmeldung.

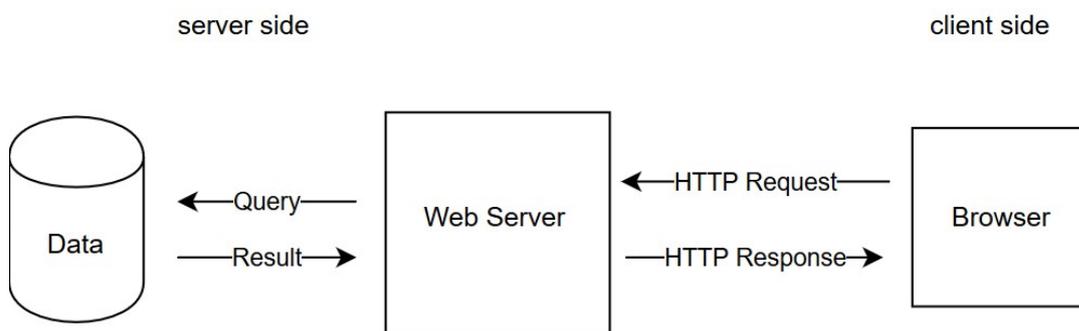


Abbildung 4.1: Beispiel Client-Webserver-Architektur

4.1 Optimierungen der Serverabfragen

Die zusätzlich benötigte Anfrage- und Antwortzeit, die eine Client-Server-Architektur mit sich bringt, kann durch einige Methoden reduziert werden. In diesem Unterkapitel werden für dieses Projekt relevante Methoden behandelt.

4.1.1 Pagination und Lazy Loading

Will man dem Benutzer grosse Datenmengen in einer visuellen Form präsentieren und die Daten serverseitig abfragen, gibt es folgende Probleme:

1. Die Ladezeit wäre viel zu lange, um alle Daten auf einmal zu laden.
2. Es können nicht beliebig grosse Datenmengen über einen Response Body versendet werden.

3. Der clientseitige Speicher ist möglicherweise zu klein.

Für solche Fälle eignen sich Pagination und das Lazy Loading von Daten. Beide Methoden haben die gleiche Grundidee. Es sollen immer nur so viele Daten geladen werden, wie zu einem bestimmten Zeitpunkt vom Benutzer benötigt.

Das Konzept der Pagination und des Lazy Loading besteht darin, die zu grosse Datenmenge in kleine Pakete zu unterteilen und somit die gesamte Datenmenge in mehreren Schritten zu laden. Das erste Datenpaket wird in den meisten Anwendungen automatisch geladen. Das Nachladen von weiteren Paketen wird vom Benutzer ausgelöst. Dies ermöglicht ein effizientes UI-Feedback für den Benutzer, der sogleich mit der Interaktion beginnen kann. Dieses Prinzip setzt eine Ordnung der Daten voraus.

Es gibt verschiedene Möglichkeiten wie man dieses Prinzip implementieren kann. Alle im Folgenden vorgestellten Möglichkeiten können mit Pagination oder Lazy Loading mittels Pagination umgesetzt werden. Pagination und Lazy Loading unterscheiden sich in diesem Konzept lediglich bezüglich des Triggers für das Laden neuer Daten. Bei der Pagination wird dieser durch einen Klick auf den entsprechenden Button ausgelöst. Beim Lazy Loading werden Daten nachgeladen, wenn durch Scrollen das Ende einer Liste erreicht wird. Die Optionen werden anhand von tabellarisch gespeicherten Daten erklärt. Für die folgenden Unterkapitel wurden Information von Moesif [13] übernommen.

Offset Pagination

Offset Pagination ist die einfachste Form einer Pagination. Mithilfe einer Offsetfunktion wird eine bestimmte Anzahl Zeilen in der Tabelle übersprungen und die nachfolgenden Einträge mit einer Funktion limitiert. Die resultierende Datenmenge ist das nächste darzustellende Datenpaket. Konkret könnte eine Anfrage an den Server wie folgt definiert werden:

```
GET /data?limit=20&offset=100
```

Diese Anfrage gibt ab der hundertsten Tabellenzeile die nächsten 20 Einträge zurück.

Vorteile

- Einfach integrierbar
- Wird von vielen Datenbanken unterstützt
- Funktioniert auch mit einer zusätzlichen Sortierung der Daten

Nachteile

- Nicht performant bei grossen Datenmengen mit grossen Offsets
- Inkonsistent, falls neue Daten in die Tabelle eingefügt werden

Diese Variante ist also insbesondere für kleinere Read-Only Datensätze geeignet.

Keyset Pagination

Muss die nächste Page geladen werden, wird bei dieser Anfrage der letzte Wert der aktuellen Page als Parameter mitgegeben. Es kann mit Hilfe von einer oder mehreren Spalten eine Referenz auf den letzten Wert der vorhergehenden Page gesetzt werden. Da die Werte in einer Spalte nicht eindeutig sein müssen, kann mit einem zusätzlichen Offset diese Referenz eindeutig zugeordnet werden. Konkret würde die erste Anfrage des Clients wie folgt aussehen:

```
GET /data?limit=20
```

Muss die nächste Page geladen werden, wird bei dieser Anfrage der letzte Wert des zu filternden Werts auf der aktuellen Page mitgegeben. Basierend auf den Daten in der Tabelle 4.1 würde die Abfrage für die Page 2 wie folgt aussehen:

```
GET /data?limit=20&date=2019-07-20.
```

Page	Id	Datum
1	001	2019-06-20
	002	2019-07-20
2	003	2019-08-20
	004	2019-08-20
3	005	2019-09-20
	006	2019-10-20

Tabelle 4.1: Beispieldatensatz für die Keyset Pagination

Die Anfrage für die Daten auf Page drei muss zum Limit und Datum noch eine dritte Information beinhalten, um sicherzustellen, dass die gleichen Daten nicht mehrfach auf verschiedenen Pages an den Client übermittelt werden. Wie erwähnt wäre ein zusätzlicher Offset Parameter eine Lösung:

```
GET /data?limit=20&date=2019-08-20&offset=1
```

Die Einbindung einer zusätzlichen Sortierung der Daten, wie zum Beispiel eine aufsteigende oder absteigende Reihenfolge der IDs in Tabelle 4.1, ist mit der Keyset Pagination ebenfalls möglich. Um die Konsistenz zu erhalten, müssen jedoch zusätzliche Mechanismen eingebaut werden. Eignen würde sich dazu, bei jedem Neusetzen der Sortierung die Daten der ersten Page zurückzugeben.

Vorteile

- Funktioniert auch mit existierenden Filtern
- Konsistente Ordnung, auch wenn neue Daten eingefügt werden
- Konsistente Performance auch bei grossen Offsets

Nachteile

- Kopplung von Paging mit Filter und Sortierung. Filter werden auch benötigt, wenn nicht gefiltert wird.

4.1. OPTIMIERUNGEN DER SERVERABFRAGEN

- Niedrige Performance für Spalten mit niedriger Kardinalität

Aufgrund der benötigten hohen Kardinalität eignet sich Keyset Pagination gut für Logdaten oder Daten mit Zeitstempel.

Seek Pagination

Die Seek Pagination ist eine Erweiterung der Keyset Pagination ohne hohe Kopplung. Um die Grenzen der Pages zu finden, wird mit IDs gearbeitet. Die Abfrage für die erste Page ist analog zu den bisherigen Varianten:

```
GET /data?limit=20
```

Für die zweite Seite muss folgende Abfrage abgesetzt werden:

```
GET /data?limit=20&afterId=20
```

Für jede weitere Seite kann bei diesem Beispiel die letzte `afterId` jeweils um 20 erhöht werden, um die Daten der nächsten Seite zu erhalten.

Vorteile

- Keine Kopplung zwischen Pagination- und Filterlogik
- Konsistente Ordnung, auch wenn neue Daten eingefügt werden
- Konsistente Performance auch bei grossen Offsets

Nachteile

- Erfordert eine vergleichsweise komplexere Implementation für die Möglichkeit, zusätzlich sortieren zu können und auf vorhergehende Pages zuzugreifen.
- Die `afterId` kann ungültig werden, wenn das Element mit dieser ID gelöscht wird.

Die Seek Pagination ermöglicht es, eine effiziente Pagination zu implementieren. Die Implementation von zusätzlicher Funktionalität wird aber deutlich komplexer. Sie ist daher für grosse Datensätze geeignet, wo sich der Aufwand für die Performance lohnt.

4.1.2 Caching

Caching beschreibt das Vorgehen, Informationen zwischenspeichern, die von einem Server abfragt werden. Will der Client eine Abfrage senden, die identisch mit einer Abfrage im Cache ist, kann auf diesen Speicher zugegriffen werden und es muss kein Request an den Server geschickt werden, was die Reaktionszeit der Applikation verbessert.

Eine einfache Variante, einen Cache zu implementieren, ist **manuell** mithilfe von Datenstrukturen der verwendeten Programmiersprache einen Speicher anzulegen. Beispielsweise könnte man in JavaScript eine Server-Response, die Objekte im JSON-Format enthält, in Variablen zwischenspeichern. Dadurch entstehen keine zusätzlichen Abhängigkeiten von browserspezifischen APIs. Man könnte für die Speicherung der Responses auch ein

Array oder Objekt verwenden und so eine Historie abbilden. Die Verwaltung ist jedoch komplex und die Daten können nicht im sekundären Speicher abgelegt werden.

Eine Alternative dazu bietet die **Cache Web API**. Diese ermöglicht es, Server-Responses zu speichern und zu verwalten. Mit `Cache.add(request)` kann ein Request direkt in den Cache gespeichert werden. Dies löst ein `fetch()` aus und speichert die zugehörige Response. `Cache.put(key, response)` bietet die Möglichkeit, die Response mit einem entsprechenden Key zu speichern. Der Key kann das Request-Objekt oder die URL beinhalten. Des Weiteren bietet die Cache Web API Methoden an, mit deren Hilfe man gespeicherte Requests anhand des Keys im Cache suchen kann: `Cache.match(request, options)`, `Cache.matchAll(request, options)`. Der Vorteil dieser Variante liegt also in den zur Verfügung gestellten Methoden. Der Nachteil ist die zusätzliche Komplexität und die Verantwortung den Cachespeicherplatz zu verwalten.

Herausforderungen

Möchte man mithilfe eines Cache die Filteranfragen verringern, besteht die Herausforderung im Erkennen von äquivalenten Filter-Requests. Zwei Filter können logisch gesehen identisch sein, jedoch eine andere Reihenfolge der gesetzten Filter aufweisen. Es gibt verschiedene Varianten, solche Anfragen auf Äquivalenz zu prüfen:

1. Mit Hilfe eines Algorithmus wird der aktuelle Request mit den bisherigen verglichen.
2. Die URL wird so aufgebaut, dass identische Filter, mit einem equals-Vergleich, erkannt werden.
3. Das Request-Objekt wird so aufgebaut, dass identische Filter mit einem equals-Vergleich erkannt werden.

4.2 Datenbankoptimierungen

4.2.1 Indexing

Bei Datenbanken dient Indexing dazu, eine effizientere Bearbeitung von Anfragen zu erreichen. Beim Erstellen eines Indexes fügt die Datenbank eine zusätzliche Datenstruktur ein, meist einen B-Tree. In SQL-Datenbanken werden in diese neue Struktur die indexierten Spaltenwerte mit einer Pointerreferenz hinzugefügt. In MongoDB werden die spezifischen Felder oder eine Sammlung von Feldern eingefügt. Durch die Verwendung dieser Struktur müssen nicht bei jeder Anfrage alle Datenbankeinträge durchsucht werden, was sub-lineare Zeitkomplexität ermöglicht. Eine lineare Suche wäre für grosse Datenbanken ineffizient.

Ein Problem des Indexing ist, dass zusätzliche Schreibzugriffe nötig sind und mehr Speicherplatz benötigt wird, um die Indexes aktuell zu halten. Da die Aktualisierung der Indexes nur nach einem Schreibzugriff vorgenommen werden muss, kann dieses Problem für Anwendungen, die nach einer Initialisierungsphase nur noch Lesezugriffe durchführen, vernachlässigt werden.

Nach Angabe der Entwickler von MongoDB liegt die häufigste Ursache für schlechte Performance einer Datenbank bei ungeeigneten oder fehlenden Indexes [14]. Die folgende Übersicht zeigt, wie man Indexing auf der MongoDB nach „best practice“ verwenden sollte.

4.2.2 Verwendung von Indexes

MongoDB bietet verschiedene Typen von Indexes an, die aber nicht alle gleich gut geeignet sind. Es wird empfohlen, vor allem Compound Indexes zu verwenden. **Compound Indexes** basieren auf mehreren Feldern. Anfragen auf mehrere Felder können damit effizienter gemacht werden als mit je einem Index pro Feld. Für Compound Indexes spielt die Reihenfolge der Felder eine Rolle. Die Entwickler der MongoDB empfehlen die folgende Ordnung (**ESR Regel**):

1. Felder, bei denen auf Gleichheit (equality) geprüft wird.
2. Felder, nach denen sortiert (sort) wird.
3. Felder, die auf Wertebereiche (range) geprüft werden.

Ebenfalls hat die Zusammensetzung der Queries einen Einfluss auf die Filtereffizienz. Wenn immer möglich sollten Covered Queries verwendet werden. Als **Covered Queries** werden Datenbankanfragen bezeichnet, für die alle Felder in einem Index vorhanden sind, wenn sie beim Filtern und Sortieren oder in der Antwort an den Client verwendet werden. Das führt dazu, dass kein direkter Zugriff auf die Quelldokumente nötig ist, was sich positiv auf die Performance auswirkt.

Indexes sollten nur für Felder mit einer hohen Kardinalität erstellt werden. Die **Kardinalität** eines Feldes ist die Anzahl der verschiedenen Werte, die auftreten können. Felder mit einer tiefen Kardinalität können jedoch in Compound Indexes integriert werden, wenn die resultierende Kombination eine hohe Kardinalität aufweist [14].

4.2.3 Herausforderungen

Für optimale Performance ist das Ziel, jede Anfrage als Covered Query stellen zu können. Das ist oftmals aus zwei Gründen nicht möglich. Erstens kann es sein, dass die Felder, für welche die Anfragen gestellt werden, nicht im Vorhinein bekannt sind, weil beispielsweise auf einem vom Nutzer eingelesenen Datensatz gearbeitet wird. Zweitens nimmt die Anzahl möglicher Kombinationen, für die ein Index erstellt werden muss, exponentiell mit der Anzahl zu betrachtender Felder zu, was zu einem grossen Speicherbedarf führen kann.



Teil II

Umsetzungsergebnis

5 | Requirements

Die Anforderungen an den Prototyp wurden an Sitzungen mit den Betreuern diskutiert und festgelegt. Die folgenden Requirements soll der Prototyp erfüllen.

5.1 Funktionale Anforderungen

Kernpunkt dieser Bachelorarbeit ist es, eine innere Datenstruktur zu erarbeiten, die es erleichtert, die facettrierte Suche für eine neue Domäne zu implementieren. Weil weniger in die Entwicklung des Grundsystems investiert werden muss, können Ressourcen freigesetzt werden, die für die Entwicklung von neuen Facetten eingesetzt werden können.

Die funktionalen Anforderungen an den Prototyp lauten wie folgt:

1. Es soll möglich sein, die Ergebnismenge mithilfe von Facetten zu filtern.
2. Der Prototyp soll eine Tabelle beinhalten, worin die Ergebnismenge dargestellt wird.
3. Ein Filter kann mithilfe einer Selektion von einem oder mehreren Facettenwerten gesetzt werden.
4. Die Filter sollen in einer Reihenfolge angeordnet werden können.
 - Filter, die nebeneinander liegen, sollen von links nach rechts mittels UND-Operation verknüpft werden. Die Selektion eines Facettenwertes wirkt für alle nachfolgenden Facetten und die Ergebnismenge als Filter.
 - Filter, die vertikal angeordnet sind, sollen mittels ODER-Operation verknüpft werden. Die Reihenfolge hat dabei keinen Einfluss auf die Ergebnismenge.
5. Jeder Facettenwert soll eine Ergebniszahl besitzen, die anzeigt, wie viele Daten in der aktuellen Ergebnismenge dieses Attribut besitzen.
6. Sind mehrere Facetten aktiv, soll, wenn ein Filter gesetzt wird, auf allen Facetten rechts vom Filter die Ergebniszahl direkt aktualisiert werden.
7. Wird ein Filter verändert, soll sogleich auch die Ergebnismenge angepasst werden.
8. Die Filter-Queries sollten unabhängig davon formuliert werden, welches Datenformat vorhanden ist (CSV, SQL-Datenbank, ...). Es soll gegen das interne Datenformat programmiert werden. Die Datenschnittstelle muss nicht optimiert sein, sollte aber problemlos angepasst oder ausgetauscht werden können.
9. Die Daten sollen möglichst feingranular untersucht werden können.

5.2 Nicht funktionale Anforderungen

5.2.1 Usability

Die Wartezeit bei einer Suche sollte 1 Sekunde nicht überschreiten. Laut der Nielsen Norman Group liegt das zeitliche Limit, bei dem der Benutzer das Gefühl hat, die Applikation reagiere sofort, bei 0.1 Sekunden. Bei einer Antwortzeit zwischen 0.1 und 1 Sekunde verliert der Benutzer das Gefühl, ein sofortiges Feedback zu erhalten und nimmt eine Verzögerung wahr [15].

Das UI wird von Expertenbenutzern angewendet. Man kann also davon ausgehen, dass eine Einführung für die Applikation durchgeführt wird.

5.2.2 Supportability

Browser mit ECMAScript 6 sollen unterstützt werden. Dies wären nach w3schools [16]:

- Chrome
- Edge
- Firefox
- Safari
- Opera

Für die Entwicklung des User Interface wird Chrome verwendet. Die visuelle Konsistenz muss nicht für alle Browser gewährleistet sein. Responsiveness wird nicht vorausgesetzt.

5.2.3 Gegebenheiten

Es sollen nicht veränderliche Daten untersucht werden. D. h. es kann zu Beginn auch eine „Initialisierung“ durchgeführt werden, um Daten vorzubereiten. Frameworks oder andere Abhängigkeiten sollten nicht verwendet werden.

5.2.4 Endresultat

Als Resultat des Projekts sollte eines der folgenden Produkte entstehen:

- Framework: Ein System, dessen Source-Code nicht geändert wird.
- Library: Eine Sammlung an Funktionen, die von Entwicklern bei der Implementierung verwendet werden.
- Pattern: Eine Definition, wie eine Implementierung gemacht werden kann. Erfahrungswerte werden strukturiert festgehalten, insbesondere was die Gemeinsamkeiten von konkreten Ansätzen sind. Zudem muss ein Beispiel enthalten sein.
- Template: Ein Beispiel als Ausgangslage für andere Entwickler, die dann den Quellcode ändern und an die eigenen Bedürfnisse anpassen können.

5.3. *OUT OF SCOPE*

Das Einlesen von Datensätzen kann auf unterschiedliche Arten umgesetzt werden. Bei einem Framework wird das Format vorgegeben. Bei einer Library werden Funktionen angeboten, die unterschiedliche Datensätze einlesen können. Beim Design Pattern würde das benötigte Interface beschrieben.

5.3 Out of Scope

Es wird nicht ein fertiges, vollständig ausgearbeitetes Produkt erwartet. Spezialitäten der facettierten Suche sind nicht Kernthema dieser Arbeit, jedoch soll dokumentiert werden, was für die Weiterentwicklung beachtet werden muss und wie man Erweiterungen integrieren kann.

6 | Designentscheide

Dieses Kapitel beschreibt grundsätzliche Entscheidungen, die in der Entwicklung des Prototypen getroffen wurden. Einige der beschriebenen Aspekte haben sich im Verlauf des Projekts verändert. In diesen Fällen wird auch erläutert, aus welchen Gründen die Änderungen vorgenommen wurden.

6.1 Innere Datenstruktur

Die Organisation der Informationen, die für die facetiierte Suche mit dem Prototyp benötigt werden, wird als innere Datenstruktur bezeichnet. Sie umfasst die Beschreibung der Facetten, die Abbildung der getätigten Selektionen und die Speicherung der Facettenwerte mit zugehöriger Trefferanzahl.

6.1.1 Facettenbeschreibung

Die Facettenbeschreibung ist die Spezifikation der Facetten, die dem Nutzer zur Verfügung gestellt werden. In einer frühen Version lag der Fokus darauf, die einzelnen Hierarchiestufen in den Facetten zu definieren. Die Beschreibung war insbesondere darauf ausgelegt, Facetten zu beschreiben, die auf mehreren Dimensionen im Datensatz basieren. Die folgende Beschreibung definiert eine solche Facette mit zwei Hierarchiestufen. Die obere Hierarchiestufe basiert auf der Spalte `Court` und die untere auf der Spalte `Surface`.

```
1 const sampleFacet1 = {
2   id: 'f1',
3   name: 'Setting',
4   hlevels: [{
5     col: 'Court',
6     separator: '|',
7     position: 0,
8     groupedby: [],
9   }, {
10    col: 'Surface',
11    separator: '|',
12    position: 0,
13    groupedby: [],
14  }],
15 };
```

Die Erkenntnis, dass eine Facette dieser Art auch durch die Verknüpfung von zwei flachen Facetten ausgedrückt werden kann, hat dazu geführt, die Beschreibung zu vereinfachen.

Die überarbeitete Beschreibung enthält eine ID und einen Namen für die Darstellung im User Interface. Dazu kommt die Angabe, auf welcher Dimension der Datenquelle die Facette basiert und ob sie flach oder hierarchisch ist. Für Facetten, die eine Gruppierung aufweisen, muss noch die Art und die konkrete Zusammensetzung der Gruppen angegeben werden. Dafür wird jeder Gruppe ein Facettenwert (name) als Gruppenname und eine Liste von Facettenwerten (foci) als Elemente der Gruppe zugewiesen.

Beschreibung definieren

Die Entwicklung eines eigenen User Interface, um Facetten zu definieren, ist nicht Teil des Projektumfangs. Die Beschreibung wird in einer Konfigurationsdatei fest codiert. Dadurch gibt es zur Laufzeit keine Möglichkeit, die Facetten zu verändern, wenn beispielsweise eine Gruppierung geändert werden muss.

Beispiele

Die nachfolgenden Beispiele beschreiben unterschiedliche Ausprägungen einer Facette, die auf einer einzelnen Spalte im Datensatz basiert. Das erste Objekt führt zu einer Facette ohne Hierarchie.

```
1 const sampleFacet2 = {
2   id: 'f2',
3   name: 'Day of week (Flat)',
4   sourceColumn: 'DAY_OF_WEEK',
5   type: 'flat',
6   groupType: 'none',
7   grouping: [],
8 };
```

Die zweite Beschreibung erzeugt eine Facette, bei der die Zahlenwerte gruppiert werden. Die einzelnen Werte bleiben in der Facette nicht enthalten.

```
1 const sampleFacet3 = {
2   id: 'f3',
3   name: 'Day of week (Range)',
4   sourceColumn: 'DAY_OF_WEEK',
5   type: 'flat',
6   groupType: 'range',
7   grouping: [{
8     name: 'unassigned',
9     foci: [],
10  }, {
11    name: 'Weekday',
12    foci: [1, 6],
13  }, {
14    name: 'Weekend',
15    foci: [6, 8],
16  }],
17 };
```

Mit der dritten Form werden die Einträge nach individuellen Attributen gruppiert. Daraus entsteht eine Hierarchie. Die Gruppennamen sind die Oberkategorien und die Werte werden als Unterkategorien eingeordnet.

```
1 const sampleFacet4 = {
2   id: 'f4',
3   name: 'Day of week (Group)',
4   sourceColumn: 'DAY_OF_WEEK',
5   type: 'hierarchical',
6   groupType: 'value',
7   grouping: [[
8     {
9       name: 'Weekday',
10      foci: [1, 2, 3, 4, 5],
11     },
12    {
13      name: 'Weekend',
14      foci: [6, 7],
15     },
16   ]],
17 };
```

6.1.2 Filterobjekt

Die Auswahl auf den Facetten muss in einer Form vorliegen, die für Serverabfragen verwendet werden kann. In dieser Arbeit wird diese Beschreibung als Filterobjekt bezeichnet. Man kann das Filterobjekt als den Teil einer Query verstehen, der beinhaltet, welche Bedingungen an die Resultate erfüllt sein müssen. Im Verlauf der Arbeit wurden drei Varianten erarbeitet. Im Folgenden wird die Entwicklung dieser Filterobjekte erörtert.

Objekt

Die erste Variante des Filterobjekts wird mittels JavaScript-Objekt umgesetzt. Der Schlüssel entspricht einem Spaltennamen aus dem Datensatz. Die zugehörigen Werte sind Arrays, die die ausgewählten Attribute enthalten.

```
1 const filterObj = {
2   Surface: ['Grass', 'Clay'],
3   Round: ['1st Round']
4 };
```

Diese Variante funktioniert, wenn die Facetten unabhängig voneinander sind und nicht in einer Reihenfolge angeordnet werden. Dann können die Selektionen der Facetten in einem einzelnen Objekt gesammelt und auf die Ergebnismenge angewendet werden. Das Objekt ist ungeeignet, wenn nur Teile des Filters angewendet werden sollen. Die Ordnung von Properties in JavaScript-Objekten war bis ES2015 nicht sichergestellt, weil Objekte als „unordered collection of properties“ definiert wurden [17]. Die Verwendung der Properties wird zugunsten der Lesbarkeit in diesem Fall vermieden.

Objekt und Array

Die Reihenfolge wird dadurch beschrieben, dass zusätzlich zum bestehenden Filterobjekt ein Array eingeführt wird, das die Reihenfolge der Facetten abbilden kann. Es kann anhand der Filterreihenfolge bestimmt werden, welche Teile des Filterobjekts angewendet werden müssen. Wenn im nachfolgenden Beispiel die Werte für die Spalte Round angefragt werden, wird nur nach Grass und Clay von Surface gefiltert. Sollen die aktuellen Werte für Surface bestimmt werden, wird gar kein Filter gesetzt, denn im Array filterOrder steht Surface an erster Stelle.

```
1 const filterObj = {
2   Surface: ['Grass', 'Clay'],
3   Round: ['1st Round']
4 };
5 const filterOrder = ['Surface', 'Round'];
```

Die Nachteile dieser Struktur sind, dass zwei Elemente aktuell und konsistent gehalten werden müssen und dass keine zusätzlichen Informationen zu den Facetten übergeben werden können. Dieser Aufbau verunmöglicht, eine Spalte aus dem Datensatz in mehreren aktiven Facetten zu haben, weil im Filterobjekt nicht unterschieden werden kann, in welcher Facette die Auswahl für diese Spalte getätigt wurde. Ausserdem gibt es keine Möglichkeit zu definieren, wie die Facetten kombiniert werden sollen.

Array von Objekten

Die dritte Variante speichert die Informationen in einem Array. Jeder Eintrag in diesem Array repräsentiert die Selektion auf einer Facette. Die Selektion kann sich dabei auf eine oder mehrere Spalten im Datensatz beziehen und es wird mit or und and definiert, wie die Facetten kombiniert sind, wenn mehrere Spalten für eine einzelne Facette involviert sind. Zusätzlich zu den selektierten Werten werden ausserdem die Informationen zur Gruppierung eingebunden. Die Gruppierung für Zahlenbereiche kann nicht clientseitig vorgenommen werden und muss auf den Server ausgelagert werden. Das nachfolgende Beispiel zeigt ein solches Filterobjekt, wie es auch im Prototyp verwendet wird.

```
1 const filter = [{
2   or: {
3     Surface: {
4       foci: ['Grass', 'Clay'],
5       groupType: 'none',
6       grouping: []
7     },
8     Round: {
9       foci: ['1st Round'],
10      groupType: 'none',
11      grouping: []
12    }
13  }
14 }, {
15   and: {
16     WRank: {
17       foci: ['1-10'],
18       groupType: 'range',
19       grouping: [{ name: '1-10', foci: [1, 11]},...]
20     },
21     Location: {
22       foci: ['Basel'],
23       groupType: 'value',
24       grouping: [[{ name: 'Switzerland', foci: ['Basel']},...
25                 ]]
26     }
27   }
28 }];
```

6.1.3 Verwaltung des Filterobjekts

Initial wurde das Filterobjekt zentral verwaltet. Dies hat zu Konsistenzproblemen geführt, weil die Selektion, die auf den Facetten gesetzt ist, und das Filterobjekt synchron gehalten werden mussten. Deshalb wird das Filterobjekt nach Bedarf neu zusammengesetzt, bevor die Serveranfrage abgesetzt wird.

6.1.4 Zustandsverwaltung

In den Facetten können die Daten je nach Typ in unterschiedlichen Datenstrukturen gespeichert werden. Für klassische Facetten kommt eine Baumstruktur zum Einsatz, da eine Hierarchie abgebildet werden muss. Der Baum wird bei der Initialisierung der Anwendung aufgebaut und ändert die Zusammensetzung nicht. Bei einer Aktualisierung werden lediglich die Zahlenwerte, die in den Knoten des Baums abgelegt sind, angepasst. Die Tabelle hingegen verwendet eine Liste. Beim Neuladen von Daten wird die Liste geleert und dann mit dem neuen Inhalt befüllt.

6.2 Datenspeicher

Für die Auswahl des Speicherobjekts muss beachtet werden, dass die Kapazität gross genug ist für Datensätze mit mehreren Millionen Einträgen. Das Filtern von 0.5 bis 1.5 Millionen Einträgen sollte nicht länger als eine Sekunde dauern.

6.2.1 Erste Version

In der ersten Implementationsphase wurde der Datenspeicher als JavaScript Datenstruktur implementiert. Dies erbrachte für die Filterfunktion eine hohe Performance, wie im Abschnitt 3.1 beschrieben. Der Nachteil dieser Speicherstruktur liegt darin, dass die Daten bei einem Reload nicht mehr vorhanden sind. Zudem war es bei Versuchen nicht möglich, Datensätze mit mehr als vier Millionen Einträge in die Datenstruktur zu laden. Auf Grund dessen und den im Abschnitt 3.1 aufgelisteten Nachteilen wurde entschieden, dass die Kriterien für das Speichern der Daten mittels einer JavaScript Datenstruktur nicht erfüllt werden können. Der Datenspeicher muss mithilfe einer anderen Struktur umgesetzt werden.

6.2.2 Endversion



In einem weiteren Schritt wurde mit dem Ziel, eine optimale Alternative für die Datenspeicherstruktur zu finden, der Fokus auf die Anbindung einer Datenbank gelegt. Die clientseitigen Speichermöglichkeiten Web Storage API und die Cache API sind, wie im Abschnitt 3.1 beschrieben, aufgrund des Anwendungsfalls und der Datenmenge nicht geeignet. Die IndexedDB API wäre eine interessante Option. Da aber der Entwickler nicht sicherstellen kann, dass die eingelesenen Daten über die ganze Sitzungsdauer des Benutzers vorhanden sind, ist dies im Rahmen dieser Arbeit ebenfalls keine zufriedenstellende Lösung.

Aus diesen Gründen wurde eine Datenbank als Datenspeicher gewählt. Da es viele verschiedene Datenbanken gibt (siehe Tabelle 6.1), die unterschiedliche Vor- und Nachteile aufweisen, müsste man für die optimale Wahl der Datenbank verschiedene Integrationen testen und vergleichen. Mit einer Suchmaschine als zusätzlichem Layer könnte möglicherweise die Filtereffizienz zusätzlich gesteigert werden.

Datenbankaufbau		Datenbanken
	SQL	MySQL, PostgreSQL, MariaDB, Oracle
NoSQL	Dokumente	BaseX, CouchDB, SimpleDB, MongoDB
	Graph	Neo4j, Amazon Neptun, SAP Hana Graph, OrientDB
	Key-Value	Amazon DynamoDB, Berkeley DB, Redis, Riak, Voldemort
	Spalten	Amazon Redshift, MariaDB-ColumnStore, Apache Cassandra

Tabelle 6.1: Übersicht einiger der bekanntesten Datenbanken

Der Fokus dieser Bachelorarbeit liegt jedoch auf der inneren Datenstruktur. Daher ist es für dieses Projekt weniger wichtig, die für die Facettensuche optimale Datenbank mit einer Suchmaschine zu implementieren, als die Möglichkeit zu bieten, andere Datenbanken an den Prototyp anbinden zu können.

Für die konkrete Wahl der Datenbank sind folgende Punkte zu beachten:

- Einlesen eines tabellarischen Datensatzes soll möglich sein.
- Es muss anhand von Queries gefiltert werden können.
- Mit JSON als Datenformat kann der Umwandlungsaufwand von der Datenbankausgabe zur Speicher-Datenstruktur auf dem Client klein gehalten werden.
- Abfragen sollen bei Datenmengen im Millionenbereich effizient bleiben.

Aus diesen Punkten lässt sich ableiten, dass relationale und dokumentenorientierte Datenbanken am geeignetsten wären. Der Entscheid ist auf Grund des JSON-Datenformats und der grossen Community auf die dokumentenorientierte Datenbank MongoDB gefallen.

Auswirkungen der Datenbankwahl

Die MongoDB speichert die Daten permanent. Die Daten sind auch nach einem Refresh verfügbar. Datenmengen im Millionenbereich können gespeichert und effizient gefiltert werden. Die Filteroperation inklusive HTTP Request und Response mit den benötigten Umwandlungsfunktionen vom Client-Filterformat in eine Datenbankanfrage ist jedoch langsamer als eine Filteroperation mittels JavaScript Arrays.

Der Entscheid für eine Datenbank als Speicherobjekt und die Bedingung, dass man auch andere Datenbanken als die MongoDB verwenden kann, erfordert eine API Schnittstelle zwischen Client und Datenbank.

6.3 Modulsystem

Die Applikation ist modular aufgebaut. Jedes Modul ist eine eigenständige Repräsentation einer Datenmenge in Form von einer oder mehreren Facetten. Pro Modul können mehrere Facetten definiert werden. Sind diese aktiv, kann auf einer oder mehreren Facetten eine Auswahl getroffen und dadurch die Ergebnismenge eingegrenzt werden. Bei einem Modul wird nicht unterschieden, ob Filter gesetzt werden können oder ob es ausschliesslich als Visualisierung der Daten verwendet wird. Mehrere Module können mit einer AND-Operation kombiniert werden. Im ersten Modul wird immer der vollständige Datensatz dargestellt. In allen nachfolgenden Modulen werden jeweils nur die Ergebnisse angezeigt, die in allen vorherigen Modulen selektiert worden sind. Ein Modul kann im Aufbau einer Suche mehrfach verwendet werden. Diese Struktur wirkt sich sowohl auf die Erweiterbarkeit aus Entwicklersicht wie auch auf die Flexibilität für Benutzer positiv aus.

Die Abbildung 6.1 zeigt den Aufbau der Anwendung ohne Module (links) im Vergleich zum finalen Prototyp mit Modulen (rechts). Ohne Module gibt es einen Auswahlbereich, der die Facetten beinhaltet, und einen Anzeigebereich, der die Ergebnismenge in tabellarischer Form anzeigt. Der **Nachteil eines nicht modularen Systems** ist die **fehlende Möglichkeit** für alternative Visualisierungen. Mithilfe der Module können z. B. kartographische Facetten hinzugefügt werden. Ein weiterer Vorteil liegt darin, dass die Tabelle, die ohne Module lediglich als Anzeigebereich diente, mit einem modularen System eine duale Funktion aufweist. Sie stellt die Ergebnismenge dar und agiert auch als Facette, auf der einzelne Einträge ausgewählt werden können.

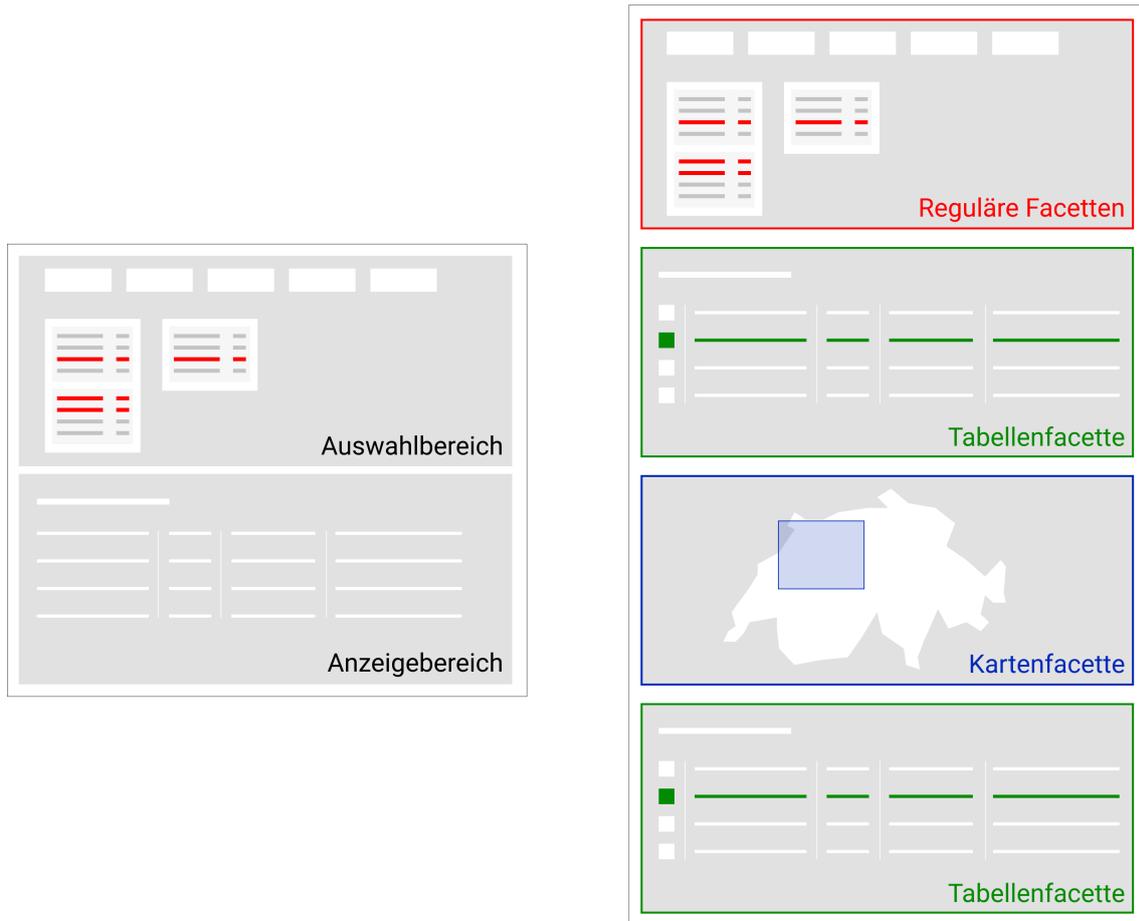


Abbildung 6.1: Statische Variante mit Auswahl- und Anzeigebereich (links) und Modulsystem (rechts)

6.4 Verhalten bei Selektion



Die wichtigste Interaktion eines Nutzers mit der Applikation ist das Treffen einer Auswahl auf einer Facette. In den nachfolgenden Abschnitten werden zwei Varianten erläutert, die als Ansätze infrage kommen.

Die Abbildung 6.2 beschreibt den Ablauf einer Selektion. Dem MainController wird im Code mitgeteilt, auf welchem Modul, in welcher Facette, welcher Wert ausgewählt wurde. Der MainController gibt die Nachricht, dass eine Selektion getätigt wurde, an das gegebene Modul weiter und löst anschliessend auf allen nachfolgenden Modulen eine Aktualisierung aus. Das Modul, auf dem die Auswahl getroffen wurde, gibt die Auswahl an die entsprechende Facette weiter und löst auf allen nachfolgenden Facetten innerhalb des Moduls eine Aktualisierung aus. Die Zielfacette fügt den neuen Wert zur Selektion hinzu.

6.4.1 Selektion erhalten

Wird auf einer Facette eine Auswahl getätigt, ändert sich nur der Selektionszustand des ausgewählten Facettenwertes. Die Selektionszustände auf allen anderen Facettenwerten bleiben unverändert. Aus UI-Sicht gibt es dabei für klassische Facetten keine Probleme,

6.4. VERHALTEN BEI SELEKTION

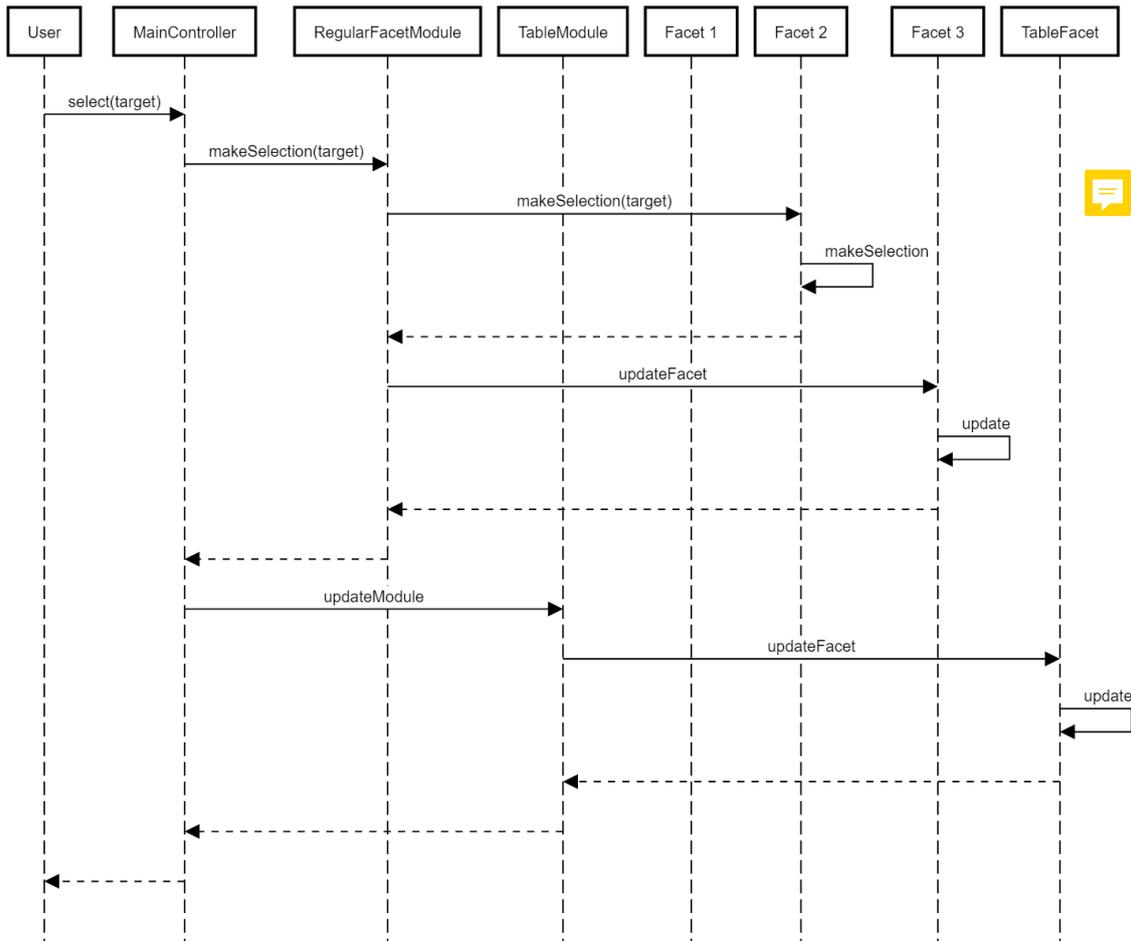


Abbildung 6.2: Interner Ablauf bei Auswahl im User Interface

weil immer alle Selektionszustände sichtbar sind und nur die Anzahl Resultate angepasst wird. Die Situation für die Tabellenfacette ist weniger offensichtlich. Dort gibt es drei denkbare Verhalten. Es gibt die Möglichkeiten, die Facette zurückzusetzen oder die Auswahl nur zu entfernen, wenn die Auswahlmöglichkeit nicht mehr sichtbar ist. Im Prototyp kommt aber die Variante zum Einsatz, bei der die Auswahl erhalten bleibt, solange sich die ausgewählten Einträge in der Ergebnismenge befinden. Zur Orientierung wird die Anzahl ausgewählter Einträge angezeigt. Das unterschiedliche Verhalten von Modulen führt möglicherweise dazu, dass für Nutzer eine umfangreichere Einführung nötig ist.

6.4.2 Selektion zurücksetzen

Alternativ können bei jeder getroffenen Auswahl alle Selektionszustände auf den nachfolgenden Facetten zurückgesetzt werden. So kann ein konsistentes Verhalten erreicht werden und es muss nicht zwischen verschiedenen Facetten- und Modultypen unterschieden werden. Die Idee der schrittweisen Einschränkung in Leserichtung wird strikt umgesetzt.

Diese Strategie hat den Nachteil, dass sie weniger fehlertolerant ist. Hat ein Nutzer auf mehreren Facetten eine Auswahl getroffen und stellt fest, dass ein Facettenwert zu viel oder zu wenig selektiert wurde, ist ein Zusatzschritt nötig, um die Ergänzung vorzunehmen ohne die Selektionszustände aller nachfolgenden Facetten zu ändern. Die Facette muss erst

6.4. VERHALTEN BEI SELEKTION

an die letzte Filterposition verschoben werden, bevor die Änderung vorgenommen werden kann. Möglicherweise muss dem Nutzer die Möglichkeit geboten werden, eine Aktion rückgängig machen zu können, um die Auswirkung von ungewollten Klicks zu reduzieren.

Im Prototyp wird der erste Ansatz angewendet. Dieser ist flexibler und gibt Nutzern mehr Kontrolle, da die Applikation keine eigenständige Änderungen an der Selektion vornimmt. Aus Sicht der Implementierung ist eine Umstellung auf die zweite Variante problemlos möglich. Wie in Abbildung 6.3 beschrieben, kann bei einer Auswahl auf allen nachfolgenden Facetten die Selektion aufgehoben werden, wenn vor der Funktion `update` erst `reset` aufgerufen wird.

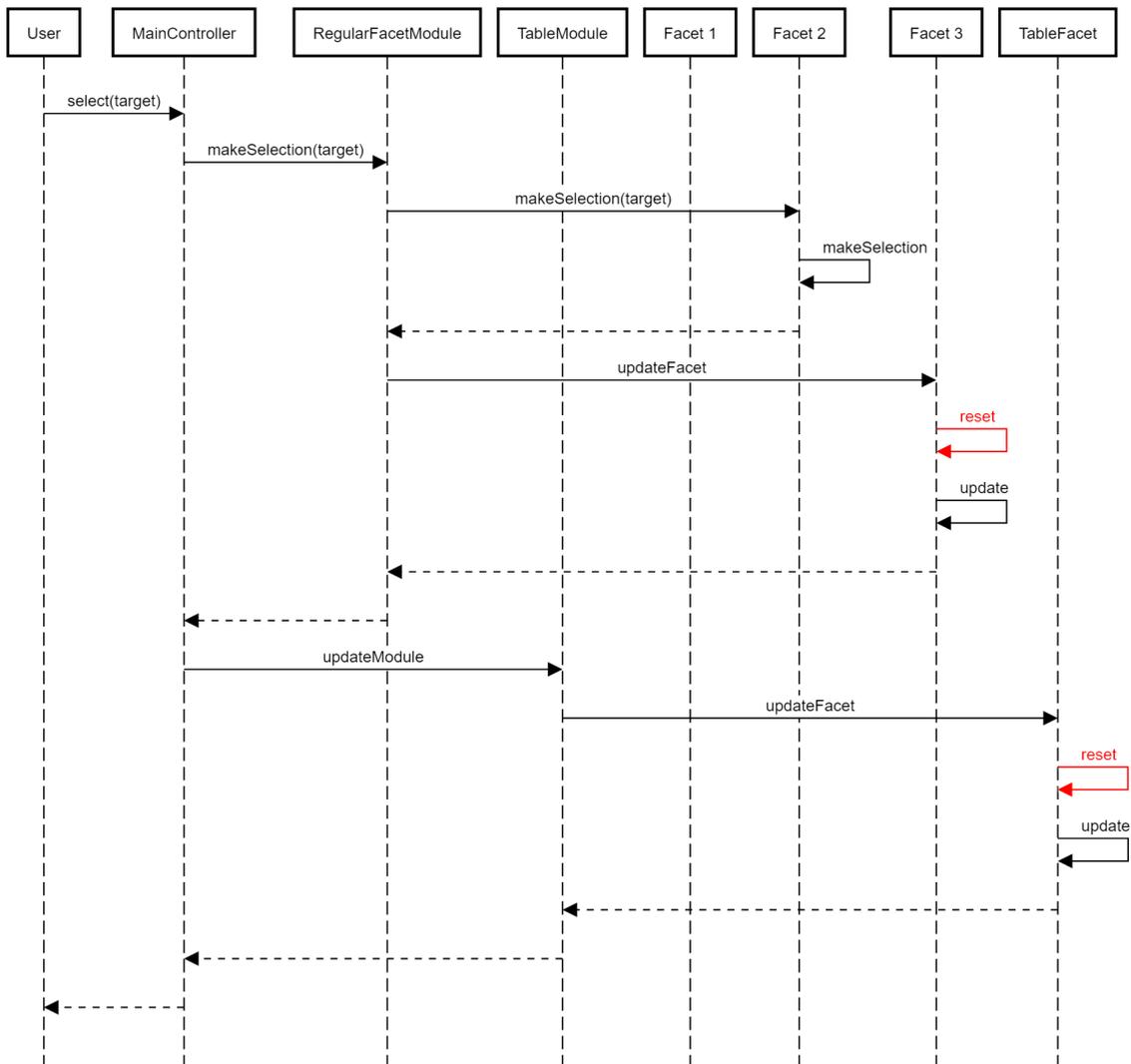


Abbildung 6.3: Vor jeder Aktualisierung wird die Auswahl zurückgesetzt

6.4.3 Weitere Beobachtungen

Das Hinzufügen einer Facette wird nicht als Interaktion verstanden, die eine Auswirkung auf die angezeigten Daten hat. Ein Zustand, der potenziell als inkonsistent interpretiert werden kann, ist die Selektion auf einer neu hinzugefügten Facette. Es sind keine Facet-

6.4. VERHALTEN BEI SELEKTION

tenwerte als selektiert erkennbar. Da jedoch eine Facette, auf der keine Selektion gesetzt ist, beim Filtern ignoriert wird, findet keine Einschränkung der Ergebnismenge statt. Die beiden Zustände, wenn alles selektiert oder deselektiert ist, sind also gleichwertig. Der Unterschied besteht darin, welche Bedeutung es für die nächste Interaktion hat. Wird auf einer vollständig selektierten Facette ein Wert ausgewählt, entfernt man alle Einträge, die diesen Wert enthalten. Ist noch kein Wert selektiert, führt die Auswahl dazu, dass ausschliesslich die Einträge angezeigt werden, die diesen Wert enthalten.

In weiterführenden Usability-Tests muss untersucht werden, welches der zwei beschriebenen Verhalten von Benutzern bevorzugt wird und welche zusätzlichen Werkzeuge die Auswahl erleichtern können. Die Funktionalität, alle selektierten Facettenwerte zu deselektieren, ist bereits implementiert. Die Selektion aller aktiven Facettenwerte wird nicht angeboten.

7 | Architektur

In diesem Kapitel wird erklärt, wie die Applikation aufgebaut ist und welche Architekturmuster verwendet werden.

7.1 Client-Server

Die Verwendung einer Client-Server-Architektur ist geeignet, um eine Datenbank als Speicher zu integrieren. Die Umsetzung für diese Arbeit ist in Abbildung 7.1 aufgezeigt.

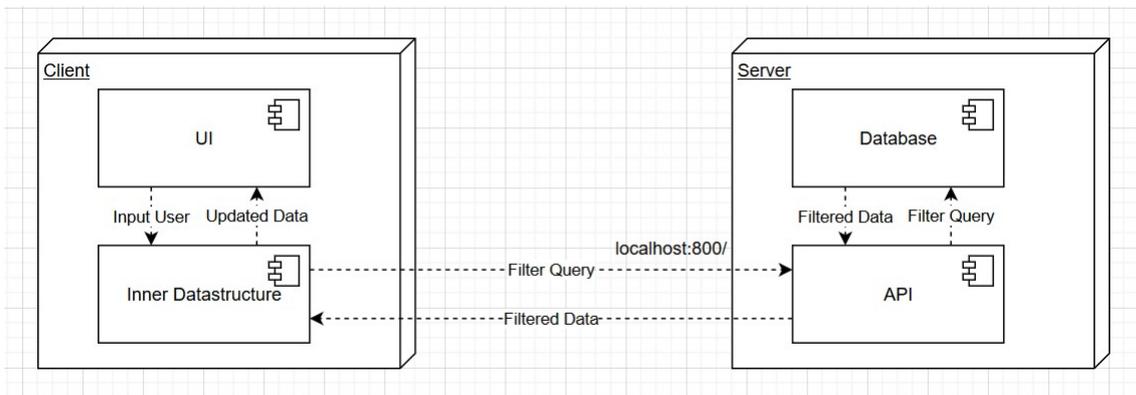


Abbildung 7.1: Aufbau der Client-Server-Kommunikation

Der Client verwaltet die innere Datenstruktur und versendet Filteranfragen an den Server. Der Filter wird vom Benutzer gesetzt und in der inneren Datenstruktur gespeichert. Der Server wandelt die Anfrage des Clients in eine Datenbankabfrage um. Diese wird auf der integrierten Datenbank verarbeitet und an den Client zurückgesendet. Die erhaltenen Daten werden auf dem Client gespeichert und im User Interface aktualisiert. Die API stellt zwei verschiedene URL-Bereiche zur Verfügung. Unter `/filter` sind alle Endpunkte definiert, die gefilterte Daten zurückgeben und unter `/file-reader` sind alle Endpunkte definiert, die das Einlesen des Datensatzes in die Datenbank betreffen.

7.2 Client

Die modulare Beschaffenheit der Applikation und die geforderte Erweiterbarkeit setzt voraus, dass eine klare „Separation of Concerns“ (Trennung der Zuständigkeiten) sichergestellt ist. Die folgenden Abschnitte beschreiben zwei Aspekte, die für die Umsetzung dieser Anforderung zentral sind.

7.2.1 MVC

Die Applikation verwendet den Model-View-Controller als Architekturmuster. Die Abbildung 7.2 illustriert die grundsätzliche Struktur. Der Nutzer interagiert mit der View. Er kann entweder passiv die Applikation betrachten oder eine Aktion auslösen. Eine ausgelöste Aktion wird an den Controller weitergeleitet, der Änderungen am Model vornimmt. Das Model informiert dann alle Views darüber, dass eine Änderung stattgefunden hat und übergibt die neuen Daten. Für die Datenabfrage wurde das Muster noch um eine Serviceebene ergänzt, die vom Controller aufgerufen wird, wenn neue Anfragen an die Datenquelle gemacht werden müssen.

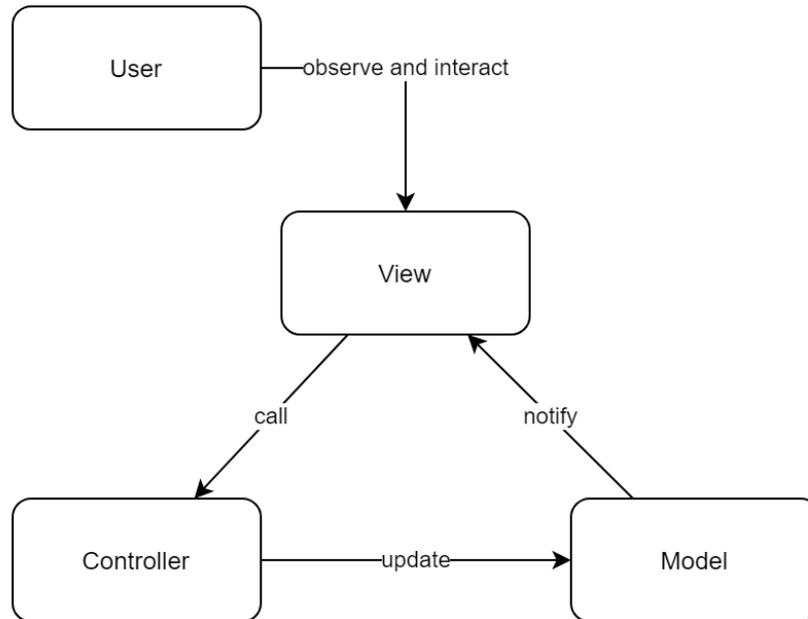


Abbildung 7.2: Struktur von MVC inklusive Benutzer

Model

Im Model werden die Daten gespeichert, die den Zustand der Anwendung beschreiben.

View

Die View ist für die Darstellung des User Interfaces zuständig. Dazu gehören die Anzeige der Informationen und das Anbieten von Interaktionsmöglichkeiten. Für die Anzeige der Informationen aus dem Model wird das Observer Pattern angewendet. Die View wird informiert, wenn sich im Model etwas ändert und kann darauf reagieren. Die Interaktionsmöglichkeiten sind Elemente, die Funktionen auf dem Controller aufrufen. Views können hinzugefügt, entfernt oder ausgetauscht werden, ohne die restliche Applikation zu ändern.

Controller

Controller enthalten die Applikationslogik. Sie stellen den Views eine Schnittstelle zur Verfügung, über die Aktionen, ausgelöst vom Benutzer, an das Model weitergeleitet werden können.

Service

Die Serviceabstraktion definiert, woher die Daten, die in der Anwendung angezeigt werden, bezogen werden. Services sind unabhängig vom Rest der Anwendung und werden bei der Initialisierung an die Controller übergeben. Die Datenquelle kann somit ausgetauscht werden, ohne Änderungen am Controller vornehmen zu müssen.

Für das End-to-End-Testing wird eine alternative Implementierung eingesetzt, die Daten aus einem lokal fest codierten Datensatz bezieht und damit ermöglicht, den Client unabhängig von der Serverimplementierung zu testen. Eine Serviceimplementierung muss das nachfolgende Interface erfüllen.

```

1 /**
2 * Interface for service.
3 *
4 * @typedef {Object} FacetService
5 * @property {function(Filter[], string): FilteredDataResponse}
   getFilteredData
6 * @property {function(FacetDescription, Filter[]): FocusCount
   []} getFilteredFoci
7 */

```

Die Funktion `getFilteredData` gibt eine Auswahl von Einträgen aus der Datenquelle zurück. Darin enthalten sind alle Dimensionen dieser Einträge. Die Funktion `getFilteredFoci` gibt für eine Dimension alle Attribute und wie oft sie vorkommen zurück.

7.2.2 Lokale Models

Der Zustand der Applikation wird nicht zentral verwaltet, sondern aufgeteilt und direkt vom Controller, der ihn nutzt, manipuliert. Auch wenn in vielen Webanwendungen aktuell mit React und Redux eine global Zustandsverwaltung eingesetzt wird, ist es für dieses Projekt weniger geeignet.

Vorteile

Einzelne Controller haben nur auf Models Zugriff, die für sie relevant sind und können so nicht unerwünscht den Zustand in andern Teilen der Applikation ändern. Die Models sind meist nur für eine einzelne Komponente relevant. Das `RegularFacetModule` muss nicht wissen, welche Daten in einer `RegularFacet` dargestellt werden. Der Code ist damit auch in der Wartbarkeit und für die Fehlersuche geeigneter, da der Ursprung der Zustandsänderung, effizient eingegrenzt werden kann. Ausserdem wirkt sich die Verwendung von lokalen Zuständen auch positiv auf die Testbarkeit aus, weil Applikationsteile getestet werden können, ohne von andern abhängig zu sein.

Nachteile

Die Struktur der Implementierung muss potenziell mehrfach überarbeitet werden, weil nicht von Beginn an klar ist, welche Daten wo gebraucht werden. Dadurch kann die Flexibilität eingeschränkt werden und Anpassungen sind mit einem höheren Aufwand verbunden. Wenn der Zustand nicht global ist, kann er auch nicht direkt zentral persistiert werden, um beispielsweise den Zustand bei einem Neuladen beizubehalten.

8 | Client

Mit der clientseitigen Anwendung können Nutzer im Browser mithilfe der facettierten Suche Datenanalysen durchführen. Der Aufbau des Webclients und die Bausteine, die in Abbildung 8.1 farblich hervorgehoben sind, werden im Folgenden beschrieben.

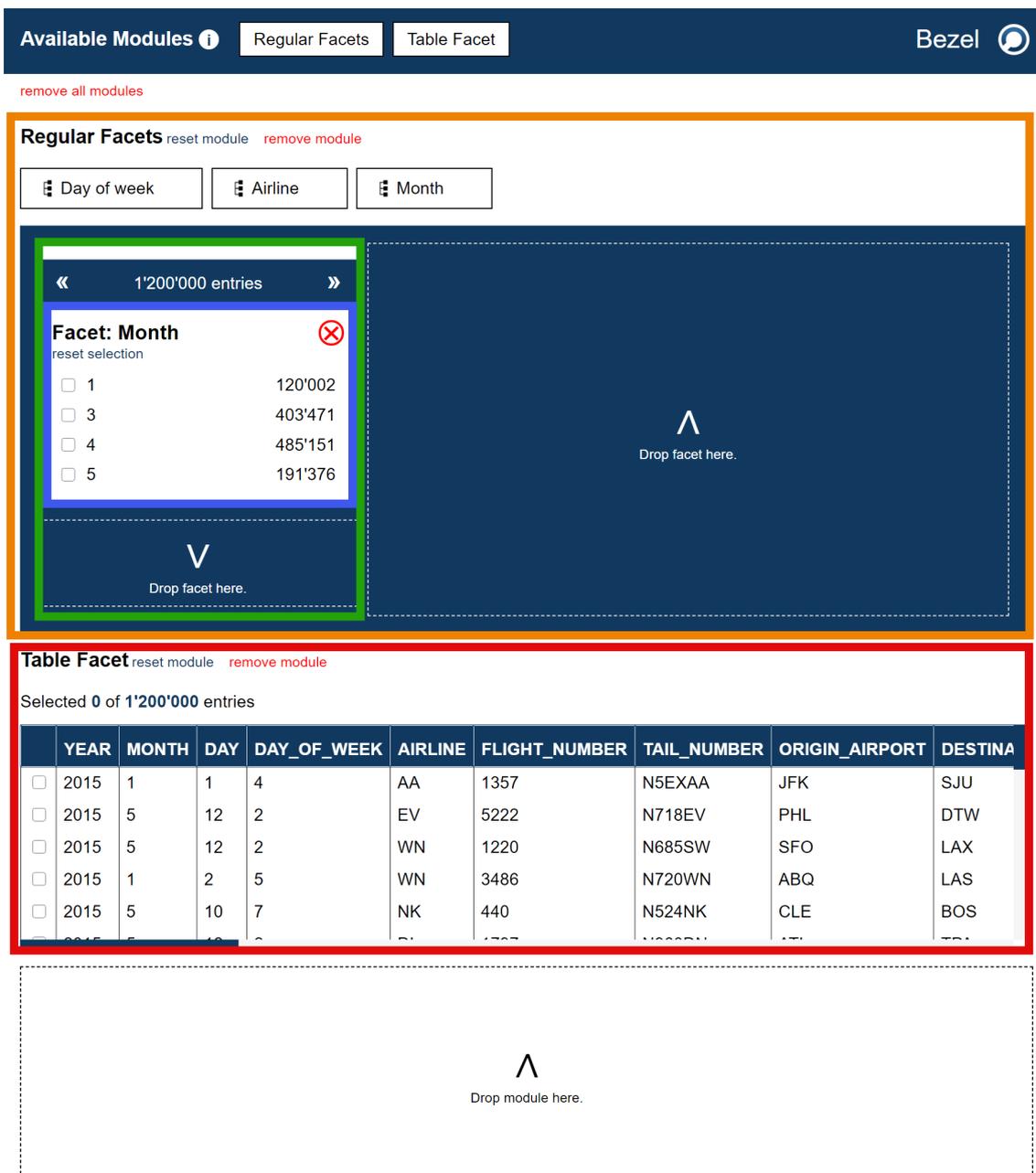


Abbildung 8.1: Screenshot des Prototyps

8.1 MainController

Der MainController steuert das Modulsystem, wie es in Abschnitt 6.3 beschrieben ist. Jede Aktion, die in der ganzen Applikation Auswirkungen haben kann, wird ebenfalls über den MainController ausgelöst. Der nachfolgende JSDoc-Ausschnitt zeigt, welche Funktionen im MainController verfügbar sind.

```

1 /**
2  * Controller for managing the modules.
3  * @typedef {Object} MainController
4  * @property {function(BezelModule):void} addModule
5  * @property {function} onModuleAdd
6  * @property {function(BezelModule):void} activateModule
7  * @property {function(function(BezelModule):void):void}
   *   onModuleActivate
8  * @property {function(BezelModule):void} deactivateModule
9  * @property {function(function(BezelModule):void):void}
   *   onModuleDeactivate
10 * @property {function(BezelModule, Facet, number, Operation):
   *   void} activateFacet
11 * @property {function(BezelModule, Facet):void} deactivateFacet
12 * @property {function(BezelModule, Facet,
   *   ObservablesCollection):void} makeSelection
13 * @property {function(BezelModule, BezelContainer, number):
   *   void} moveInFacet
14 * @property {function():void} reset
15 * @property {function(BezelModule):void} resetModule
16 * @property {function(BezelModule, Facet):void} resetFacet
17 * @property {function(string):BezelModule} getModuleByName
18 */

```

8.1.1 Modulverwaltung

Der Controller bietet die Möglichkeit, neue Module zur Auswahl hinzuzufügen. Aktuell wird dies bei der Initialisierung der Anwendung ausgeführt und kann nicht vom Nutzer manuell gemacht werden. Diese Funktionalität kann erst zu einem späteren Zeitpunkt im User Interface integriert werden, wenn die Möglichkeit vorhanden ist, Facetten und Module zur Laufzeit zu definieren.

Im Kontext der Anwendung wird beim Aktivieren eines Moduls die View erstellt und dem DOM angefügt. Mit der Funktion `onModuleActivate` können aus der View anhand des Observer Pattern, Listener registriert werden. Diese Listener werden dann informiert, wenn auf dem MainController mit `activateModule` ein verfügbares Modul aktiviert wird.

Das Model besteht aus zwei Listen. Die erste Liste enthält alle verfügbaren Module und wird mit `addModule` manipuliert, die zweite enthält die aktiven Module und wird durch `activateModule` und `deactivateModule` geändert.

8.1.2 Auswahl manipulieren

Eine Änderung der aktuellen Selektion wird im `MainController` mit `makeSelection` ausgelöst. Dieser Funktion wird übergeben, in welchem Modul, auf welcher Facette und für welchen Facettenwert die Auswahl geändert werden soll. Der Auftrag, die Auswahl zu ändern, wird an das Zielmodul weitergeleitet. Mit der Funktion `update` wird auf allen nachfolgenden Modulen eine Aktualisierung ausgelöst, um die angezeigten Daten der neuen Auswahl entsprechend anzupassen.

8.2 Module

Module können verschiedene Formen annehmen. In Abbildung 8.1 sind die zwei implementierten Module `RegularFacetModule` (orange) und `TableModule` (rot) zu sehen. Alle Module müssen das folgende Interface erfüllen.

```

1 /**
2 * @typedef {Object} BezelModule
3 * @property {string} id
4 * @property {string} name
5 * @property {string} type
6 * @property {function(ObservablesCollection):void}
   makeSelection
7 * @property {function():void} reset
8 * @property {function(Filter):void} update
9 * @property {Filter[]} getFilter
10 * @property {BezelModule} getNewInstance
11 */

```

8.2.1 Facetten verwalten

Die implementierten Module `RegularFacetModule` und `TableModule` unterscheiden sich strukturell dadurch, dass im `TableModule` nur eine Facette verwendet wird, während das `RegularFacetModule` eine beliebige Anzahl verwalten kann und damit auch mehr Funktionalität anbietet. Im `RegularFacetModule` können Facetten hinzugefügt, aktiviert, deaktiviert und verschoben werden.

8.2.2 Zwischenstruktur

Das `RegularFacetModule` hat ausserdem die Eigenheit, dass die aktiven Facetten nicht direkt im Modul abgelegt sind. Sie sind in einer Zwischenstruktur, dem `FacetContainer`, zu finden, die in Abbildung 8.1 grün markiert ist. Mehrere Facetten, die sich im gleichen `FacetContainer` befinden, werden mit einer Disjunktion verbunden. Eine Struktur dieser Art muss mindestens die Funktionen `update` und `getFilter` anbieten. Die Funktion `update` löst auf allen enthaltenen Facetten eine Aktualisierung aus. Die Funktion `getFilter` gibt ein Objekt zurück, das die Selektionen der enthaltenen Facetten repräsentiert.

8.2.3 Auswahl manipulieren

Wie im `MainController` gibt es auch in den Modulen eine Funktion für das Durchführen einer Selektion. Die Funktion nimmt die Zielfacette, den Zielfacettenwert und den Filter, der in den vorangehenden Modulen gesetzt wurde, entgegen. Der Facettenwert, der selektiert werden soll, wird an die Zielfacette weitergegeben und alle nachfolgenden Facetten werden aktualisiert.

8.2.4 Filter zusammensetzen

Mit `getFilter` kann ein Array kreiert werden, das die aktuelle Selektion auf dem Modul repräsentiert. Dieses Array wird mitgeschickt, wenn vom Service aktualisierte Daten abgefragt werden.

8.3 Facet

Eine Facette ist in diesem Kontext die Struktur, die Daten aus dem Datensatz speichert. Jede Facette muss das folgende Interface implementieren.

```

1 /**
2 * @typedef {Object} Facet
3 * @property {function(Filter[]):void} update
4 * @property {function(ObservablesCollection):void}
   makeSelection
5 * @property {function():Filter} getFilter
6 */

```

8.3.1 Facettenimplementierungen

Im Prototyp sind zwei verschiedene Darstellungsarten von Facetten implementiert. Die `RegularFacet` (blau markiert in Abbildung 8.1) kann für eine vorgegebene Dimension die Facettenwerte verwalten. Die Facettenwerte können sowohl als Strings wie auch als Numbers betrachtet werden. Abhängig davon sind unterschiedliche Darstellungen möglich (siehe Abbildung 8.2). Facettenwerte, die als Text interpretiert werden, können entweder in einer einfachen Liste oder, durch eine Gruppierung ergänzt, als Hierarchie dargestellt werden. Für Zahlen gibt es die Möglichkeit, Bereiche zusammenzufassen. Auf die individuellen Werte der einzelnen Einträge kann nicht zugegriffen werden, wenn die Facette nach Bereichen zusammengefasst ist.

In der `TableFacet` (rot markiert in Abbildung 8.1) sind für einen Teil der Einträge aus dem Datensatz alle Attribute bekannt. In der Konfigurationsdatei der Anwendung kann festgelegt werden, wie viele Einträge in der Tabelle initial geladen werden. Weitere werden mit Lazy Loading nachgeladen. Es kann bei der Initialisierung auch ein Array mit Spaltennamen übergeben werden, die in der Tabellendarstellung ignoriert werden sollen.

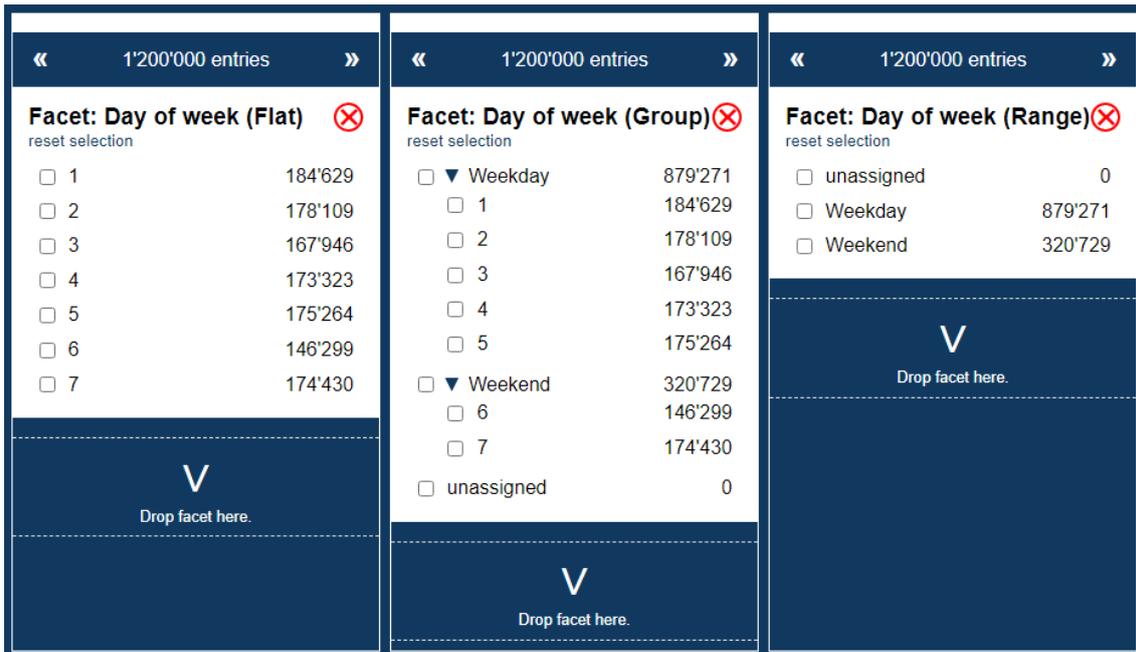


Abbildung 8.2: Darstellung als einfache Liste (links), als Gruppierung von Strings (mitte) und Zusammenfassung der Werte nach Zahlenbereichen (rechts)

8.4 Asynchrone Aktualisierung

Die Funktion zum Aktualisieren von Facetten ist asynchron und wird auf jeder Facette einzeln aufgerufen. Die Asynchronität ist wichtig, weil bei Serveranfragen a priori nicht bekannt ist, wie lange auf eine Antwort gewartet werden muss. Es kann auch nicht vorausgesetzt werden, dass die Antworten auf Serveranfragen in der Reihenfolge zurückkommen, in der sie geschickt wurden. Die Abbildung 8.3 zeigt eine Situation, wo die Antwort auf den zweiten Request vor der Antwort auf den ersten zurückkommt. Dies ist insbesondere der Fall, wenn die Antwort auf den zweiten Request vom Cache abgerufen werden kann, ohne eine Server- oder Datenbankabfrage machen zu müssen.

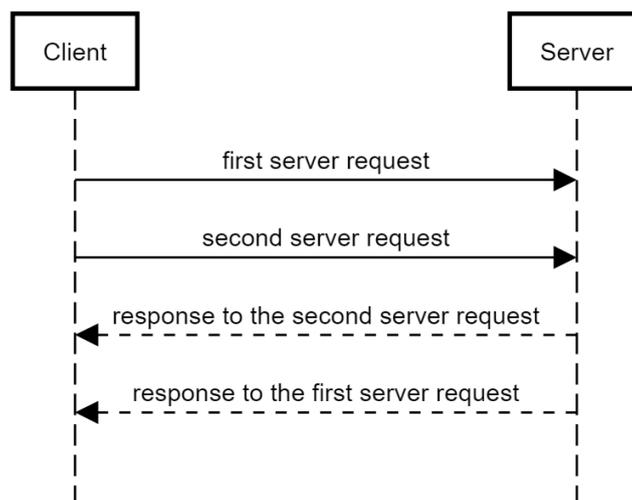


Abbildung 8.3: Beispiel eines asynchronen Serverrequests

Das Problem ist, dass Antworten auf frühere Requests möglicherweise nicht mehr relevant sind. Wenn mit den erhaltenen Informationen eine Aktualisierung auf dem Model durchgeführt wird, kann ein inkonsistenter Zustand entstehen, in dem die gespeicherten Daten nicht mehr mit der getätigten Selektion übereinstimmt. Im Client gibt es unterschiedliche Möglichkeiten, mit dieser Situation umzugehen.



8.4.1 Abfragen blockieren

Die Problematik kann umgangen werden, wenn man verhindert, dass mehrere Anfragen geschickt werden können, bevor die Serverantwort erhalten wurde. Dieser Ansatz ist aber aus zwei Gründen nicht geeignet. Es ist einerseits ein Widerspruch zur Grundidee der Asynchronität für Serveranfragen und der Nutzer soll die Anwendung auch in der Wartezeit benutzen können. Andererseits würde das Problem auftreten, dass die Anwendung nicht mehr benutzt werden kann, wenn vom Server aufgrund eines Fehlers keine Antwort kommt.

Wenn ein Nutzer beispielsweise in einer Facette drei Attribute auswählen möchte, müsste er nach jeder Auswahl erst darauf warten, dass die Informationen aktualisiert werden, bevor er eine weitere Auswahl treffen kann. Weil sich der Nutzer in diesem Fall nur für die Situation interessiert, wo alle drei Attribute ausgewählt sind, bedeutet dies aus seiner Sicht, dass die Gesamtzeit zum Erreichen der gewünschten Auswahl deutlich länger wird.

8.4.2 Abfragen ignorieren

Die Implementierung, die im Prototyp zum Einsatz kommt, überprüft nach Erhalt der Antwort vom Server, ob die erhaltenen Daten für den aktuellen Zustand der Anwendung noch relevant sind und passt das Model nur an, wenn in der Zwischenzeit keine Änderungen an der Auswahl vorgenommen wurden. Wenn der Filter nicht mehr mit dem Filter übereinstimmt, der vor der Anfrage gesetzt war, bedeutet dies, dass die Informationen nicht mehr gebraucht werden. Dies führt dazu, dass die Antwort ignoriert wird.

Für die `TableFacet` muss ein Spezialfall betrachtet werden. Wie in Abschnitt 6.4 beschrieben, ändert sich die Auswahl, wenn ein selektierter Eintrag aus der Ergebnismenge fällt. Während des Wartens auf eine Antwort vom Server entsteht die Situation, dass ein Eintrag in der Tabelle noch angezeigt wird, auch wenn er nicht mehr in der Ergebnismenge wäre, weil das Model erst angepasst wird, wenn die neuen Daten verfügbar sind. Dieser Eintrag ist dann im User Interface noch selektierbar, er darf aber nicht der Selektion hinzugefügt werden, da sonst ein Wert selektiert wäre, der gar nicht in der Ergebnismenge enthalten ist.



9 | Server

Als Server wurde Express für Node.js verwendet. Express bietet den Vorteil einer simplen Implementierung durch ein minimalistisches Framework. Wie für den Client kann JavaScript verwendet werden. Der Server kann in drei Teile gegliedert werden: die Schnittstelle zum Client, die Vorbereitung der Datenbankabfragen und die Datenbank.



Abbildung 9.1: Serveraufbau

9.1 Datenbankeinbindung

9.1.1 Dispatcher

Die Schnittstelle ist in zwei dispatcher Dateien aufgeteilt. Im `file-reader-dispatcher.mjs` sind zwei Endpunkte definiert. `domain/file-reader/generate-id` ist zuständig, neue IDs in der Datenbank zu generieren, die für den Client lesbar sind. Über den zweiten Endpunkt `domain/file-reader/ensure-indexes` soll das Indexieren ermöglicht werden. Der `file-reader-dispatcher.mjs` wird benötigt, wenn ein neuer Datensatz in die Datenbank eingelesen werden soll. Dies ist mit dem im Rahmen des Projekts erarbeiteten Prototyp noch nicht möglich. Die Schnittstelle für die Erweiterung wird für diese Erweiterung schon zur Verfügung gestellt, jedoch werden noch nicht alle benötigten Endpunkte für das Übermitteln der Daten angeboten.

Der `filter-dispatcher.mjs` ist die Schnittstelle, die für die Interaktion mit den Facetten benötigt wird. Mit `domain/filter/table-entries` können alle Dateneinträge, die den Filter erfüllen, abgefragt werden. Der zweite Endpunkt `domain/filter/facet-values` ist dafür zuständig, die Trefferanzahl pro Facettenwert zu kalkulieren und zurückzugeben.

Die definierten Endpunkte rufen datenbankspezifische Funktionen auf, die die Datenbankabfrage bereitstellen und auslösen.

9.1.2 Spezifische Datenbankimplementation

Im Prototyp werden die datenbankspezifischen Funktionen für die MongoDB bereitgestellt. Um das Einbinden von weiteren Datenbankimplementationen zu ermöglichen, werden zwei Interfaces angeboten. Wie in Abbildung 9.2 ersichtlich wird ein Interface für das Einlesen und ein zweites für das Filtern der Daten zur Verfügung gestellt.

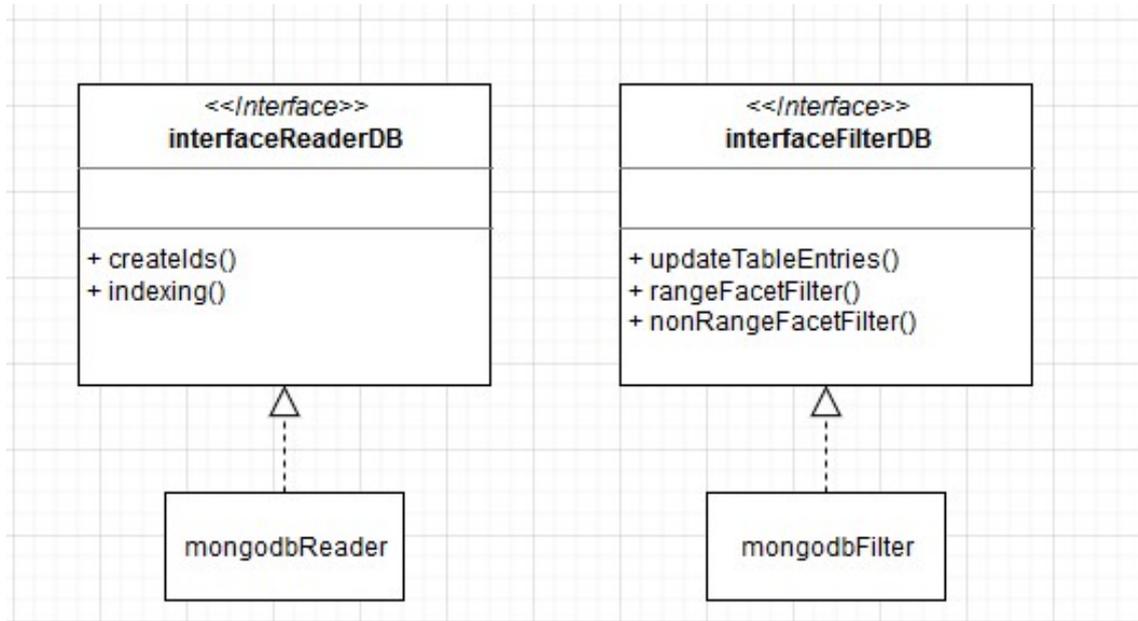


Abbildung 9.2: Übersicht der zwei zur Verfügung gestellten Interfaces

Diese definieren die Funktionen, die vorhanden sein müssen, um erfolgreiche API-Anfragen gewährleisten zu können. Mit der verwendeten Implementierung von Interfaces kann überprüft werden, ob die Vorgaben für ein konkretes Interface erfüllt werden. Es kann nicht geprüft werden, ob die Funktionen richtig implementiert sind.

```

1 /**
2  * Defines a general interface
3  * @interface
4  * @param {string} name - Name of concrete interface
5  * @param {string[]} methods - Method names of concrete
   interface
6  * @property {function(Object)} ensureImplements - Checks if
   given object corresponds with expected concrete interface
7  */

```

Mit Hilfe dieses benutzerdefinierten Interface können konkrete Interfaces erstellt werden. Im folgenden Auszug wird das Interface AccessFilterDB erstellt, das drei Methoden definiert.

```

1 const interfaceFilterDB = new Interface('AccessFilterDB', [
   updateTableEntries', 'nonRangeFacetFilter', '
   rangeFacetFilter']);

```

Die konkreten Interfaces besitzen dann den Objektkonstruktor Interface [18]. Mit der vom Interface bereitgestellten Methode `ensureImplements` kann überprüft werden, ob ein Objekt die Methoden des Interface implementiert. Falls dies nicht der Fall ist, wird eine Exception geworfen.

9.1.3 Datenbankimplementierung

Datenbankimplementierungen werden in einem separaten Package `server/concreteDB` abgelegt. Der Provider stellt die entsprechenden Funktionen zur Verfügung, um die benötigte Datenbankimplementierung einzubinden. Die zu benutzende Datenbank kann dann wie folgt definiert werden:

```
1 /**
2  * Define here concrete database implementation
3  */
4 useMongoDB();
```

9.2 Optimierung der Datenbankperformance

Mithilfe von Indexes kann die Datenbankperformance erhöht werden. Die durchgeführten Zeitmessungen demonstrieren die Effekte von Indexes. Es handelt sich dabei um Echtzeitmessungen, bei denen die verstrichene Zeit zwischen der Selektion im User Interface und der zugehörigen Antwort bestimmt wird. Solche Messungen hängen von vielen externen Faktoren wie der Verwendung von lokalen Caches oder der Rechnerauslastung ab und dienen damit lediglich als anekdotische Evidenz. Die Messungen können nicht als Grundlage für Projektionen genommen werden, wie sich die Antwortzeiten für grössere Datensätze entwickeln.

9.2.1 Vorgehen

Für die Messungen wurde der Datensatz mit Flugverspätungen verwendet. Vier vordefinierte Szenarien wurden mit und ohne Indexes jeweils fünfmal durchgespielt. Für alle Kombinationen wurde dann der Mittelwert berechnet. Die angegebenen Zeiten beschreiben, wie lange der Client auf die Antwort des Servers warten musste.

Die Szenarien lauten wie folgt:

1. Zwei Tabellenfacetten hinzufügen und auf der ersten Facette 1, 2, 3 und 4 Selektionen machen. Gemessen wird die Zeit für die Aktualisierung der zweiten Facette.
2. Reguläre Facetten „Day of week“ und „Airline“ hinzufügen. 1, 2 und 3 Selektionen auf „Day of week“ machen. Gemessen wird die Zeit für die Aktualisierung von „Airline“.
3. Reguläre Facetten „Day of week“ und „Airline“ sowie Tabellenfacette hinzufügen. Auf den regulären Facetten jeweils 0, 1 und 2 Selektionen machen. Gemessen wird die Zeit für die Aktualisierung der Tabellenfacette.

9.3. PROBLEME

4. Reguläre Facetten „Day of week“ und „Month“ (als range-Facette) hinzufügen. 1, 2 und 3 Selektionen auf „Day of week“ machen. Gemessen wird die Zeit für die Aktualisierung von „Month“.

Szenario	Selektion pro Facette	ohne Indexes [ms]	mit Indexes [ms]
1	1	2446	9
1	2	2394	8
1	3	2441	9
1	4	2450	8
2	1	897	440
2	2	1010	783
2	3	1139	1190
3	1 + 0	844	89
3	1 + 1	837	309
3	2 + 1	888	250
3	2 + 2	900	500
4	1	910	383
4	2	1025	797
4	3	1158	1152

Tabelle 9.1: Resultate der Zeitmessungen

9.2.2 Resultate

Die Zeitmessungen zeigen, dass eine Verbesserung insbesondere dann auftritt, wenn nur eine Spalte des Datensatzes betroffen ist. Bei der Auswahl auf den Tabellenfacetten im ersten Szenario muss nur die ID beachtet werden. Für komplexere Suchanfragen müsste der Einfluss der Indexes noch weiter untersucht werden.

9.3 Probleme

Die Auslagerung des Filterprozesses auf die Datenbank hat den Nachteil, dass das Filtern nicht generalisiert werden kann. Entwickler, die eine andere Datenbank anbinden möchten, müssen den Filterprozess neu implementieren. Es steht ein Interface mit den zu implementierenden Funktionen zur Verfügung, die Logik ist jedoch nicht generalisierbar, da dies von der Datenbank abhängig ist.

Express.js ist für den Entwicklungsprozess ein praktisches und übersichtliches Framework, ermöglicht jedoch keine echte Nebenläufigkeit, weil Node.js Single-Threaded ist. Um Multithreading zu ermöglichen werden mehrere Instanzen benötigt. Der Entwickler wird jedoch gezwungen, Express als Server zu verwenden. Da im Projekt, der Fokus nicht auf der Servertechnologie lag wurde diese Problem nicht behandelt.

9.3. PROBLEME

Durch die Wahl von MongoDB zeigten sich Limitationen, die eine dokumentenorientierte Datenbank mit sich bringt. Das Hauptproblem liegt in der ineffizienten Sortierung von Dokumenten. Darum wurde im Rahmen dieses Projekts keine Sortierfunktion von Einträgen für den Benutzer bereitgestellt. Die Schnittstelle bietet jedoch die Möglichkeit, entsprechende Parameter zu übergeben. Somit kann die Sortierung in einem zukünftigen Projekt mit Hilfe von einer anderen Datenbank implementiert werden.

10 | User Interface

Die Usability und die Ästhetik der Benutzeroberfläche stehen nicht im Zentrum dieser Arbeit. Die Anforderung ist, dass die Anwendung für Expertenbenutzer nach einer zehnmütigen Einführung bedienbar sein muss. Dieses Kapitel behandelt Aspekte, die im User Interface umgesetzt sind.

10.1 Interaktionen

Im Folgenden werden die Interaktionen beschrieben, wie sie im Prototyp implementiert sind. Da einige Interaktionen über Drag and Drop funktionieren, kann die Anwendung nicht auf Mobilgeräten mit Touchscreen verwendet werden.

10.1.1 Hinzufügen und entfernen

Sowohl Module wie auch Facetten können mit Drag und Drop hinzugefügt werden. Jedes hinzugefügte Element hat einen Button, über den es wieder entfernt werden kann. Facetten können innerhalb des Moduls auf zwei Arten verschoben werden. Mit Drag und Drop kann eine Gruppe von Facetten an eine beliebige Stelle im Modul verschoben werden. Alternativ hat es auch Buttons, über die eine Verschiebung um eine Position nach links oder rechts vorgenommen werden kann.

10.1.2 Auswählen

Einzelne Facettenwerte können dem Filter hinzugefügt werden. Es können auch ganze Gruppen selektiert werden.

10.1.3 Zurücksetzen

Jedes Element der Anwendung kann auch zurückgesetzt werden. Auf einer Facette kann mit einem Mausklick die ganze Selektion entfernt werden. Wenn das Modul zurückgesetzt wird, werden alle Facetten und dadurch auch deren Selektion entfernt. Es gibt auch einen Button, über den die gesamte Anwendung zurückgesetzt werden kann.

10.2 Usability

Im Abschnitt 7.2 ist beschrieben, dass die Views vom Rest der Anwendung entkoppelt sind und dadurch ein Austausch der Benutzeroberfläche möglich ist. Die Usability kann so verbessert werden, ohne die zugrundeliegende Logik zu verändern.

10.2.1 Feedback

Der Nutzer muss klares Feedback darüber erhalten, welche Auswirkungen seine Interaktionen haben und in welchem Zustand sich die Anwendung befindet.

Anzahl Einträge

Eine aus Nutzersicht sehr wichtige Angabe ist die Anzeige, wie viele Einträge in einer Facette noch verfügbar sind. Es hilft dem Nutzer, die Suchanfrage nachzuvollziehen, wenn schrittweise betrachtet werden kann, wo in der Kombination der Facetten welche Einschränkungen vorgenommen wurden.

Loading-Indikator

Der Nutzer kann sehen, dass die Anwendung noch auf eine Serverantwort wartet und die aktuell angezeigten Daten möglicherweise nicht mit der Auswahl übereinstimmen. Die implementierte Anzeige, dass noch nicht fertig geladen wurde, gibt keine spezifischen Angaben darüber, was noch nicht aktualisiert ist. Idealerweise wird direkt auf den Facetten angezeigt, wo noch Änderungen ausstehen. Es kann auch signalisiert werden, welche Funktionen temporär nicht verfügbar sind. Beispielsweise ist es nicht möglich, auf einer TableFacet eine Auswahl zu treffen solange die Aktualisierung noch nicht abgeschlossen ist.

10.2.2 Kombination von Facetten

Die Benutzeroberfläche muss visualisieren, wie die Facetten kombiniert werden. Innerhalb von Modulen gelten die folgenden Regeln. Facetten, die untereinander stehen, werden mit einer Disjunktion (Oder) verbunden. Facetten, die nebeneinander stehen, werden mit einer Konjunktion (Und) verbunden.

11 | Testing

Die Implementierung des Clients wird mit Unittests und End-to-End-Testing überprüft. Das Testing wird ohne zusätzliche Abhängigkeiten oder externe Frameworks durchgeführt. Die Anwendung bietet mit einer separaten URL einen alternativen Einstiegspunkt, über den die Tests direkt im Browser ausgeführt werden können. Die API und die Datenbank werden manuell mittels Szenarien getestet.

11.1 Testumgebung

Die verwendete Testumgebung basiert auf einer Implementierung, die im Modul Web Clients [19] eingeführt wird. Für das End-to-End-Testing wurde die Umgebung so ergänzt, dass auch asynchrone Funktionen getestet werden können.

In den Tests kann definiert werden, welche Bedingungen an bestimmten Stellen erfüllt sein müssen. Dafür gibt es zwei Methoden. Es kann entweder geprüft werden, ob zwei Werte identisch sind oder ob ein boolescher Ausdruck wahr ist.

Eine Übersicht der Testresultate wird direkt im Browser angezeigt. Bei fehlschlagenden Tests wird angegeben, an welcher Stelle sie sich befinden. Zudem gibt es in der Konsole weitere Informationen. Wenn zwei Werte verglichen werden, werden sowohl der erwartete wie auch der tatsächliche angezeigt.

11.2 Unit

Die für die Models implementierten Datenstrukturen werden durch Unittests abgedeckt. Weil die Datenstrukturen generisch implementiert sind, können sie ohne Wissen über die konkreten Daten, die bei der Verwendung darin abgelegt werden, getestet werden.

Die Korrektheit der Controller wird auch teilweise mit Unittests verifiziert. Die Tests decken jedoch nicht den ganzen Umfang ab, da aufgrund der Abhängigkeiten von andern Applikationsteilen umfangreiches Mocking nötig wäre. Die Testumgebung bietet dafür keine Unterstützung. Die für die Tests des Modulsystems manuell implementierten Mock-Module sind schlecht wartbar und unflexibel. Deshalb werden die Controller im Rahmen der End-to-End-Tests geprüft.

11.3 End-to-End

Das End-to-End-Testing ist mit automatisierten Tests des User Interface umgesetzt. Mit der Methode `click`, die für Elemente im DOM definiert ist, wird das Durchführen eines Mausklicks des Nutzers simuliert. Das Testing wird unter Verwendung eines lokalen Service durchgeführt. Dadurch können die Tests von der serverseitigen Implementierung unabhängig gemacht werden.

Die Tests für unterschiedliche Teile des Clients werden manuell ausgelöst. Es ist nicht praktikabel, diese Tests asynchron auszuführen, weil die Tests den DOM verändern können. Wenn mehrere Tests Änderungen vornehmen, können Überprüfungen fehlschlagen, obwohl die Implementierung korrekt ist.

11.3.1 Verzögerung

Interaktionen, die Änderungen am DOM vornehmen oder andere asynchrone Funktionen aufrufen, können dazu führen, dass der Test eine Bedingung abfragt, bevor die Änderung vollständig ausgeführt wurde. Das führt dazu, dass ein Test fehlschlägt, obwohl die Implementation korrekt ist. Als Gegenmassnahme wurde die Möglichkeit implementiert, nach einem simulierten Klick eine Verzögerung einzubauen. Die Anpassung am User Interface kann so abgeschlossen werden, bevor die Korrektheit überprüft wird. Diese Methode hat den Nachteil, dass die Dauer der Verzögerung willkürlich und fix gewählt werden muss. Abhängig von der Komplexität der Veränderungen wird mehr Zeit benötigt. Eine längere Verzögerung bedeutet auch, dass die Gesamtdauer der Tests signifikant grösser werden kann. Für den Umfang dieses Projekts fällt diese Problematik nicht ins Gewicht, muss aber für grössere Applikationen mit umfangreichem Testing beachtet werden. Die Möglichkeit, die Verzögerung manuell festzulegen, hat für dieses Projekt sogar den Vorteil, dass die Tests besser veranschaulicht werden können. Mit einer genügend grossen Verzögerung zwischen den Aktionen kann ein Beobachter nachvollziehen, welche Operationen in den Tests durchgeführt werden.

11.3.2 Einschränkungen

Die Tests müssen in allen Browsern, die unterstützt werden sollen, separat ausgeführt werden, um die Kompatibilität sicherzustellen. Die Methode `Array.prototype.flat()` beispielsweise wurde mit ES2019 eingeführt [20]. Sie gibt für ein verschachteltes Array ein neues Array zurück, das auf eine Ebene reduziert wurde. In Microsoft Edge kann die Methode erst seit der Umstellung auf Chromium verwendet werden. Es genügt daher auch nicht, nur in der aktuellsten Version zu testen, denn der Rollout der neuen Version von Edge wurde noch nicht abgeschlossen. Weil der Prototyp auf Expertennutzer ausgelegt ist und die Kompatibilität in diesem Projekt nicht im Zentrum steht, werden ältere Browserversionen nicht unterstützt. Auch die Darstellung des User Interface kann von Browser zu Browser unterschiedlich sein und muss manuell geprüft werden. Die Funktionalität kann aber geprüft werden, denn die Methode `click` auf HTML-Elementen funktioniert unabhängig davon, ob die Elemente sichtbar und wie sie platziert sind.

Mit der verwendeten Einrichtung ist es auch schwierig, den Prototyp auf Memory Leaks zu testen. Deshalb muss die Logik des Codes direkt analysiert werden, um Leaks

aufzudecken. Eine häufige Quelle ist bei Verwendung des Observer Pattern, dass registrierte Listener nicht entfernt werden, wenn sie nicht mehr benötigt werden.

11.4 API

Die API-Tests wurden nicht mit der beschriebenen Testumgebung durchgeführt. Um sicherzustellen, dass die Anfragen vom Client richtig verarbeitet werden, wurden Szenarien erarbeitet und mit den Eckdaten festgehalten. Die Eckdaten bestehen aus dem gesetzten Filter und den erwarteten Zahlenwerten. Die Zahlenwerte bestehen aus der Gesamttrefferanzahl und der Trefferanzahl pro Facettenwert. Die Szenarien wurden mithilfe des originalen Datensatzes erarbeitet und mit dem realen Datensatz durchgespielt. Die Ergebnisse sind die Referenzwerte für die Tests, wobei jeder Test einem Szenario entspricht.

Der Testprozess besteht daraus, bei gestarteter Applikation die Szenarien einzeln abzuarbeiten und die Filter entsprechend zu setzen. Die angezeigten Werte im Browser können dann mit den Soll-Werten in den Szenarien abgeglichen werden. Stimmt ein Wert nicht mit dem Soll-Wert überein, schlägt der Test für das entsprechende Szenario fehl.

11.4.1 Einschränkungen

Die Tests müssen alle manuell durchgeführt werden, was im Vergleich zu automatisierten Tests weniger effizient ist. Bei einem Fehlschlagen der Tests kann man als Tester auch nicht sogleich die Fehlerquelle erkennen. Der Debugging-Prozess ist daher anspruchsvoller.

Es wurden nicht alle möglichen Szenarien aller Beispieldatensätze, die im Rahmen dieses Projekts verwendet wurden, definiert und festgehalten. Das Testing wurde spezifisch auf den ATP-Datensatz ausgelegt.

12 | Endprodukt

Dieses Kapitel beschreibt, in welcher Form der Prototyp dieser Arbeit vorliegt. Der Prototyp wird als eigenständige Applikation gestartet. Es handelt sich nicht um eine Library, die in eine bestehende Anwendung eingebaut wird, sondern um eine Vorlage, auf deren Basis weitere Entwicklungen vorgenommen werden können.

12.1 Nutzung

Die Anwendung kann von Expertennutzern zur Datenanalyse mittels facettierter Suche verwendet werden. In der Dokumentation für Anwender ist beschrieben, wie die Konfiguration geändert werden kann, um einen neuen Datensatz untersuchen zu können.

12.2 Erweiterbarkeit

Entwickler können den Prototyp übernehmen und an ihre Anforderungen anpassen. Die Änderungen werden direkt im Quellcode vorgenommen. In der Entwicklerdokumentation ist beschrieben, wie Ergänzungen in die bestehende Codebasis eingebaut werden können. Neue Module und Facetten können in den Prototyp integriert werden. Auch die Datenquelle kann ausgetauscht werden, wenn beispielsweise ein anderer Datenbanktyp verwendet werden soll.

Teil III

Schluss

13 | Fazit

Die Facettensuche gewinnt im Web immer mehr an Bedeutung, da sie dem Benutzer eine systematische Eingrenzung der Suchmenge ermöglicht. Für diese interaktive Suche im Web wurde im Rahmen dieses Projekts eine innere Datenstruktur aus Beispieldatensätzen und Szenarien extrahiert und implementiert. Das Endprodukt bietet eine Plattform, um diese innere Datenstruktur auszutesten und dient auch zur Veranschaulichung des Mehrwerts der facettierten Suche. Die Arbeit zeigt die Vielzahl von Faktoren auf, die bei der Implementierung einer effizienten inneren Datenstruktur beachtet werden müssen. Es lohnt sich, eine generalisierte Lösung der inneren Datenstruktur zur Verfügung zu haben.

Im Projekt wurde erarbeitet, dass die innere Datenstruktur in Facettenbeschreibung, Abbildung der getätigten Selektionen und die Speicherung der Trefferanzahlen unterteilt werden kann. Es ist möglich, die Facettenbeschreibung dahingehend zu generalisieren, dass verschiedene Facettentypen abgebildet werden können. Bei der Definition dieser Beschreibung gibt es mehrere Varianten, wie man verschiedene Spezialfälle behandelt. Die Handhabung dieser Spezialfälle sollte man auf das Zielpublikum und die Definition der Facette ausrichten.

Die optimale Datenstruktur für die Speicherung der aktuellen Trefferanzahl und der getätigten Selektion kann für verschiedenen Darstellungen einer Facette unterschiedlich ausfallen. In diesem Projekt wurde eine Baumstruktur für reguläre Facetten und eine Liste für die Tabellenfacetten gewählt.

Die Performance der Facettensuche ist nicht nur von der inneren Datenstruktur abhängig, sondern auch von der Wahl der Speicherdatenstruktur. Die Wahl einer Datenbank ist sinnvoll, wenn die Anzahl der Einträge im Datensatz im Millionenbereich liegt. Die Effizienz kann durch die optimale Wahl des Datenbanktyps für die Funktionen Filtern und Sortieren gesteigert werden.

Das Endprodukt dieser Arbeit ist eine Client-Server-Applikation. Der Einsatz von Interfaces macht es möglich, zwischen verschiedenen Datenbanken als Filter- und Sortier-Tool zu wechseln und neue Datenbankimplementationen hinzuzufügen. Durch den modularen Aufbau der Facetten und der Verwendung einer MVC-Architektur kann der Client um Facettenarten und Darstellungen erweitert werden. In einem nächsten Schritt würde die Applikation so erweitert werden, dass sie in Form eines Frameworks verwendet werden kann.

Der Projektumfang hat sich im Verlauf der Arbeit geändert. Das automatisierte Einlesen eines Datensatzes war anfangs ein Bestandteil des Projekts. Der Fokus verschob sich aber auf den modularen Aufbau der verschiedenen Facettenarten.

14 | Ausblick

Basierend auf den Resultaten dieser Arbeit und der Implementierung des Prototyps gibt es mögliche Weiterentwicklungen, die in den folgenden Abschnitten beschrieben werden. Diese Ergänzungen und zusätzlichen Funktionen konnten aus unterschiedlichen Gründen im Rahmen des Projekts nicht umgesetzt werden. Weil der Fokus auf der grundsätzlichen Implementierung der inneren Datenstruktur lag, wurden beispielsweise alternativen Visualisierungen für die bestehenden Facetten nicht in den Umfang aufgenommen. Auch der Entwicklung des umliegenden Systems wurde eine tiefere Priorität zugewiesen. Deshalb ist es im Prototyp nicht möglich, Datensätze direkt einzulesen und Facetten im User Interface zu definieren.

14.1 Usability des Prototyps

Die Usability kann auf zwei Ebenen verbessert werden. Zum einen auf der Ebene der Visualisierung im User Interface wo eruiert werden muss, welche Änderungen die Anwendung intuitiver und übersichtlicher gestalten können. Zum andern gibt es auch Erweiterungen der Funktionalität, die für die Usability wichtig sind.

14.1.1 User Interface

In Zusammenarbeit mit der Zielgruppe können Anpassungen an der Darstellung der Anwendung vorgenommen werden. Die Grössenverhältnisse müssen an die Bedürfnisse der Nutzer angepasst werden. Eine kleinere Schriftart ist möglicherweise besser geeignet, um mehr Informationen auf einen Blick erfassen zu können.

14.1.2 Modulsystem

Das Modulsystem kann dynamischer gestaltet werden. Die Möglichkeit, Module temporär ein- und auszublenden, gäbe dem Nutzer einen besseren Überblick über den Zustand der Anwendung. Analog zu den regulären Facetten sollen auch Module neu angeordnet werden können.

14.1.3 Facetten definieren

Die Beschreibung der Facetten ist im Prototyp fest codiert. Mit der Entwicklung eines User Interface für die Definition der Facetten kann die Anwendung für Nutzer zugänglicher gemacht werden. Wenn die Facetten im User Interface definiert werden können, muss auch eine Möglichkeit geboten werden, sie zu persistieren. Das aktuelle Format der Facettenbeschreibung kann beibehalten werden. Die Implementierung muss ergänzt werden, so

dass die JavaScript-Objekte mit der Beschreibung in JSON übersetzt und an den Server geschickt werden können, wo sie in der Datenbank abgelegt werden.

14.1.4 Tabellenfacette

Für die Tabellenfacette müssen zwei Erweiterungen vorgenommen werden, damit sie aus Anwendersicht nützlich ist. Es soll möglich sein, auf der Tabelle mehrere Einträge durch Halten der Shift-Taste gleichzeitig zu selektieren. Zur optimalen Nutzung dieses Features soll es ausserdem möglich sein, die Tabelle nach Spalten zu sortieren. Die Sortierfunktion muss sowohl im clientseitigen Service und auf dem Server noch implementiert werden.

14.2 Weitere Facettenimplementierungen

Es gibt weitere Facetten, die umgesetzt und in das System integriert werden können. Dazu gehören sowohl alternative Visualisierungen wie auch zusätzliche Arten der Selektion. Im Folgenden werden Facettentypen beschrieben, die im Verlauf des Projekts zur Sprache kamen. Die Implementierungs-idee geht vom aktuellen Stand des Prototyps aus und beschreibt ein Vorgehen, das möglichst kleine Anpassungen am Backend voraussetzt. Es ist davon auszugehen, dass eine alternative serverseitige Implementierung deutlich bessere Performance bieten kann. Die Nützlichkeit der Facetten muss in Zusammenarbeit mit der Zielgruppe vor der Implementierung noch verifiziert werden.

14.2.1 Kartenfacette

Eine mögliche Erweiterung ist die Implementierung einer Facette, die eine kartografische Darstellung anbietet. Eine Voraussetzung dafür ist, dass im Datensatz Standortdaten wie Längen- und Breitengrade gegeben sind.

Idee

Die Implementierung kann eine modifizierte Version der existierenden `RegularFacet` für Zahlenbereiche verwenden. Eine `RegularFacet` wird für die Längengrade und eine andere für die Breitengrade eingesetzt. Die nötige Modifikation ist, dass die Facette erlaubt, den Zahlenbereich für die Serverabfrage dynamisch anzupassen.

Schwierigkeiten

Die Performance leidet, wenn die vorhandene Implementierung der klassischen Facetten verwendet wird. Weil Längen- und Breitengrade separat betrachtet werden, müssen für eine Aktualisierung zwei Serveranfragen gemacht werden. Ausserdem muss serverseitig neue Funktionalität implementiert werden, wenn die Koordinaten in einer einzelnen Spalte im Datensatz vorhanden sind. Dazu könnte man beim ersten Laden die Spalte auf der Datenbank aufteilen und dann mit den beiden resultierenden Spalten arbeiten.

Die Visualisierung stellt ein weiteres Problem dar. Es muss dynamisch, abhängig vom Zoomgrad, eine Gruppierung vorgenommen werden, weil die Grösse der Datensätze nicht zulässt, dass alle Einträge im clientseitigen Speicher gehalten und auf der Karte abgebildet werden können. Alternativ kann die Karte auch nur als Werkzeug zur Selektion ohne Visualisierung des Datensatzes verwendet werden.

14.2.2 Polydimensionale Facetten über mehrere Spalten

Im Rahmen dieses Projekts wurde die Annahme getroffen, dass jedes Subjekt genau ein Attribut pro Dimension hat. Deshalb wurden polydimensionale Facetten, wie sie im Abschnitt 2.1 beschrieben sind, nicht in den Umfang aufgenommen.

Idee

Clientseitig kann ein neuer Facettentyp implementiert werden, der für jede Spalte eine `RegularFacet` enthält. Um die Facettenwerte der polydimensionalen Facette zu erhalten, werden die Werte der einzelnen Facetten vereinigt und die Duplikate entfernt. Die Anzahl Einträge pro Facettenwert werden als Summe aus den Ergebnissen der einzelnen Spalten berechnet. Die Facetten werden durch eine Disjunktion verknüpft und eine Selektion wird auf allen Facetten gleichermassen vorgenommen.

Schwierigkeiten

Wie auch bei der Kartenfacette führt diese Implementierungsidee dazu, dass für jede Aktualisierung mehrere Serverabfragen benötigt werden.

14.2.3 Diagramme

Alternative Visualisierungen der regulären Facetten können mit Diagrammen und Graphen umgesetzt werden. Diese können als zusätzliches Modul oder im Modul der regulären Facetten in die Anwendung eingebaut werden. Es gibt Libraries, mit denen die Darstellung effizient umgesetzt werden kann. Beispiele dafür sind Highcharts¹, D3.js² und dc.js³.

14.3 Performance

Die Performanceziele für den Prototyp wurden erreicht. Die nachfolgenden Ausführungen beschreiben Möglichkeiten, wie die Performance noch zusätzlich verbessert werden kann.

14.3.1 Caching

Im Prototyp werden lediglich die Ausgangszustände der regulären Facetten gecacht. Im Unterabschnitt 4.1.2 wurde beschrieben, welche Arten von Caching ergänzend implementiert werden können. Eine grosse Performanceverbesserung entsteht insbesondere dann, wenn ein Nutzer Facettenwerte selektiert und direkt wieder deselektiert, um herauszufinden, welchen Einfluss dieser einzelne Facettenwert hat. Nachdem die Auswahl rückgängig gemacht wurde, sind die Daten lokal verfügbar und es wird keine zweite Serverabfrage nötig. Der explorative Ansatz basiert genau darauf, eine Auswahl zu treffen und abhängig vom Resultat zu entscheiden, ob eine weitere Eingrenzung vorgenommen oder die Auswahl rückgängig gemacht werden soll. Deshalb ist Caching für diesen Anwendungsfall besonders geeignet und kann die Benutzererfahrung deutlich verbessern.

¹<https://www.highcharts.com>

²<https://d3js.org>

³<https://dc-js.github.io/dc.js/>

14.3.2 Backend

Die serverseitige Implementierung des Prototyps war nicht im Fokus der Arbeit und bietet noch Potenzial für eine Verbesserung der Performance. Die Architektur der Serverimplementierung erlaubt es, andere Datenbankzugriffe einzubinden. Die Typen von Datenbanken, die für die Anwendung in Frage kommen, wurden in Abschnitt 3.3 beschrieben. Wie dort erwähnt, müssen mit alternativen Implementierungen wie Elasticsearch Performance-Tests durchgeführt werden, um den idealen Typ zu finden.

Teil IV
Appendix

Ehrlichkeitserklärung

Hiermit erklären wir, die vorliegende Projektarbeit „Facettierte Suche als Tool für die Explorative Datenanalyse im Web“ selbständig, ohne Hilfe Dritter und einzig unter Benutzung der angegebenen Quellen verfasst zu haben.

Ort und Datum: Lenzburg 28.8.2020

Karin Duss

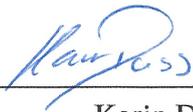
A. Berchtold

Andreas Berchtold

Ehrlichkeitserklärung

Hiermit erklären wir, die vorliegende Projektarbeit „Facettierte Suche als Tool für Explorative Datenanalyse im Web“ selbständig, ohne Hilfe Dritter und einzig unter Benutzung der angegebenen Quellen verfasst zu haben.

Ort und Datum: Erlinobach 28.08.20



Karin Duss

Andreas Berchtold

Literaturverzeichnis



- [1] B. Zheng, W. Zhang und X. F. B. Feng, „A survey of faceted search,“ *Journal of Web engineering*, Jg. 12, Nr. 1&2, S. 041–064, 2013.
- [2] G. Marchionini, „Exploratory Search: From Finding to Understanding,“ *Commun. ACM*, Jg. 49, Nr. 4, S. 41–46, Apr. 2006, ISSN: 0001-0782. DOI: [10.1145/1121949.1121979](https://doi.org/10.1145/1121949.1121979). Adresse: <https://doi.org/10.1145/1121949.1121979>.
- [3] (2020). „Client-side storage,“ Adresse: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage (besucht am 10. 08. 2020).
- [4] (2019). „Basic concepts,“ Adresse: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB (besucht am 10. 08. 2020).
- [5] (2019). „Das relationale Datenbankmodell,“ Adresse: <https://www.ionos.de/digitalguide/hosting/hosting-technik/relationale-datenbanken/> (besucht am 10. 08. 2020).
- [6] (2019). „NoSQL – Funktion und Vorteile von NoSQL-Datenbanken,“ Adresse: <https://www.ionos.de/digitalguide/hosting/hosting-technik/nosql/> (besucht am 10. 08. 2020).
- [7] (2019). „Dokumentenorientierte Datenbank: Wie funktioniert der Document Store?“ Adresse: <https://www.ionos.de/digitalguide/hosting/hosting-technik/dokumentenorientierte-datenbank/> (besucht am 12. 08. 2020).
- [8] (2019). „Graph Database (Graphdatenbank) erklärt,“ Adresse: <https://www.ionos.de/digitalguide/hosting/hosting-technik/graphdatenbank/> (besucht am 12. 08. 2020).
- [9] (2020). „Key-Value-Store: Wie funktionieren Schlüssel-Werte-Datenbanken?“ Adresse: <https://www.ionos.de/digitalguide/hosting/hosting-technik/key-value-store/> (besucht am 12. 08. 2020).
- [10] (2020). „Spaltenorientierte Datenbanken,“ Adresse: <https://datenbanken-verstehen.de/lexikon/spaltenorientierte-datenbanken/> (besucht am 10. 08. 2020).
- [11] (2020). „Spaltenorientierte Datenbank: Erklärung des Systems,“ Adresse: <https://www.ionos.de/digitalguide/hosting/hosting-technik/spaltenorientierte-datenbank/> (besucht am 12. 08. 2020).
- [12] (2020). „Elasticsearch Resiliency Status,“ Adresse: https://www.elastic.co/guide/en/elasticsearch/resiliency/current/index.html#_completed (besucht am 14. 08. 2020).

LITERATURVERZEICHNIS

- [13] D. Gilling. (2020). „REST API Design: Filtering, Sorting, and Pagnation,“ Adresse: <https://www.moesif.com/blog/technical/api-design/REST-API-Design-Filtering-Sorting-and-Pagination/> (besucht am 16.08.2020).
- [14] M. Keep und H. Ingo. (2020). „Performance Best Practices: Indexing,“ Adresse: <https://www.mongodb.com/blog/post/performance-best-practices-indexing> (besucht am 06.08.2020).
- [15] (1993). „Response Times: The 3 Important Limits,“ Adresse: <https://www.nngroup.com/articles/response-times-3-important-limits/> (besucht am 15.08.2020).
- [16] (2020). „ECMAScript 6 - ECMAScript 2015,“ Adresse: https://www.w3schools.com/Js/js_es6.asp (besucht am 15.08.2020).
- [17] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 3. Aufl. Dez. 1999, S. 4. Adresse: <http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf>.
- [18] (2020). „Object.prototype.constructor,“ Adresse: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor (besucht am 27.08.2020).
- [19] (2020). „Modulbeschreibung - Web Clients,“ Adresse: <https://www.fhnw.ch/de/studium/module/9309249> (besucht am 18.08.2020).
- [20] (2019). „ECMAScript 2019 Language Specification,“ Adresse: <http://www.ecma-international.org/ecma-262/10.0/index.html> (besucht am 19.08.2020).

Abbildungsverzeichnis

2.1	Beispiel für eine flache Facette	10
2.2	Beispiel für eine flache polydimensionale Facette	11
2.3	Beispiel für eine hierarchisch gruppierte Facette	11
2.4	Beispiel für eine hierarchisch Bereichs-Facette	12
2.5	Beispiel UI für Facettensuche im Datenanalysebereich	13
4.1	Beispiel Client-Webserver-Architektur	19
6.1	Statische Variante mit Auswahl- und Anzeigebereich (links) und Modulsystem (rechts)	36
6.2	Interner Ablauf bei Auswahl im User Interface	37
6.3	Vor jeder Aktualisierung wird die Auswahl zurückgesetzt	38
7.1	Aufbau der Client-Server-Kommunikation	40
7.2	Struktur von MVC inklusive Benutzer	41
8.1	Screenshot des Prototyps	43
8.2	Darstellung als einfache Liste (links), als Gruppierung von Strings (mitte) und Zusammenfassung der Werte nach Zahlenbereichen (rechts)	47
8.3	Beispiel eines asynchronen Serverrequests	47
9.1	Serveraufbau	49
9.2	Übersicht der zwei zur Verfügung gestellten Interfaces	50

Tabellenverzeichnis

3.1	Beispieldatensatz für eine spaltenorientierte Datenbank	18
4.1	Beispieldatensatz für die Keyset Pagination	21
6.1	Übersicht einiger der bekanntesten Datenbanken	34
9.1	Resultate der Zeitmessungen	52

Glossar

Attribut Ein einzelner Facettenwert der einem oder mehreren Subjekten zugeordnet werden kann. 10, 73

CRUD Create Read Update Delete. 15

Dimension Ein Aspekt des Subjekts, der durch die Facette beschrieben wird. 10, 46

Facettenwert Attribut oder Kategorie. 10, 73

JSON JavaScript Object Notation. 16, 35

Kategorie Eine Sammlung von Attributen. 10, 73

klassische Facette Facetten die standardmässig dargestellt werden. 36

Noise Relevante Datensätze sind nicht sichtbar, weil zu viele Ergebnisse angezeigt werden. 13

regulären Facetten Synonym für klassische Facette. 7

Silence Das Ausbleiben von Resultaten bei einer Suchanfrage. 13

Subjekt Ein einzelnes Dokument, das anhand der Facetten klassifiziert wird. Im Kontext von relationalen Datenbanken ist ein Subjekt eine Zeile. 10

20FS_IMVS20: Facettierte Suche als Tool für Explorative Datenanalyse im Web

Betreuer:	Dierk König Dieter Holz	Priorität 1	Priorität 2
		Arbeitsumfang:	P5 (180h pro Student)
		Teamgrösse:	2er Team
Sprachen:	Deutsch oder Englisch		P5 oder P6
			2er Team

Ausgangslage

Explorative Datenanalyse ist ein Kernbestandteil aller Aktivitäten im Bereich Data Science. Strukturierte Datenbestände von genügender Grösse lassen sich nicht mehr durch einfaches "Draufschauen" untersuchen, sondern brauchen Werkzeugunterstützung um interessante - oft unerwartete - Zusammenhänge in einer grossen Menge von Datensätzen zu finden. Zu diesem Zweck werden die Daten nach Dimensionen gegliedert und die Dimensionen ontologisch unterteilt (häufig in eine Baumstruktur). Daraus entstehen Facetten, die kombiniert werden können, um den Datenbestand zu filtern.

Noch gibt es keine offen verfügbaren Werkzeuge für diese Art der Untersuchung.

Ziel der Arbeit

Ziel der Arbeit ist, ein web-basiertes Werkzeug zu konzipieren und zu bauen, dass

- vorstrukturierte Datenbestände einlesen kann,
- vorgegebenen Ontologien einlesen kann und
- eine hochinteraktive Benutzeroberfläche bietet um die Daten entlang dieser Facetten darzustellen.

Problemstellung

Die Studierenden müssen

- den Ansatz der facettierten Suche verstehen,
- ein passendes Format für die einzulesenden Daten definieren,
- ein effizientes Datenmodell mit dazugehörigem API finden,
- eine hochdynamische Benutzeroberfläche bauen, die im Browser dargestellt werden kann.

Technologien/Fachliche Schwerpunkte/Referenzen

Web-Technologien, wie in den Modulen WebProgramming und Web-Clients unterrichtet.