Der Global Search des Metadata-Hub

MASTER THESIS

Robin Kreuzer

Eingereicht am 15. Dezember 2021

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Betreuer:
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 15. Dezember 2021

License

Erlangen, 15. Dezember 2021

Abstract

The Metadata Hub is a software that can scan millions of files, store their metadata, and make it available to the user or third-party applications via intelligent search. This software runs on different servers. Thus, the data can only be queried in each instance.

Therefore the work shows how the functions of the Metadata Hub, in particular the search, can be made available orchestrated in a so-called Global Search, so that this Global Search provides the same interface as a normal Metadata-Hub instance, so that e.g. third party software does not have to be adapted separately for it. The communication between the normal instances and the Global Search instance is based here exactly like the communication from an external client on GraphQL. The work also takes into account what the Metadata Hub is all about: simplicity in use.

Zusammenfassung

Der Metadata-Hub ist eine Software, die Millionen von Dateien scannen, deren Metadaten abspeichern und diese dem Benutzer oder Drittsoftware über eine intelligente Suche zur Verfügung stellen kann. Diese Software läuft auf verschiedenen Servern. Somit können die Daten nur jeweils Instanz bezogen abgefragt werden.

Deshalb zeigt die Arbeit wie die Funktionen des Metadata-Hub, davon insbesondere die Suche in einem sogenannten Global Search orchestriert zur Verfügung gestellt werden können, sodass dieser Global Search nach außen so aussieht, wie wenn er eine ganz normale Metadata-Hub Instanz wäre, somit bspw. Drittsoftware dafür nicht separat angepasst werden muss. Die Kommunikation zwischen den normalen Instanzen und der Global-Search-Instanz basiert hierbei genau wie die Kommunikation von einem außen stehenden Clienten auf GraphQL. Die Arbeit beachtet zudem das, was den Metadata-Hub ausmacht: Einfachheit in der Benutzung

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Zielsetzung

Ein Metadata-Hub kann Metadaten von mehreren Millionen Daten aufnehmen und dem Benutzer über eine Suche zugänglich machen. Was ist aber, wenn ein Metadata-Hub mehrere Milliarden von Daten aufnehmen können und diese dann über die Suche zugänglich machen soll? Oder wie kann ein Benutzer möglichst einfach auf alle Metadaten zugreifen, die möglicherweise sogar von verschiedenen (Sub-)Unternehmen gesammelt worden sind.

Für den Fall, dass mehrere Milliarden Daten aufgenommen werden sollen, bieten sich zwei Lösungen an, wie in Abb. 1.1 zu sehen ist. Zum einen kann ich die Seite des Einsammelns (Harvesten) verstärken, was nicht Zielsetzung dieser Arbeit ist, zum anderen kann ich die einzusammelnden Daten auf verschiedene Metadata-Hub (MdH) Instanzen aufteilen und diese dann zentral über einen Global Search abfragen.

Dieser Ansatz (grün-blaue Seite) hat wie die Abbildung zeigt gegenüber der roten Seite, auf der ausschließlich das Einsammeln verstärkt wird, folgende Vorteile: Ich kann die einzelnen Metadata-Hub Instanzen auf verschiedenen Servern installieren, jede Instanz mit ihrer eigenen Datenbank. Somit kann ich nun beim Harvesten als auch bei der Suche auf mehr Rechenleistung zurückgreifen. Dies wäre für die rote Seite, zwar auch möglich, beschränkt sich hierbei dann aber nur auf das Harvesten. Die Daten beim Harvesten schließlich in die Datenbank einzufügen, wäre hier genau wie die Suche auf eine Instanz und somit auf die Rechenleistung eines Server beschränkt.

Deshalb ist das Ziel der Arbeit, den Global Search auf der grünen Seite zu implementieren und Teile der bisherigen Implementierung (blau) so anzupassen, dass der Global Search den späteren Anforderungen entsprechend mit den Client Instanzen auf die Weise kommunizieren kann, dass er selbst nach außen wie ein einzelner großer Metadata-Hub wirkt, der Metadaten von Milliarden von Daten enthalten kann.

Der Ansatz der grünen Seite deckt auch den Fall ab, Metadata-Hub's verschiede-

ner Organisationen zu durchsuchen, wobei die Daten hier schon von vornherein auf verschiedene Metadata-Hub Instanzen aufgeteilt worden sind.

Im Grundlagenkapitel Abschnitt 2.1 gehe ich näher auf den bisherigen existierenden Metadata-Hub ein und erkläre dort unter anderem dessen genaue Funktionsweise.

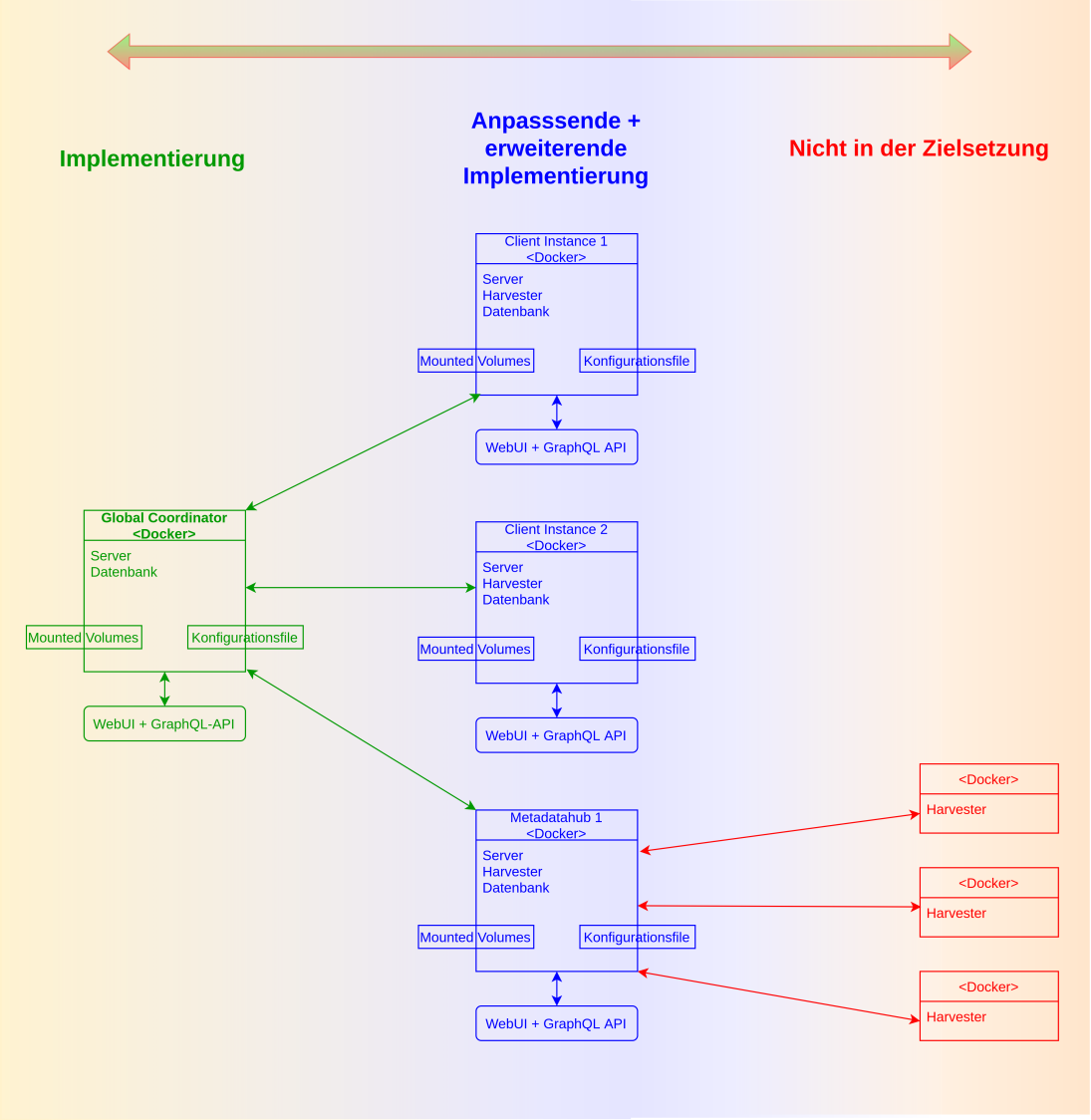

Abbildung 1.1: Zielsetzung

1.2 Aufbau der Arbeit

In diesem Kapitel gebe ich einen kurzen Gesamtüberblick, wie diese Arbeit aufgebaut ist.

Neben der diesem Kapitel vorangegangenen Zielsetzung gehe ich im folgenden Unterkapitel noch auf verschiedene Begrifflichkeiten und Abkürzungen ein, die für diese Arbeit wichtig sind.

Im darauf folgenden Grundlagenkapitel erkläre ich schließlich etwas genauer was der Metadata-Hub ist, was ein Benutzer damit machen kann und wie seine bisherige Entstehungsgeschichte ist, also von der Idee bis zur aktuellen Umsetzung.

Danach beschreibe ich den Metadata-Hub aus einer technologischen Sichtweise: Die Architektur des Metadata-Hubs. Also welche Programmiersprachen zum Einsatz kommen, aus welchen Komponenten die Software besteht, wie diese zusammenarbeiten und in Docker gekapselt sind. Außerdem gehe ich dort dann auch etwas genauer auf den Server ein, da dieser für die Umsetzung des Global Search die entscheidende Rolle spielt.

Die Graph Query Language (GraphQL) spielt hierbei und für die Kommunikation eine wichtige Rolle, deshalb erkläre ich GraphQL etwas genauer und welche Möglichkeiten überdies das darauf aufbauende und im Metadata-Hub eingesetzte Java Framework GraphQL bietet, zudem diese Möglichkeiten für die Umsetzung des Global Search relevant sind. Außerdem sind das Fabrik und Dependency Injection Entwurfsmuster sowie für die parallele Ausführung die Java CompletionStage API wichtig, die das Grundlagenkapitel abschließen.

Im dritten Kapitel folgen dann die Anforderungen, die der Global Search funktional und nicht-funktional erfüllen muss. Die funktionalen Anforderungen beziehen sich dann auf konkrete Features, die der Global Search bereitstellen können soll, wie z.B., dass der Benutzer in der WebUI des Global Search die Möglichkeit haben soll, konkrete Client Instanzen auswählen zu können.Die nicht-funktionalen Anforderungen beziehen sich dann andererseits auf Kriterien, die funktionsübergreifend für alle funktionalen Anforderungen gelten sollen.

Die beiden darauf folgenden Kapitel beschreiben nun, die Architektur und Umsetzung des Global Search und die daran gestellten Anforderungen.

Dabei ist das erste beider Kapitel mit der Koordinierungs-Schicht der Teil, der primär die nicht-funktionalen Anforderungen abdeckt. Diese Schicht ist dafür relevant, wie der Global Search mit den Client-Instanzen verbunden werden kann, wie die Kommunikation und Fehlerbehandlung funktioniert und dient somit als Framework für die der Schicht nachgelagerten Funktionen.

Deren Umsetzung beschreibe ich dann im darauf folgenden Kapitel, das die *Architektur und Implementierung der Global Search Funktionen* enthält. Darin gehe ich dann auf die vier zentralen Funktionen ein, die so auch schon im bisherigen Metadata-Hub existieren, jedoch im Kontext des Global Search jetzt die Aufgabe haben, die Teilergebnisse der einzelnen Client-Instanzen zu einem Ausgabeergebnis zusammenzufügen. Hierbei erkläre ich anfangs jeweils kurz, wofür die entspre-

chende Funktion relevant ist und welche Besonderheiten sie mit sich bringt. Nach deren jeweiligen Teilkapiteln, welche die Umsetzung beschreiben, werte ich die zu diesen Funktionen zugehörigen Anforderungen direkt im Anschluss jeweils aus.

Alle weiteren funktionalen und nicht-funktionalen Anforderungen werte ich schließlich im sechsten Kapitel aus. Dabei evaluiere ich, ob die Anforderungen erfüllt, teilweise erfüllt oder gar nicht erfüllt werden konnten.

In Kapitel 7 fasse ich die Ergebnisse der Arbeit noch einmal kurz zusammen und gebe einen Ausblick, wie der Global Search weiterentwickelt werden könnte.

1.3 Begriffsdefinitionen

Um den Metadata-Hub und den Global Coordinator besser erklären zu können, greife ich auf verschiedene Begriffe, die ich im folgenden kurz erwähnen möchte:

Metadata-Hub (MdH) beschreibt die Software in ihrer Gesamtheit.

Global Search (GLS) ist der Teil des Metadata-Hub, der bisherige einzelne Client-Instanzen koordiniert beziehungsweise deren jeweiligen Ausgaben in eine Gesamtausgabe überführt.

Instanz: Der MdH ist aus einer theoretischen Perspektive nur der Bauplan der Software. Eine Instanz davon ist dann das aktive Programm. Und von einem MdH kann es mehrere (MdH) Instanzen geben, die auf dem gleichen Computer, als auch auf verschiedenen laufen können. Ich weise an dieser Stelle aber darauf hin, dass ich aus praktischen Gründen auch öfters einfach nur MdH statt MdH Instanz schreibe.

Normale Instanze/Client-Instanz meint eine Instanz der bisherige Software des MdH, mit der Metadaten eingesammelt und einem Benutzer verfügbar durchsuchbar gemacht werden können. Client-Instanz insbesondere soll verdeutlichen, dass diese Instanz einem GLS untergeordnet ist.

Global-Search-Instanz: Wie bei Client-Instanz schon Beschrieben, soll das die Ausführung der aktuellen Software verdeutlichen, in dem Fall der GLS Software. Auch hier sei darauf hingewiesen, dass ich aus praktischen Gründen den Zusatz Instanz nicht immer schreiben werde.

Ausgewählte Client-Instanzen sind die Instanzen, die für die aktuelle Anfrage relevant sind. Ein GLS kann mehrere Client-Instanzen haben, aber muss nicht immer alle davon für eine aktuelle Anfrage verwenden und das soll *ausgewählte Client-Instanzen* verdeutlichen

Client hat in der Informatik viele Bedeutungen. Im Kontext dieser Arbeit ver-

wende ich diesen Begriff hauptsächlich, wenn Dienst X eine Funktion bereitstellt, die von Dienst Y konsumiert werden kann. Hierbei agiert der Dienst Y in der Rolle eines Clients. Das gilt auch dann, wenn ein menschlicher Benutzer eine Funktion von Dienst X in Anspruch nimmt. Möchte ich jedoch die Rolle eines menschlichen Clients explizit hervorheben, verwende ich in diesem Fall den Begriff Benutzer.

Ähnliches gilt wenn der GLS und Client-Instanzen miteinander kommunizieren; Doch differenziere ich es an dieser Stelle explizit, indem ich dann von einer Client-Instanz bzw. Global-Search(-Instanz) schreibe.

Benutzer: Diesen Begriff verwende ich vorrangig, wenn ich einen menschlichen Client hervorheben möchte. Aber ich verwende diesen Begriff auch, wenn ich zwischen einem externen Aufrufer und einem internen Aufrufer differenzieren möchte. Das heißt auch Drittsoftware, die die MdH API verwendet umschreibe ich stellvertretend mit Benutzer, wenn es aus der Erklärungsperspektive irrelevant ist, ob es sich hierbei um einen manuellen oder automatisierten Aufruf handelt.

API ist eine Abkürzung für Programmierschnittstelle. Diese kann von anderen bspw. Drittprogrammen verwendet werden, um verschiedene Aktionen in dem Programm, das diese Schnittstelle bereitstellt, auszulösen oder Daten von diesem abzufragen. Der Metadata-Hub bietet eine API basierend auf GraphQL an.

Anfrage: Eine Anfrage kann sowohl von einem Benutzer an den GLS oder an eine normale Client-Instanz gesendet werden, aber auch von einem GLS an die (ausgewählten) Client-Instanzen. Aus den jeweiligen Kontexten geht dann hervor, welches von beidem gemeint ist.

MDH-CTL: Kommandozeilen-Wergzeug, um den MdH einzurichten und auf diesen, aber auch auf andere über die Kommandozeile zuzugreifen. Außerdem stellt dieses Tool eine Python-Schnittstelle zur Verfügung, um darüber auf den MdH zugreifen zu können

Metadata-Tags sind von den Schlüssel-Wert-Paaren der Metadaten die Schlüssel. Der Metadata-Tag von `fileName=xyz.png` ist `FileName`

Harvesting: Einsammeln und Auswerten von Metadaten.

DataFetcher ist im Kontext von GraphQL Java eine Klasse oder eine Lambda-Funktion, die mit einem GraphQL-Feld verbunden werden kann und dessen Ergebnis berechnet. Auf eine spezielle Formatierung verzichte ich bei DataFetcher.

2 Grundlagen

Dieses Kapitel geht in seinen ersten Unterkapiteln näher auf den bisherigen Metadata-Hub ein und beschreibt neben seiner Funktionsweise unter anderem auch dessen Architektur. Die letzten Unterkapitel erklären dann die Grundlagen von GraphQL sowie weitere Aspekte, die für die spätere Umsetzung des Global Search relevant sind.

2.1 Metadata-Hub

Dieses Kapitel ist in drei Unterkapitel aufgeteilt. Anfangs beschreibe ich wie der Metadata-Hub entstanden ist, schließlich dessen Funktionsweise und zuletzt gehe ich auf dessen Aufbau und dessen Architektur näher ein.

2.1.1 Enstehung

Wie ist es zu der Idee *Metadata-Hub* gekommen. Wie wurde das Projekt gestartet? Zwei Fragen, die dieses Kapitel beantwortet.

In Gesprächen mit Partnerfirmen entstand, bei der Firma GRAU Data GmbH, Mitte 2019 die Idee zum Metadata-Hub. (Haag, 2021)

Professor Dr. Dirk. Riehle bietet über seine Professur Firmen aus der Wirtschaft an, dass Studentengruppen in Form einer Universitätsveranstaltung mittels `Scrum` eine Softwareidee dieser Firma umsetzen. Diese Software steht schließlich aber jedem zur Verfügung, da sie als `Open Source` Software auf GitHub veröffentlicht werden muss. So hat sich dann auch die Firma GRAU DATA um die Entwicklung eines Softwareprojekts bei Dirk Riehle für das Sommersemester 2020 beworben und zwar mit der Idee des Metadata-Hub. Dieses wurde dann von einer Studentengruppe, zu der auch ich gehört habe, ein Semester lang entwickelt. Schließlich sind dann ich und ein Mitkommilitone, der ebenfalls an diesem Softwareprojekt mitgearbeitet hat, als Werkstudenten von der Firma GRAU DATA übernommen worden, diese Software nun exklusiv für diese Firma weiterzuentwickeln. Hinzugekommen sind seitdem zahlreiche weitere Funktionen und Verbesserungen. Wie die

aktuelle Software mittlerweile aussieht bzw. funktioniert erkläre ich im folgenden Kapitel. Neben der Max-Planck-Gesellschaft interessieren sich mittlerweile einige weitere Firmen sowie das ZUSE-Institut aus Berlin für diese Software; also Firmen mit sehr vielen Daten wobei eine einzelne Metadata-Hub Instanz hierbei nicht viel leisten kann. Umso wichtiger ist so ein Global Search, mit dem es möglich sein wird, diese Menge an Metadaten zu verarbeiten.

2.1.2 Funktionsweise

Damit der Metadata-Hub so funktioniert wie vom User gewollt, muss er diesen erst einmal richtig installieren bzw. konfigurieren. Das MdH Kommandozeilen Interface MdH-CTL ermöglicht dem Systemadministrator derartige Konfiguration. Hierbei kann er angeben, welche Verzeichnisse dem MdH zur Verfügung stehen. Dabei kann er lokale Verzeichnisse auswählen, also solche die sich auf dem Server befinden, auf dem dann auch der MdH laufen soll. Außerdem kann er hier auch über die Ferne eingebundene Verzeichnisse wie z.B NFS-Shares bestimmen. Weiterhin muss der Systemadministrator einen Port festlegen, worüber dann auf die Weboberfläche und auf die GraphQL-API zugegriffen werden kann. Abschließend muss er noch ein Passwort für einen Administrator als auch für einen normalen User festlegen, über das sich der Benutzer später einloggen kann. Ein Administrator hat auf alle Funktionen Zugriff, der normale User kann hauptsächlich nur auf die Suchfunktionalitäten zugreifen. Auf die Harvest-Funktionalität und die sensiblen Zurücksetzungsoperationen hat dieser keinen Zugriff. Nachdem der Systemadministrator den MdH über das MdH-CTL eingerichtet und die Docker-Instanz ebenfalls mit dem Tool gestartet hat, kann der Benutzer jetzt auf den Metadata-Hub zugreifen.

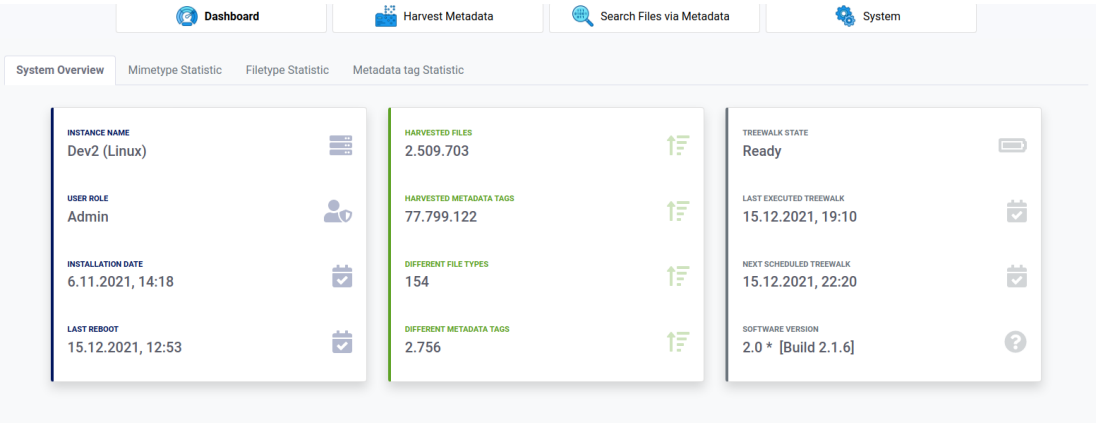

Abbildung 2.1: Systemübersicht

Sobald er sich eingeloggt hat, zeigt die webbasierte Benutzeroberfläche eine Systemübersicht an, wie in Abb. 2.1 gezeigt. In dieser findet er ausgewählte wichtige In-

formationen wie z.B. wie viele Dateien geharvestet sind oder wie viele Metadata-Tags insgesamt in dieser Instanz abgespeichert sind. Um eine etwas genauere Übersicht darüber zu bekommen, was für Daten in dieser Instanz geharvestet vorliegen, bietet sich ein Blick in die weiteren Dashboardseiten an. Als Beispiel habe ich hierfür noch Abb. 2.2 eingefügt. Darin ist eine Verteilungsstatistik zu den Metadaten-Tags zu sehen. Da in dieser Instanz hauptsächlich Bilddateien geharvestet sind, verwundert es nicht, dass Metadaten-Tags, die hauptsächlich bei Bilddateien zu finden sind, diese Statistik dominieren.

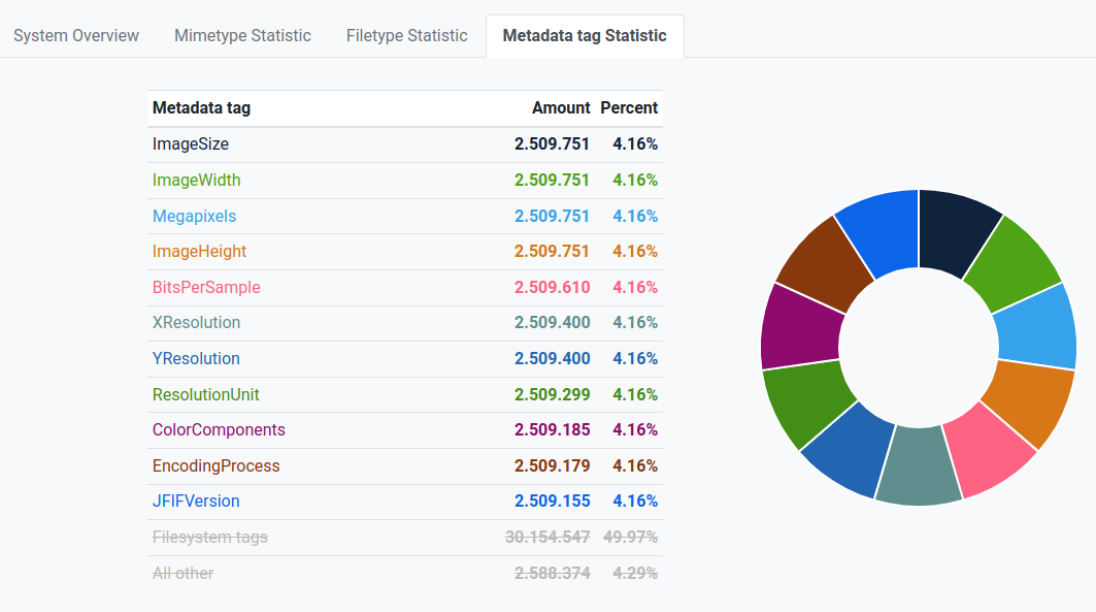

Abbildung 2.2: Metadaten-Tag Statistik

In der `Harvest`-Kategorie hat ein als Administrator eingeloggter Benutzer die Möglichkeit, das Einsammeln der Metadaten-Tags direkt zu starten oder auch (regelmäßig) zu planen, sodass der Metadata-Hub beispielsweise jeden Sonntag das Dateisystem erneut durchsucht und Änderungen an Dateien oder Metadaten-Tags neuer Dateien abspeichert. Dabei kann er außerdem spezifizieren, wie stark die CPU beansprucht werden darf, bis er den Harvest-Vorgang für die Zeit, solange die CPU über der Schwelle ist, pausiert. In der Kategorie `Additional Extractors` hat der Administrator die Möglichkeit anzugeben mit welchen weiteren Extraktoren Metadaten von den zu harvestenden Dateien berechnet/heraus-gelesen werden sollen. So kann er sich bspw. zu jeder Datei auch einen `Fingerprint` (SHA256 Prüfsumme der Datei) berechnen und abspeichern lassen. Die unter `Directories` zur Verfügung stehenden Einträge sind die Verzeichnise, die der Systemadministrator dem MdH zu Beginn beim Setup zur Verfügung gestellt hat. Abb. 2.2 zeigt wie die dazugehörige Benutzerschnittstelle aussieht.

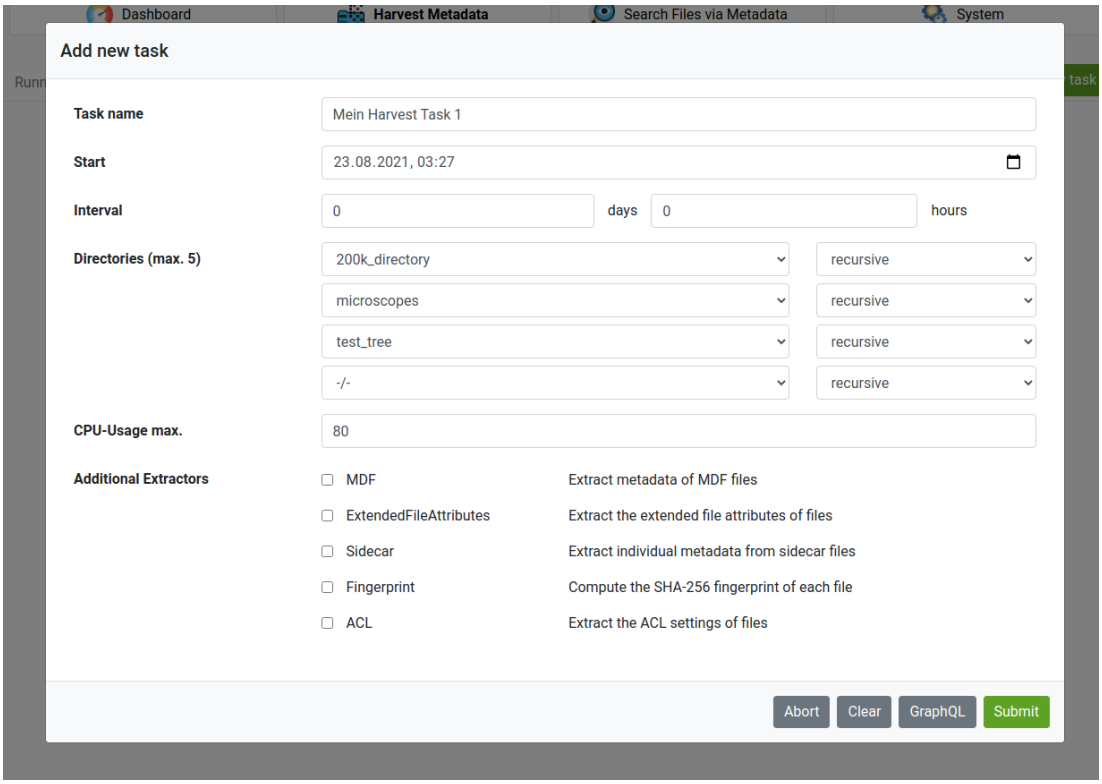

Abbildung 2.3: Hinzufügen einer neuen Harvest-Aufgabe

Sobald die Daten geharvestet sind, steht dem User mit der Suche ein umfangreiches Werkzeug zur Verfügung, nach den Metadaten gezielt, aber auch mit komplexen booleschen Ausdrücken, suchen zu können. Dabei steht ihm neben der direkten Metadaten-Tag Autovervollständigung bei Eingaben in die Suche auch ein Metadaten-Tag-Explorer zur Verfügung. Über diesen hat er die Möglichkeit, sich alle Metadaten gruppiert nach Dateitypen anzeigen zu lassen, gezielt nach ihnen zu suchen und diese dann auch der Suchmaske hinzuzufügen. Außerdem kann er darüber einstellen, welche Metadaten ihm in der Ausgabetabelle direkt angezeigt werden sollen.

Der Harvester bestimmt beim Einsammeln der Metadaten, welchen Datentyp diese jeweils haben. Hierfür sind dem MdH drei Datentypen bekannt: Text, Zahlen, Zeitstempel. Die Suchmaske kann damit das Eingabefeld für den Wert automatisch für eine zielgerichtete auf den zur Grunde liegenden Datentypen optimale Eingabe anpassen. So kann der Benutzer beispielsweise bei Metadaten, die vom Typ Zeitstempel sind, die Suche direkt mit einem `Date-Time-Picker` befüllen. Sollte es zu einem Metadaten-Tag mehrere Datentypen geben, dann wird der Datentyp genommen, der bei diesem Metadaten-Tag am meisten vorkommt. Der Benutzer hat hierbei aber die Möglichkeit diese automatische Vorgabe zu überspielen.

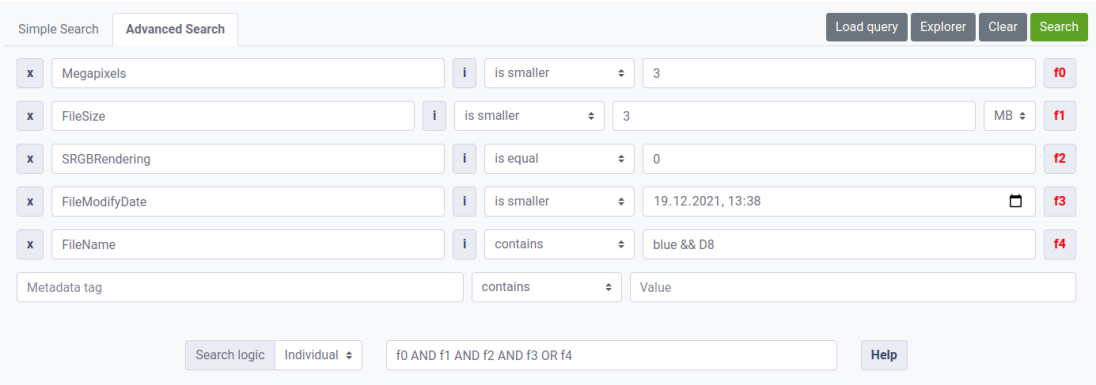

Abbildung 2.4: Sucheingabemaske

Mit dem `i` in Abb. 2.4 erhält der Benutzer zusätzliche Informationen zu dem verwendeten Metadaten-Tag, bspw. wie oft er insgesamt vorkommt. Auch sieht er hier drei Beispiele und er kann dort das eben erwähnte Überspielen des Datentyps einstellen. An gleiche Informationen gelangt der Benutzer auch über den `Explorer`. Die Abbildung zeigt eine anspruchsvolle Suche, bei der die einzelnen Suchreihen wie es standardmäßig der Fall ist, nicht einfach nur `AND` verknüpft sind, sondern so wie es der Benutzer in `Search logic` einstellen kann.

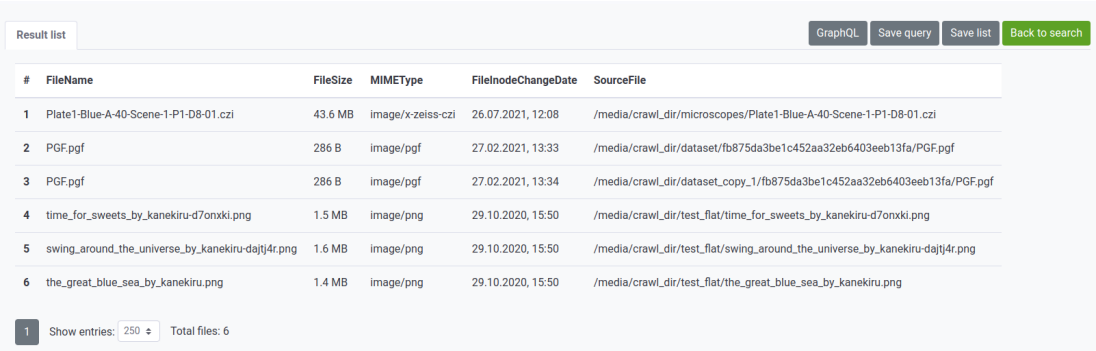

Abbildung 2.5: Suchergebnis

Abb. 2.5 zeigt das Ergebnis der Suche. In diesem Beispiel gibt es nur 6 Ergebnisreihen. Es können hier aber Millionen Ergebnisse zurückkommen. Über den `Paginator` kann der Benutzer dann durch diese Ergebnisreihen scrollen. Standardmäßig zeigt der MdH hier 5 Metadaten-Tags in den Spalten an. Weitere kann sich der Benutzer in der Suche über den `Explorer` hinzufügen. Mit einem Klick auf eine bestimmte Ergebnisreihe kann sich der Benutzer zudem alle Metadaten-Tags zu dieser Datei anzeigen lassen. Außerdem kann er darüber seiner Suche weitere Metadaten-Tags mit dem darin angezeigten `Tag` hinzufügen und damit seine Suche weiter verfeinern. Über `Save query` kann er sich den Inhalt der Suchmaske abspeichern lassen, um ihn später mit `Load query` wieder zu laden. `Save list`

dagegen bietet dem Benutzer verschiedene Exportier-Möglichkeiten an. So kann er sich die Ergebnisliste als `JSON`, `HTML` und in weiteren Formaten abspeichern.

Mit dem GraphQL-Button, der auch schon in Abb. 2.3 zu sehen war, bietet der MdH eine Low-Code Funktion. Damit kann sich ein Benutzer den zu dieser Anfrage entsprechenden GraphQL Code anzeigen lassen, den er anschließend abspeichern kann, oder den er an die in der WebUI vorhandenen interaktiven GraphiQL Konsole senden kann. Jegliche Interaktion zwischen der WebUI und dem Backend basiert auf GraphQL. Somit ist es dem Benutzer möglich, automatisiert auf alle WebUI Funktionen, aber auch weitere nicht direkt in der WebUI verfügbaren Funktionen direkt über GraphQL oder auch über die Kommandozeile, wie auch über die zur Verfügung gestellte Python-Pakete auf den MdH zuzugreifen.

Die eben schon erwähnte GraphiQL-Konsole und Funktionen, den MdH zu resetten oder auch die WebUI im Design anzupassen finden sich im `System`-Reiter und runden den MdH ab.

Mehr Informationen zu der webbasierten GraphiQL-Konsole finden sich hier: „GraphiQL“ (n. d.)

2.2 Graph Query Language (GraphQL)

Um die Architektur des Metadata-Hub besser verstehen zu können, führe ich in diesem Kapitel in die Grundlagen GraphQL ein.

GraphQL wurde 2012 von Facebook entwickelt, um damit die ganze Facebook-Api darstellen zu können. Drei Jahre später haben sie sich entschieden GraphQL als Open-Source unter https://graphql.org/ zu veröffentlichen, das dort stetig, auch mit der sich dann formierten Gemeinschaft weiterentwickelt wird. (Byron, 2015)

Im folgenden ersten Unterkapitel beschreibe ich allgemein, was GraphQL ist, wie es aufgebaut ist und sich von `REST` unterscheidet. Im zweiten Unterkapitel gehe ich dann näher darauf ein, wie `GraphQL Java` die Spezfifikation GraphQLs umgesetzt hat und welche weiteren Besonderheiten `GraphQL Java` aufweist.

2.2.1 GraphQL Allgemein

Anders als bei `REST` gibt es bei GraphQL nur eine Schnittstelle in den Server hinein, über die er dann die GraphQL-Anfrage (in Textform) und mögliche dazugehörige Variablen (in JSON-Form) annehmen kann. Letzteres ist aus der Sicht eines Benutzers optional. Der zentrale Gegenstand einer GraphQL-API ist das GraphQL-Schema. In diesem können Typen mit Feldern definiert werden. In den Wurzeltypen `Query` und `Mutation`, die später als Einsprungpunkte dienen, defi-

niere ich von mir sogenannte Funktionsfelder, also normale Felder mit Parametern. Was ich damit genau meine zeige ich im folgenden Beispielschema, das ich aus dem MdH entnommen habe. (Facebook, 2018; Foundation, 2021)

```
type Query{
   getMetadataTags(
      fileTypeScope: [String!],
      mimeTypeScope: [String!],
      nameFilter: String,
      excludeMetadata: [String!],
      limit: Long,
      offset: Long
   ): [Metadata!]!,
   get...
}

type Metadata {
   name: String!
   count: Long!
   type: String!
   searchDeactivated: Boolean!
   virtual: Boolean!
   randomExample(count:Long): [MetadataExample!]!
   detailTypeItemList: [DetailTypeItem!]!
}
```

In diesem Beispiel ist der Wurzeltyp `Query` zu erkennen. Dieser hat ein Feld `getMetadataTags`. Dass er noch weitere Felder haben könnte ist mit `get...` angedeutet. Das Feld `getMetadataTags` nimmt Parameter entgegen, die der GraphQL-Server dann nutzen kann, eine diesen entsprechende Ausgabe zu erzeugen. Dieses Feld gibt eine Liste von `Metadata` zurück, das selbst wieder ein Typ ist. Die Klammern bei `Metadata` nach dem Doppelpunkt von `getMetadataTags` bedeutet, dass eine Liste von dem darin enthaltenen Typ zurückgegeben werden soll. Das direkte Ausrufezeichen hat zur Folge, dass kein Listenelement `NULL` sein darf und das Ausrufezeichen hinter der Klammer steht dafür, dass kein `NULL` statt der Liste zurückgegeben werden darf. Typen wie `String`, `Long` oder `Boolean` werden in GraphQL auch Skalare genannt, da sie anders wie der Typ `Metadata`, sich nicht mehr weiter unterteilen lassen. Auf die Angabe vom Typ `DetailTypeItem` und `MetadataExample` habe ich hier verzichtet. Werden Typen bzw. Skalare in den Parametern verwendet, dann werden diese als Eingabe-Typen bezeichnet, die genauso aufgebaut sind wie normale Typen. Auch hier ist den Ausrufezeichen eine besondere Aufmerksamkeit geschenkt. Sind sie vorhanden, ist dieser Parameter nicht mehr optional. Dieses vorgestellte Schema ist dem Server zur

Übersetzungszeit bekannt.(Facebook, 2018; Foundation, 2021)

Zur Laufzeit bekommt er nun Anfragen, die zu diesem Schema passen müssen. Folgendes Beispiel soll das illustrieren.

Anfrage an den Server:

```
query{
  getMetadataTags(nameFilter:"Mega",
                  fileTypeScope:["png","jpg"])
  {
    name
    count
    type
  }
}
```

JSON-Ergebnis:

```
{
  "data": {
    "getMetadataTags": [
      {
        "name": "Megapixels",
        "count": 10,
        "type": "num"
      }
    ]
  }
}
```

Wie anhand des Beispiels gut zu sehen ist, habe ich hier sowohl bei den Parametern, als auch bei den Feldern von `Metadata` nicht alles belegt bzw. abgefragt. Gerade die Tatsache, dass ich bei GraphQL nicht alle Felder abfragen muss, ist ein Vorteil gegenüber von `REST`, wo ich immer alle Felder zurückbekomme, die der Server für einen bestimmten REST-Befehl zur Auslieferung notwendig hält. Auch, dass klar ist, welchen Datentyp sowohl ein Anfrageparameter als auch ein Feld in der Ausgabe haben muss, ist im Vergleich zu REST ein klarer Pluspunkt für GraphQL, das derartigen Meachnismus nativ unterstützt. Nun könnte ich mein GraphQL-Schema dahingehend designen, dass es dem einer REST-Implementierung ähnlich ist. Damit würde ich aber nur die eben genannten Vorteile GraphQLs bekomen und dessen Potential noch nicht völlig ausschöpfen. Man sollte hier graph- bzw. objektorientiert denken und weniger serviceorientiert, wie REST oftmals fälschlicherweise umgestezt wird. In dem eben aufgezeigten

Beispielschema habe ich bei dem Typ `Metadata` noch das Feld `randomExample(count:Long): [MetadataExample!]!`. Die Felder mit Parametern stehen nicht nur in Wurzeltypen wie `Query` oder `Mutation` zur Verfügung, sondern können in allen Typen verwendet werden. So kann ich später in GraphQL nicht nur über lange Pfade tief in den Graphen „abtauchen“, sondern diesen „unterwegs“ auch noch über Parameter beeinflussen, um das ganze an dieser Stelle einmal etwas bildhafter auszudrücken. In `REST` ist derartiges natürlich auch möglich, aber nicht mit einer einzigen Anfrage. Hier muss ich so beispielsweise erst einmal die Metadaten-Tags über ähnliche Suchparameter wie in getMetadataTags finden, an denen ich interessiert bin. Habe ich dann eine Liste von 100 Metadata-Tags bekommen und möchte zu jedem ein Beispiel bekommen, dann muss ich dafür 100 weitere *API-Calls* absetzen, um an diese Ergebnisse zu kommen. („GraphQL is the better REST“, 2021)

Bevor ich diese kurze Einführung beende, möchte ich noch erklären, was der Unterschied zwischen `Query` und `Mutation` ist: Der GraphQL-Spezifikation folgend sollten in einem `Query` nur Leseanfragen möglich sein, die den Zustand auf dem Server nicht ändern, während Anfragen die über `Mutations` gestellt werden, den Serverzustand ändern dürfen. Ansonsten gibt es nicht viel weitere Unterschiede.(Facebook, 2018)

GraphQL bietet noch viel mehr Möglichkeiten an, um anspruchsvolle Schemata und Abfragen gestalten zu können. Dafür empfehle ich einen Blick auf deren Webseite https://graphql.org/.

Die Beispiele in dem Kapitel habe ich selbst erstellt und kommen im MdH zum Einsatz. Das dafür benötigte Wissen habe ich aus der eben erwähnten Webseite (Foundation, 2021) und aus der technischen GraphQL-Spezifikation. (Facebook, 2018) Die Vergleiche mit REST sind einfach ein Gegenüberstellen beider Technologien, wobei ich mir diese Quelle „GraphQL is the better REST“ (2021) zur Hilfe genommen habe.

2.2.2 GraphQL Java

GraphQL Java ist eine `Open Source` Implementierungauf Basis der offiziellen GraphQL Spezifikation. Darüber hinausgehend bietet es noch ein paar weitere Funktionen, die so in der Spezifikation nicht beschrieben sind. Leider ist deren offizielle Dokumentation nicht ganz synchron mit ihrer aktuellen Code-Basis und hat auch nicht alle Funktionen darin dokumentiert. Um derartige Funktionen dann doch nutzen zu können hilft aber ein Blick in deren GitHub Repository. Fragen dazu in ihrem Diskussionsthread in GitHub sind auch immer gern gesehen. Dass die Dokumentation nicht immer ganz auf dem aktuellen Stand ist, verwundert nicht, da dieses Projekt nur zwei ehrenamtliche Hauptentwickler hat, die dieses Framework seit 2015 entwickeln. („GraphQL Java GitHub“, 2021)

GraphQL Java hat keinen HTTP-Server inkludiert. Das heißt, dass GraphQL Java auf Programmierebene eine Schnittstelle anbietet, an die der `GraphQL Query`, sowie ein paar weitere Daten, bspw. Kontext-Daten übergeben werden können. Um an die Daten zu kommen, die hier übergeben werden, ist es aus Programierersicht erforderlich einen HTTP-Server zu starten, der diese entgegen nimmt. GraphQL muss bevor es Anfragen verarbeiten kann vorab konfiguriert werden. Dazu zählt anfangs erst einmal das GraphQL-Schema, das diesem Framework bekannt gemacht werden muss. Dies kann über eine Schema-Datei passieren, die GraphQL übergeben wird, oder auch programmatisch, mit entsprechenden Methodenaufrufen. (Marek, 2021)

Nun ist es erforderlich, vor allem die oben erwähnten Felder in den Wurzeltypen mit sogenannten DataFetchern zu verbinden. Dazu muss ich programmatisch über einen Parameter auf ein Feld verweisen, und diesem dann einen DataFetcher anhängen. Das GraphQL Framework stellt hierfür ein Interface DataFetcher bereit, das eine von mir geschriebene DataFetcher-Klasse implementieren muss. Dies ist möglich, weil das Interface nur eine öffentliche Methode hat, diese DataFetcher nicht als eigene Klassen auszuprogrammieren, sondern diese in GraphQL Java als Lambda-Funktionen zu definieren. (Marek, 2021)

```
//Verbinden eines Feldes mit einem DataFetcher:
runtimeWiringBuilder.
     type(newTypeWiring("Query")
         .dataFetcher("getMetadataTags", this::getMetadataTags)
         .dataFetcher("getFileTypes", this::getFileTypes)
         ...

//Methoden, die dann als Lambda verwendet werden:
protected List<MetadataSchema>
            getMetadataTags(DataFetchingEnvironment dfe)
   {
      //mache etwas und gebe eine Liste
      //von MetadataSchema zurück
   }

protected List<FileTypeSchema>
           getFileTypes(DataFetchingEnvironment dfe)
   {
      //mache etwas und gebe eine Liste
      //von FileTypeSchema zurück
   }

   ...
```

Zurückgeben kann ich in diesen Lambda-Funktionen `Plain Old Java Objects (POJO)`, `MAPs`, `Java-Primitive` und auch `Listen davon`. Diese werden dann von GraphQL Java ausgelesen und in das Rückgabe-JSON gepackt. Sollten die Lambda-Funktionen etwas anderes zurück geben, als es aus dem GraphQL-Schema hervorgehend erwartet wird, dann gibt GraphQL Java dem Benutzer hier im JSON einen Fehler zurück. (Marek, 2021)

Vor allem wenn hier ein `POJO` zurückgegeben wird, was beispielsweise `MetadataSchema` ist, kann dieses selbst weitere `POJOs` inkludiert haben. GraphQL wird dann gemäß der Anfrage dieses Objekt soweit und dahingehend traversieren und die Ergebnisse in das Ausgabe-JSON packen, wie es der Benutzer in seiner Anfrage definiert hat. (Marek, 2021)

Das jedem DataFetcher übergebene `DataFetchingEnvironment dfe` ist ein Objekt, über das ich beispielsweise die Feldparameter bekommen kann, oder auch welche Felder überhaupt angefragt werden. Um das vorherige Kapitel noch einmal kurz aufzugreifen: Ich könnte hiermit in Erfahrung bringen, dass der Client nur am Namen eines Metadaten-Tags interessiert ist, aber nicht nach dem `count` fragt. Somit kann ich beispielsweise etwaige SQL-Abfragen dahingehend optimieren und das `POJO` dann nur mit den Daten befüllen, die das Framework später aus diesem auslesen wird. (Marek, 2021)

Eine Besonderheit sei hier noch zu erwähnen. Statt der `POJOs`, `Maps`, `etc.` kann ich hier auch ein `Future`, um genauer zu sein ein `CompletableFuture` zurückgeben, das es mir ermöglicht, Ergebnisse asynchron zu berechnen. GraphQL Java führt dann die notwendigen `join()`-Operationen aus, um an das eigentliche Ergebnis zu kommen. (Marek, 2021)

`CompletableFuture` ist eine implementierende Klasse vom Interface `CompletionStage`, das ich in einem späteren Grundlagenkapitel noch etwas genauer erkläre, aber für die Umsetzung des GLS dann in einem anderen Kontext benötige, als in dem, den mir das GraphQL Framework anbietet.

Für weitere Information empfehle ich einen Blick sowohl auf deren Webseite https://graphql.java.com/ (Marek, 2021) als auch auf deren GitHub-Seite („GraphQL Java GitHub“, 2021), die mir für dieses Unterkapitel beide als Quelle dienten.

Als weiterführende Literatur kann ich hier noch abschließend zu diesem Thema das Buch „GraphQL: Eine Einführung in APIS mit GraphQL“(Kress, 2020) nennen, das einerseits erklärt, wie man eine GraphQL API im Allgemeinen entwerfen sollte, andererseits auch noch ein paar auf GraphQL Java zugeschnittene Kapitel beinhaltet.

2.3 Architektur Metadata-Hub

In diesem Kapitel beschreibe ich die Architektur insbesondere des Java GraphQL Servers, da dieser dann später für die Umsetzung des Global Search die entscheidende Komponente ist, die dafür angepasst und erweitert werden muss. Vorab gehe ich hier aber auch noch auf die grobgranulare Architektur im Gesamten ein.

2.3.1 Grobgranulare Architektur

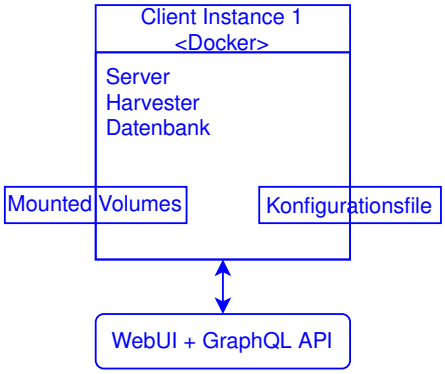

Abbildung 2.6: MdH Docker Container

Wie in in Abb. 2.6 zu sehen, ist unsere Software in einem Docker Container, mit dem ein Client nur über definierte Schnittstellen kommunizieren kann. Und dazu gehören die dem Docker Container bekannt zu machenden Verzeichnisse, welche die Software dann zum Extrahieren der Metadaten verwenden kann. Außerdem hat der Container so die Möglichkeit die Datenbankdaten außerhalb des Containers zu speichern, sodass diese bei einem Ersetzen des Containers z.B von einem neueren Container weiterverwendet werden können. Über die Konfigurationsdatei wird der Container auf einer grobgranularen Ebene konfiguriert. Der Name dieser Docker-MdH-Instanz, Passwörter, das Logging-Verhalten und weiteres kann darüber konfiguriert werden. Zuletzt ist dann noch eine Netzwerkschnittstelle in den Dockercontainer hinein freigegeben, über diese ein Benutzer die WebUI aufrufen und mit der GraphQL-Schnittstelle kommunizieren kann.

Diese Schnittstelle ist an den, wie in dieser Abbildung zu sehen, Server geknüpft, der die dafür notwendige Funktionalität bereitstellt. Der Harvester ist ein in Python geschriebenes Programm, dessen Hauptaufgabe bislang ist, die Dateien aus den dem Container zur Verfügung gestellten Verzeichnissen zu durchsuchen und deren Metadaten gegebenenfalls etwas nachbearbeitet und schließlich in die Datenbank ablegt. Dabei persistiert er die Metadaten in der `Files`-Tabelle. Außerdem schreibt er die Metadaten noch aggregiert in die `Metadata`-Tabelle. Diese enthält so beispielsweise direkt ablesbar Informationen darüber, wie oft ein bestimmter Metadata-Tag vorkommt, von welchen Datentypen dieser ist. Also ob

vom Typ `String`, `Numerisch` oder `Timestamp`. Weiterhin speichert der Harvester in diese Tabelle diese Informationen aufgeteilt in `Filetypen` und `Mimetypen`. Diese Tabelle dient dem Server dann dazu, die Statistiken direkt anzeigen zu können ohne diese Daten vorher selbst noch einmal aus der `Files`-Tabelle mit z.B. mehreren 10 Millionen Einträgen berechnen zu müssen. Außerdem dient dieser Datensatz dazu, die SQL-Suchanfrage zwecks „Datentypencasten“ optimiert zusammen bauen zu können. Die Datenbank ist eine `PostgresSql` Datenbank.

Der Harvester hat selbst keine Schnittstelle nach außen, über die ein Benutzer mit diesem direkt kommuniziert könnte. Der Server übernimmt die externe Kommunikation mit dem Harvester. Dabei dient der Server als ein Art Proxy für den Harvester und da der Harvester intern selbst kein GraphQL verwendet, dient der Server in dem Fall auch als ein Adapter, diese Nachrichten für den Harvester verständlich zu machen.

Jegliche Drittsoftware, aber auch die von uns zur Verfügung gestellten Tools wie der MdH-CTL (ein Kommandozeilentool, das eine native Schnittstelle anbietet, darüber direkt in der Sprache Python programmieren zu können) verwenden die GraphQL-Schnittstelle, um mit dem Server, letztendlich dann auch mit dem Harvester kommunizieren zu können. Dem zur Folge bedient sich auch die WebUI dieser Schnittstelle.

Diese ist als `Singlepage`-Anwendung konzipiert und ändert ihren Zustand bzw. das, was sie anzeigt nur darüber, was sie vom Server über die GraphQL-API an Informationen erhält. Es wäre an der Stelle naheliegend, hierbei bspw. auf ein Framework wie `React.js` oder `Vue.js` zurückzugreifen. Aus historischen Gründen verwendet die WebUI hier aber ein minimalistisch von mir selbst entworfenes Framework, das die WebUI durchaus in Komponenten unterteilen lässt. Diese sind dann aber in sich weniger deklarativ, sondern eher klassisch iterativ auszuprogrammieren. Dabei setze ich auf JQuery auf. Um aus den ganzen Einzel-JavaScript-Dateien dann ein `Bundle` zu erzeugen und auch um in die WebUI einfach weitere Abhängigkeiten aufnehmen zu können setze ich hierbei auf den Paketmanager NPM[1] und auf Webpack[2].

Die Abb. 2.7 fast diese grobgranulare Architektur noch einmal zusammen. Ergänzend ist hier noch hinzuzufügen, dass der `GraphQL API Server` aus der Grafik natürlich auch direkt mit dem `Harvester with Extractors` kommunizieren kann, darauf aber in der Grafik verzichtet wurde, um sie nicht zu überladen.

[1] https://www.npmjs.com/
[2] https://webpack.js.org/

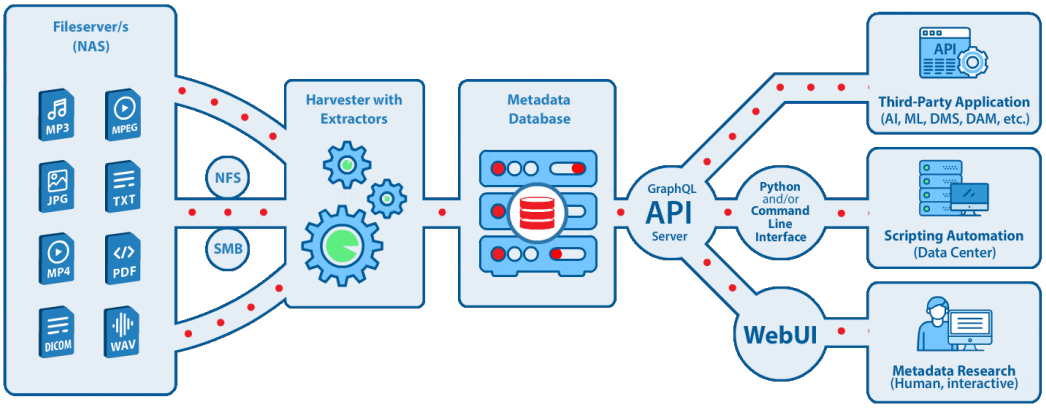

Abbildung 2.7: Grobgranulare Architektur (GRAU-DATA, 2021)

2.3.2 Architektur GraphQL Java Server

Um die Architektur des Java-GraphQL-Servers gut verstehen zu können, hilft ein Blick auf Abb. 2.8. Das darin enthaltene Diagramm zeigt sehr gut die Komponenten, die den Server definieren und deren Zusammenspiel zeigen.

- **HTTP-Server**

  Der `HTTP-Server` nimmt die Anfragen vom Client, der ein Browser oder ein anderes Drittprogramm sein kann, an. Die Anfrage selbst ist eine normale HTTP-Anfrage mit `GET`, `POST` und weiteren `HEADER`-Daten. Als Antwort sendet der Server eine HTTP-Antwort, die neben möglichen Daten auch einen Status-Code enthalten kann, wie z.B. den Statuscode `200` im Erfolgsfall. Der HTTP-Server analysiert die Anfragen und entscheidet dann, an welchen `HTTP-Controller` diese weitergeleitet werden soll. Neben eher sekundären in der Abbildung als `Andere Controller` bezeichneten `Controllern`, ist der `GraphQL-Controller`, der, an den der Großteil der Anfragen weitergeleitet wird.

- **(Auth-)Filter**

  Die Filter sind für Aufgaben gedacht, die alle Controller betreffen. Diese können genau wie die Controller selbst auch auf die Anfrageparameter zugreifen. Somit kann ein `Authentifikationsfilter` beispielsweise prüfen, ob das Passwort, das über ein Feld im `HEADER` übertragen wird, richtig ist und welche Benutzerrechte zu diesem Passwort gehören. Auf diese Weise kann ein Filter schon vorab auf einer grobgranularen Ebene Anfragen an die `Controller` unterbinden, auf die der Benutzer grundsätzlich keinen Zugriff haben soll. Der `GraphQL-Controller` ist ausschließlich über einen eingeloggten Benutzer zugreifbar. Da dieser `Controller` aber sämtli-

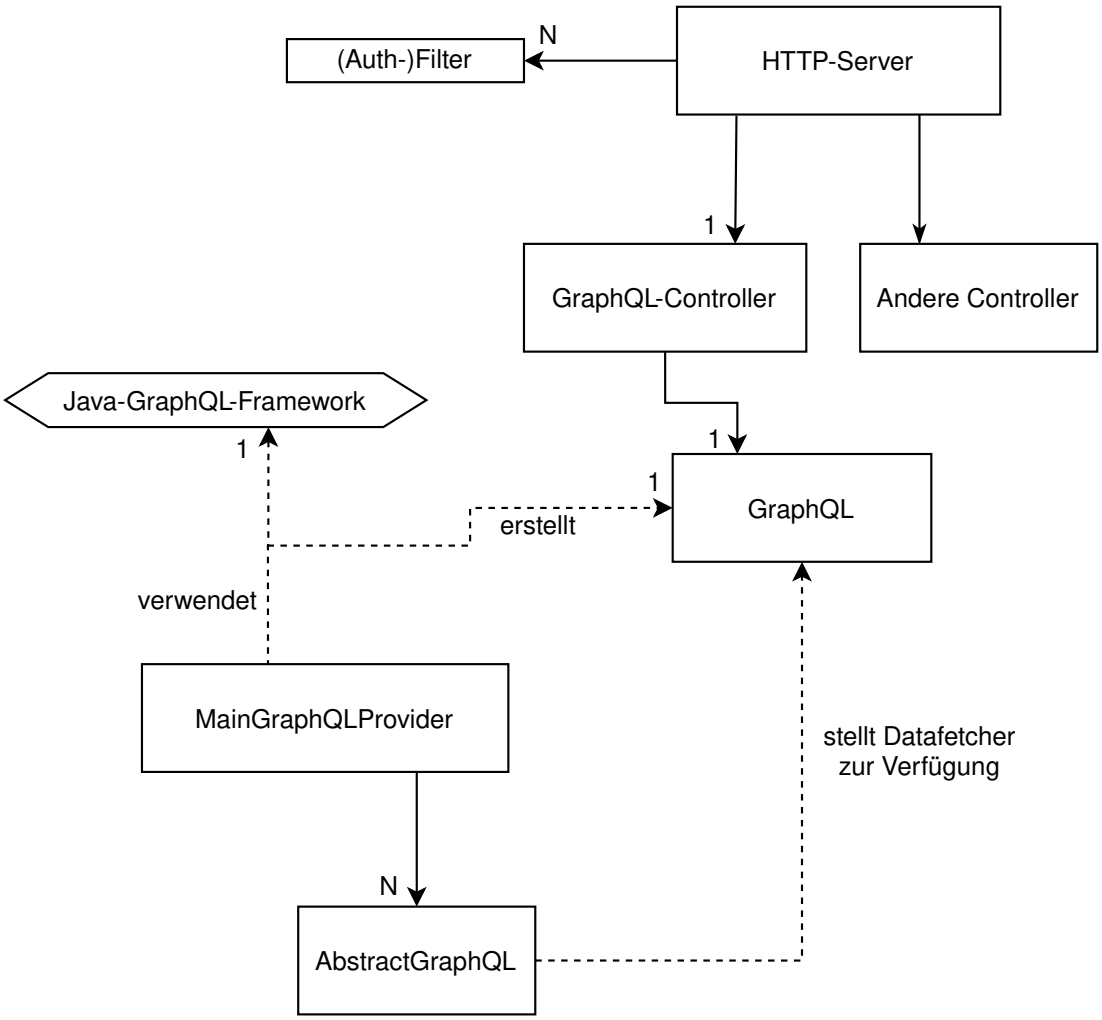

Abbildung 2.8: Architektur Java-Server Teil 1

che GraphQL-Anfragen bearbeitet, können hier auf der Filterebene diesbezüglich keine fein granulareren Berechtigungen geprüft werden. Dies übernehmen die DataFetchern dann selbst.

- **Andere Controller**

  Hier sind derartige `Controller`, die bspw. für das Einloggen zuständig sind, denn das ist ein Teil, bei dem GraphQL noch nicht involviert ist. Außerdem gibt es hier auch noch `Hilfs-Controller` um verschiedene Anfragen aus unserer WebUI bearbeiten zu können, die von WebUI-Bibliotheken generiert werden, aber nicht über GraphQL kommunizieren können. Dabei ist aber zu beachten, dass der dann in dem Fall als `Rest-Controller` arbeitende Empfänger, als Adapter fungiert, der diese Anfragen schließlich in GraphQL-Anfragen umwandelt.

- **GraphQL Controller**

  In diesem `Controller` kommen alle GraphQL Anfragen an. Eine dartige Anfrage kann bspw. über einen `GET`-Parameter verschickt werden. wie z.B `http://Server-URL/graphqlEndpoint?data=GraphQL-Query`. Außerdem kann dieser `GraphQL-Query` noch direkt über `POST` oder als `JSON` kodiert über `POST` ankommen. Letzteres ist vor allem deshalb noch interessant, da neben dem `Query`, dessen Inhaltsparameter getrennt von diesem `Query` gesendet werden können.

  Der `GraphQL-Controller` übergibt diese Anfrage nun an das zentrale `GraphQL-Objekt`. Außerdem erstellt er für dieses Objekt noch ein `Kontext-Objekt`, auf das die DataFetcher dann zugreifen können. Darin enthalten sind bspw. die Anfrage an sich im Klartext, Informationen, welcher Benutzer mit welchen Benutzerrechten gerade einloggt ist, damit die DataFetcher hierbei dementsprechend später benutzerspezifisch differenzieren können, und so ihren Dienst verweigern können, wenn ein normaler Benutzer versucht auf Funktionalitäten zuzugreifen, die für den Administrator gedacht sind. Da später auf die ursprüngliche Anfrage nicht mehr zugegriffen werden kann, diese aber für das Mitloggen zu einem späteren Zeitpunkt und für anderes noch relevant ist, reiche ich auch diese über das `Kontext-Objekt` durch.

  Das `GraphQL-Objekt` wird schließlich angewiesen, die Anfrage auszuführen. Das Ergebnis, das der `GraphQL-Controller` als gemäß der GraphQL-Spezifikation entsprechenden `JSON`-Format von diesem zurückerhält wird so an den HTTP-Server und schließlich an den anfragenden Client zurückgegeben. Hierbei wird, egal ob die Anfrage zu einem Ergebnis ohne Fehler oder mit Fehlern führt, die immer als `JSON` kodierte Antwort in jedem Fall mit dem Statuscode `200` zurückgegeben.

- **Java-GraphQL-Framework**

  Darin enthalten ist `https://github.com/graphql-java/graphql-java` („GraphQL Java GitHub“, 2021)

- **GraphQL**

  Das ist das zentrale Objekt, mit dem der GraphQL-Controller kommuniziert. Dieses erstellt das Java-GraphQL-Framework basierend auf den Regeln, die es vom MainGraphQLProvider zur Verfügung gestellt bekommt. Darin enthalten ist dann das GraphQL-Schema, die dazugehörigen DataFetcher und weiteres wie z.B. Datenstrukturen, die den GraphQL-Ausführungs-

  prozess instrumentieren und verändern können. Letzteres ist aber weniger für das Grundlagenkapitel interessant, später dann für die Erweiterungen in Richtung Global Search schon.

- **MainGraphQLProvider**

  Ein Objekt dieser Klasse steuert mittels des GraphQL Frameworks den Bauprozess des zentralen `GraphQL-Objekts`. Dabei stellt er diesem die GraphQL Schema Beschreibung zur Verfügung und verbindet meist die `GraphQL-Felder`, die direkt in `Query` und `Mutation` definiert sind mit den von mir zur Verfügung gestellten DataFetchern. Weiterhin kann dieser Provider dem Bauprozess Code, der für die Instrumierung notwendig ist zur Verfügung stellen. Sobald das `GraphQL-Objekt` gebaut ist, kann es in seiner Konfiguration nicht mehr verändert werden.

- **AbstractGraphQL**

  Damit ich nicht den gesamten Konfigurationscode in den `MainGraphQL-Provider` schreiben muss, teile ich mir diesen Vorgang mit Hilfe der `AbstractGraphQL`-Klasse und den von dieser abgeleiteten Klassen auf. Damit ist es mir möglich, den Programmcode nach?„Features“ zu strukturieren. Eine von `AbstractGraphQL` abgeleitete Klasse beinhaltet einen Teil der GraphQL-Spezifikation und die zu dieser dazugehörigen DataFetchern. Der `MainGraphQLProvider` führt dieses dann zu einem „Großen und Ganzen“ zusammen. Schlussendlich werden die DataFetcher von dem gebauten `GraphQL-Objekt` verwendet. Abb. 2.9 illustriert `AbstractGraphQL` noch einmal etwas detaillierter. Dabei können dessen konkrete Unterklassen jeweils einen oder mehrere DataFetcher haben. Zur Namensgebung ist hier noch folgendes zu erwähnen: Alle in Abb. 2.8 gezeigten Klassen sind eher Hilfsklassen, um das GraphQL-Framework mit dem Http-Server genau wie mit den DataFetchern sinnvoll zu verbinden und zu konfigurieren. Die DataFetcher mit ihren zugehörigen Container Klassen stellen dann den eigentlichen „Feature-Code“ dar. Dies wurde auch in der „Java-Paketierung“ so

beachtet und damit ist die `GraphQL`-Klasse aus Abb. 2.8 keine Unterklasse von `AbstractGraphQL`, da die von `AbstractGraphQL` abgeleiteten Kindklassen von mir in ein anderes `Java-Package` eingegliedert worden sind. Der Name GraphQL bzw. dessen Gebrauch als Postfix soll im „Feature"-Code hervorheben, dass die Klassen, die diesen Namen verwenden, jeweils den Einsprungpunkt von „außen" darstellen.

Die DataFetcher sind im MdH oft als Lambda-Funktionen direkt in deren Containerklasse programmiert, statt diese als „vollwertige" Java-Klasse zu programmieren. Da eine Lambda-Funktion auch auf seine Umgebung Zugriff hat, habe ich das in Abb. 2.9 mit Doppelpfeilen angedeutet.

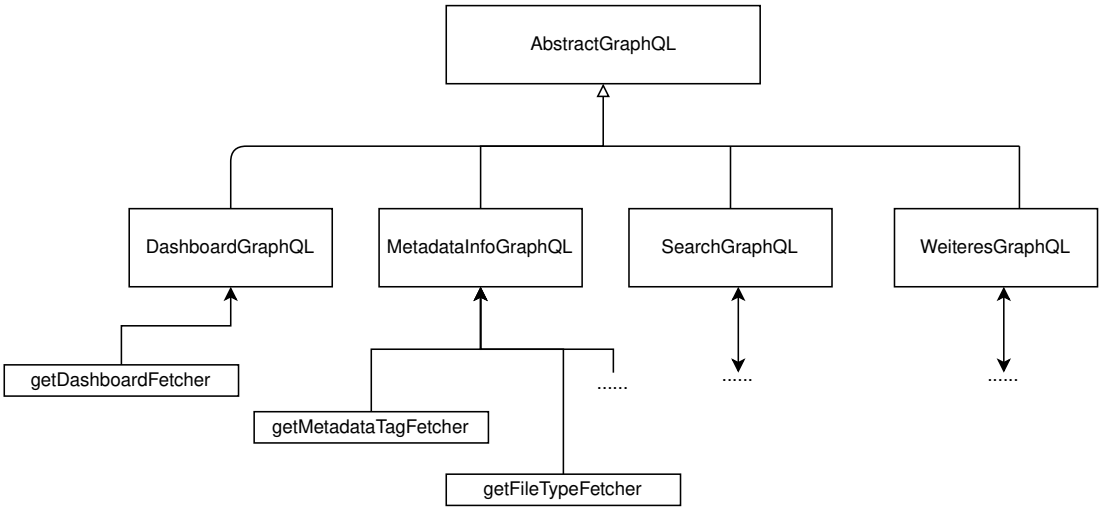

Abbildung 2.9: Architektur Java-Server Teil 2

Um sich das ganze in Aktion etwas besser vorstellen zu können, gehe ich jetzt einen Beispielaufruf bildhaft durch.

Folgender `GraphQL-Query` wird mit einem Passwort im `HEADER` Feld `X-Authorization-Bearer` an `http://server-url/graphqlendpoint/?data=...` gesendet.

```
  query{
  systemInfo{
    instanceName
    installDate
    harvestedFiles
  }
}
```

Der `Http-Server` erkennt anhand des Pfades `graphqlendpoint`, dass der `GraphQL-Controller` für die weitere Bearbeitung zuständig ist und übergibt ihm die Anfrage. Zuvor hat der `(Auth-)Filter` anhand des Passworts im Header erkannt,

dass der User berechtigt ist, auf die Ressource `GraphQL-Controller` zuzugreifen. Dieser bereitet die Anfrage für das GraphQL-Objekt vor. Dazu erstellt er ein Kontext-Objekt, in das er die Anfrage an sich als Klartext packt und außerdem die über den `Filter` zuvor in Erfahrung gebrachten Benutzerrechte. Das `GraphQL-Objekt` erkennt, dass es für das Feld `systemInfo` ein entsprechender DataFetcher vorhanden ist und übergibt die Anfrage an diesen. Dieser DataFetcher hat nun die Möglichkeit auf das `Kontext-Objekt` zuzugreifen, aber auch auf die Anfrageparameter im `GraphQL-Query` wie z.B, dass die drei Felder `instanceName`, `installDate` und `harvestedFiles` von Interesse sind. Grundsätzlich stünden in `systemInfo` auch noch weitere Felder zur Verfügung, diese werden aber in der jetzigen Anfrage nicht weiter abgefragt. So muss der DataFetcher deren zugehörigen Daten gegebenenfalls auch nicht aus der Datenbank laden.

Der DataFetcher gibt schließlich ein `Plain Old Data Object` zurück, das die angefragten Daten enthält. Er könnte hier auch einfach eine `Map` mit den dazugehörigen Daten zurückgeben. Das GraphQL-Framework ließt dieses aus und erstellt dafür die entsprechende `JSON`-Antwort, die über das `GraphQL-Objekt`, über den `GraphQL-Controller` und den `HTTP-Server` zurück an den Benutzer gegeben wird.

2.4 Entwurfsmuster

Entwurfsmuster sind eine Art Bauplan für immer wiederkehrende Probleme im Softwaredesign. Folgendes kurzes Beispiel sei an dieser Stelle genannt. Um das Verhalten eines Objekts während der Laufzeit zu ändern kann ich das Strategiemuster verwenden. Dieses zeigt mir, wie ich programmieren muss, um das Verhalten während der Laufzeit eines Objekts zu ändern, sodass ich für diese Problemstellung „das Rad nicht immer wieder erneut erfinden“ muss. (Gamma et al., 2015)

Die womöglich bekannteste Sammlung von Entwurfsmustern ist die der *Gang of Four*. Von diesen stammt auch das Fabrikmuster, das ich im folgenden Unterkapitel vorstellen werde. Aber auch das eben erwähnte Strategiemuster ist von ihnen. (Gamma et al., 2015)

2.4.1 Fabrikmuster

Das (abstrakte) Fabrikmuster gehört zur Gattung der Erzeugermuster. Das sind Muster, mit denen sich Objekte auf eine ganz bestimmte Art und Weise erzeugen lassen und dementsprechend diverse Vorteile, manchmal aber auch Nachteile mit sich bringen. (Gamma et al., 2015)

Um dieses Muster zu erklären, greife ich nochmal auf das Beispiel in Abb. 2.9 zurück. Darin sind folgende Klassen zu sehen: `DashboardGraphQL`, `MetadataInfo-`

`GraphQL`, `SearchGraphQL` und weitere. Diese Klassen lassen sich beispielsweise einfach mit `new DashboardGraphQL` instanziieren. Nun wird es später für die Global-Search Implementierung aber so sein, dass ich hier zwar die gleichen Schnittstellen, auch die gleichen darin enthaltenen DataFetcher-Schnittstellen brauche, aber mit einem anderem Verhalten darin. Und gerade für so etwas bietet sich das abstrakte Fabrikmuster an.

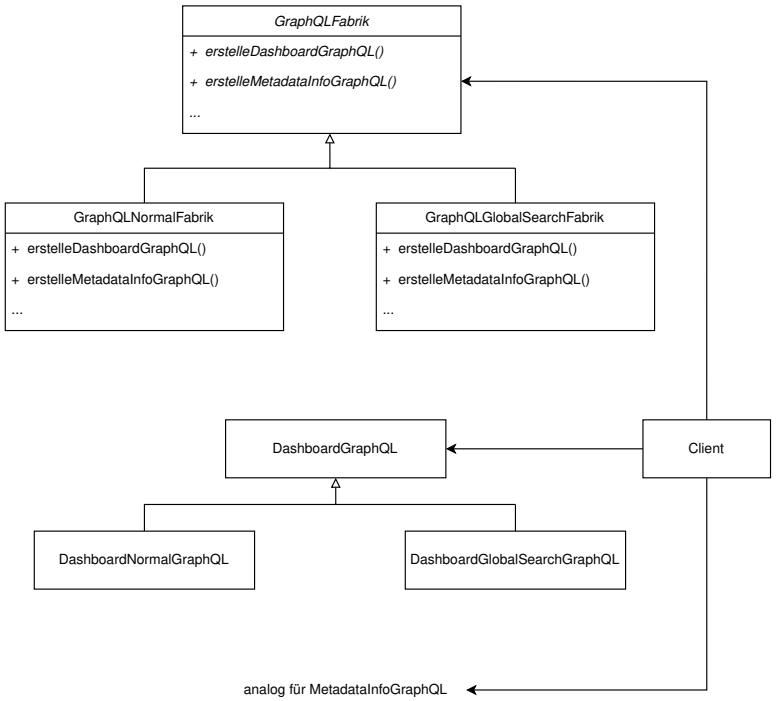

Abbildung 2.10: Abstrakte Fabrik (vgl. Gamma et al., 2015)

In Abb. 2.10 ist die kursiver Schrift geschriebene `GraphQLFabrik` als die abstrakte Fabrik zu sehen. Konkrete Instanziierungen gibt es dann nur von dessen abgeleiteten Klassen wie `GraphQLNormalFabrik` und `GraphQLGlobalSearchFabrik`. Der Client muss die konkrete Fabrik nicht kennen. Zum Konfigurationszeitpunkt wird beispielsweise festgelegt, welche Fabrik der Client oder andere Klassen erhalten, um sich damit dann in Folge Objekte einer bestimmten Familie zu erzeugen. Mit Familie meine ich hier folgendes: Alles was zum Global Search gehört wäre eine Familie und die Dinge, die zum normalen Metadata-Hub gehören wären eine andere Familie. Die Objekte, die dann hierbei erzeugt werden, erben ihrerseits von einer Elternklasse, mit dessen Schnittstelle dann der Client kommuniziert.

Um es konkret zu machen: Je nach dem, ob der MdH später eine normale Instanz sein soll oder als Global Search agieren soll, bekommt der Client die entsprechende Fabrik. Der Client selbst oder auch diese Fabrik weitergegeben an andere Klassen kann/können sich dann die für sich/sie relevanten Objekte erzeugen. Somit kann an der Stelle an der ein `DashboardGraphQL`-Objekt benötigt wird,

dieses aus der Fabrik heraus erstellt werden, die konkrete Implementierung ist für das Objekt, das `DashboardGraphQL` verwendet, irrelevant. Entscheidend ist nur, dass es die notwendigen Schnittstellen ansprechen kann, was aber durch den Vererbungsmechanismus von `DashboardGraphQL` zu `DashboardNormalGraphQL` beziehungsweise `DashboardGlobalSearchGraphQL` gegeben ist.

Um das schon einmal vorweg zu nehmen, ich kann, wenn ich das Entwurfsmuster abstrakte Fabrik verwende, also gezielt Klassen und damit vor allem `Verhalten` austauschen in Abhängigkeit, wie meine Anwendung konfiguriert wurde unter Beibehaltung des restlichen Ökosystems. Damit können so also beispielsweise die Schnittstellen, die ich in meinem GraphQL-Schema über Felder und Typen definiert habe gleich bleiben, nur wird, je nachdem welche Rolle (`Normal`, `Global Search`) ich dem Mdh konfigurierend zugeordnet habe, dann eine andere `XXXX-GraphQL]`-Klasse und somit auch andere DataFetcher verwendet. Dies ist meinen Implementierungskapiteln schon einmal etwas vorgegriffenen und ich komme darauf später noch einmal zurück.

Das Beispiel und die Implementierung ist von mir, das Wissen über das darin enthaltene Fabrikmuster habe ich aus Gamma et al. (2015) entnommen

2.4.2 Dependency Injection

Während ich beim Fabrikmuster die Objekte selbst erstellen muss, verfolgt `Dependency Injection`, kurz DI, einen anderen, ich würde aber nicht sagen ersetzenden sondern viel mehr erweitertenden Ansatz im Vergleich zur Fabrik. („Dependency Injection vs Factory Pattern“, 2009)

Wie der englische Name DI schon andeutet, wird hierbei eine Abhängigkeit irgendwo hinein injiziert. Und das hat zur Folge, dass sich Objekte, weitere Objekte, die sie brauchen, nicht mehr selbst erstellen, sondern von außen injiziert bekommen. Das kann zur `Konstruktionszeit` passieren, aber auch später über `Setter`-Methoden. (Fowler, 2004) Als ich vorhin davon geschriebenen habe, dass der Client die Fabrik an weitere Klassen weitergibt, dann ist das schon als Art DI zu sehen. Dennoch wird das eigentliche Zielobjekt dann erst innerhalb dieser Klassen erzeugt, was dann kein DI mehr ist.

Frameworks wie `Google Guice` ermöglichen aber auch genau das. Hier schreibe ich in die Klassen, die bspw. `DashboardGraphQL` benötigen, dies einfach als Membervariablen in diese und `Google Guice` injiziert dorthin dann eine Instanz einer konkreten implementierenden Klasse wie z.B die Global-Search- oder die Normal-Variante. Auch hier kann `Google Guice` vorab konfiguriert werden, dass es dann je nach MdH-Konfiguration die Global Search oder die normalen Klassen injiziert. („Guice“, 2021) Insofern ist dieser abstrakte Fabrikmechanismus in diesem DI-Framework versteckt („Dependency Injection vs Factory Pattern“, 2009)

.

Da DI aus Programmiersicht meiner Ansicht nach die noch etwas elegantere Lösung bietet, findet auch genau das Anwendung im MdH.

2.5 CompletableFuture in Java

`CompletableFuture` implementiert als einzige Klasse das Interface `CompletionStage`, außerdem implementiert diese Klasse das Interface `Future`. Damit stellt diese Klasse eine Erweiterung des normalen Java-`Future` dar. („Class CompletableFuture<T>“, n. d.)

Ein normales `Future` bietet Operationen an, über die das `Future` asynchron ein Ergebnis aufnehmen kann und später dann auch von einem anderen Thread über beispielsweise die `get()`-Methode abgerufen werden kann. Möchte man dieses Ergebnis nun asynchron weiterverarbeiten und wieder abrufen und dies nicht nur ein weiteres mal, sondern wesentlich öfters, dann lässt sich das mit normalen Futures durchaus lösen, aber mit sehr viel „Boilderplatecode“. Und da kommt jetzt das `CompletableFuture` ins Spiel. Dieses bietet für das eben beschriebene Szenario die notwendige Infrastruktur. Statt das Ergebnis selbst und direkt zurückzugeben, gibt das `CompletableFuture` bspw. mit `thenApplyAsync()` ein weiteres, ein neues `CompletableFuture` zurück. Und daran anknüpfend kann die Kette mit `thenApplyAsync()` fortgesetzt werden. Jedes dieser `thenApplyAsync()`-Aufrufe stellt eine eigene Stufe (Stage) dar, die das Ergebnis der vorherigen Stufe bekommt und ein neues Ergebnis berechnet und dieses dann an die nächste Stufe weitergibt. Schließlich kann ich das Endergebnis von der letzten Stufe mit dem bereits bekannten `get()` aus diesen asynchronen Berechnungsstufen herausholen. Wie der Name von `thenApplyAsync()` vermuten lässt, wird das Ergebnis hierbei nicht einfach synchron angenommen, sondern asynchron weiterverarbeitet. Das ist noch ein Vorteil von `CompletableFutures`, da sie derartige asynchrone Verarbeitung bereits von sich aus bereitstellen und so es keines externen `Executors` bedarf. (baeldung, 2021)

Ein kurzer Beispielcode aus baeldung (2021):

```
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> future = completableFuture
  .thenApplyAsync(s -> s + " World");

assertEquals("Hello World", future.get());
```

Standardmäßig verwendet `CompletableFuture` den Java Standard `ForkJoinPool .commonPool()` als `Executor`. Dieser kann aber für jede Stufe durch einen anderen `Executor` überschrieben werden.(baeldung, 2021) Den `Executor` ändern zu können macht durchaus Sinn. Der Java Standard `ForkJoinPool.commonPool()-Executor` hat maximal so viele Threads wie die CPU an Kernen hat, auf dem das Programm dann läuft, und ist somit für CPU-intensive Aufgaben sinnvoll, während ich für `IO`-intensive Aufgabe lieber einen `Executor` mit sehr viel mehr Threads haben möchte.(Gopal, 2019)

`CompletableFutures` bieten noch einen weiteren Vorteil. Diese reichen Exceptions, die irgendwo in der Stufen-Kette geworfen werden, der Kette entlang nach unten ohne weiteres Zutun durch, sodass diese Kette sauber beendet werden kann. Das `get()` am Ende gibt dann statt des Ergebnisses die Exception aus. (baeldung, 2021)

GraphQL selbst bietet, wie ich bereits geschrieben habe, die Möglichkeit, anstelle des Ergebnis direkt zurückzugeben auch ein `CompletableFuture` zurückzugeben. Für diese Arbeit verwende ich die `CompletableFutures` aber in einem anderen Kontext und zwar direkt in meinen Algorithmen. Wie und weshalb das Sinn macht, erkläre ich später in meinen Implementierungskapiteln.

Für mehr Informationen und dafür, wie feingranular sich `CompletableFutures` einsetzen lassen, empfehle ich einen Blick auf den Blogeintrag von baeldung (2021), der mir auch als Grundlage für dieses Kapitel diente.

3 Anforderungen

3.1 Funktionale Anforderungen

3.1.1 Auswahlmöglichkeit der Clients

Der Global Search muss erfahren können, wie er die Client-Instanzen erreichen kann, die er orchestrieren soll. Dazu soll der Benutzer die Möglichkeit bekommen, diese Informationen über eine Konfigurationsdatei ähnlich wie sie bereits für normale Client-Instanzen eingesetzt wird, dem Global Search bekannt zu geben.

Neben der statischen Konfiguration, welche Client-Instanzen dem Global Search zugeordnet sind, soll der Benutzer darüber hinausgehend die Möglichkeit haben, die Client-Instanzen für jede Anfrage individuell beschränken zu können.

3.1.2 Client Instanzen in der Webui

Die Funktionalität Auswahlmöglichkeit der Clients soll dem Benutzer in der WebUI zur Verfügung gestellt werden. Außerdem soll der Benutzer die Möglichkeit haben, sich in der WebUI schnell einen Kurzüberblick über die ihm zur Verfügung stehenden Client-Instanzen machen zu können. Das beinhaltet auch, dass er sich Fehlerinformationen anzeigen lassen kann, wenn Client-Instanzen nicht erreichbar sind oder aus anderen Gründen nicht wie erwartet mit ihnen kommuniziert werden kann.

3.1.3 Einfache Proxy-Aufrufe für alle GraphQL-Schnittstellen

Für alle GraphQL Schnittstellen, die der Metadata-Hub hat, soll der Global Search eine Möglichkeit anbieten, diese beziehungsweise dahinter liegende Funktionalität als Art Proxy aufrufen zu können. Ein einfacher Proxy-Aufruf bedeutet hier, dass das Ergebnis nicht weiter verarbeitet werden soll. Ein Proxy-Aufruf, der an verschiedene Client-Instanzen geschickt wird, soll aber erlaubt sein. Die Ergebnisse können in dem Fall als Liste zurückgegeben werden. Eine Zuordnung der Listenelemente zu den jeweiligen Client Instanzen muss möglich sein.

3.1.4 Dashboard

Das Dashboard des Global Search soll die Dashboardinformationen seiner Client Instanzen sinnvoll zusammengefügt beinhalten. Das Dashboard einer normalen Metadata-Hub Instanz ist so konzipiert, dass dessen Daten schnell abgerufen werden können, da viele dieser Daten auf der Instanz zwischengespeichert in der Datenbank abgelegt sind. Das Dashboard des Global Search soll ähnlich schnell abrufbar sein.

3.1.5 Metadata-Info

`Metadata-Info` umfasst einige API-Schnittstellen, mit denen der Benutzer Informationen über einen ganz bestimmten Metadaten-Tag, oder über eine Suche über mehrere auf einmal abrufen kann. Auch zählen hierzu die Schnittstellen über die er Informationen über `Filetypen` und `Mimetypen` sowie diesen zuordenbare Metadaten-Tags erhalten kann. Diese Informationen werden für verschiedene Funktionen des Metadata-Hubs benötigt. So werden damit bspw. die Statistiken generiert oder dem Benutzer basierend auf diesen Daten Vorschläge für die Suche angeboten. Deshalb sollen die API-Schnittstellen, die Metadata-Info umfasst auch im Global Search voll funktionsfähig zur Verfügung gestellt werden.

3.1.6 Query-Storage

`Query-Storage` beinhaltet die Schnittstellen mit denen ausgefüllte Suchanfrage-Eingaben abgespeichert, später erneut geladen und auch wieder gelöscht werden können. Der Global Search soll die Möglichkeit bieten, auf den Client Instanzen abgespeicherte Abfragen zu laden. Außerdem soll er selbst Suchanfrage-Eingaben abspeichern können, diese aber nur auf seiner eigenen Instanz, also auf dem Global Search selbst.

3.1.7 Search

Der Global Search soll schließlich einen den eigentlich „Search“anbieten mittels dessen der Benutzer auf allen dem Global Search bekannten beziehungsweise vom Benutzer zur Laufzeit eingeschränkten Menge von Client-Instanzen nach Metadaten suchen kann. Außerdem soll er neben der technischen Limitierung der Client-Instanzen, diese auch als Teil der Suchanfrage selbst einschränken können. Dazu ist ein virtueller Metadaten-Tag einzuführen, nach diesem der Benutzer ganz regulär wie nach einem normalen anderen Metadata Tag auch suchen kann. Der virtuelle Metadaten-Tag soll außerdem dem Benutzer in der Ergebnistabelle anzeigen, welche Zeile zu welcher Client-Instanz gehört.

3.2 Nicht-funktionale Anforderungen

3.2.1 GraphQL für Kommunikation

Eine normale Metadata-Hub Instanz bietet derzeit eine GraphQL Schnittstelle an, über die die WebUI, das MdH-CTL und Drittanbietersoftware mit dieser kommunizieren kann. So soll folglich auch der Global Search eine Schnittstelle basierend auf GraphQL anbieten, worüber eben genannte Softwaretarifakte mit diesem kommunizieren können sollen. Außerdem soll auch die interne Kommunikation zwischen dem Global Search und seinen Client-Instanzen auf GraphQL basieren.

3.2.2 Fehlerbehandlung

Bei der Kommunikation zwischen dem Global Search und seinen Instanzen können verschiedene Fehler auftreten wie z.B, dass Client-Instanzen nicht erreichbar sind, diese unter einer anderen Version laufen als der Global Search erwartet oder weiteres. Beim Auftreten eines Fehlers, soll dem Client dieser zugänglich gemacht werden.

Es soll zudem eine möglichst starke Konsistenz erreicht werden. Das bedeutet, dass der Funktionsaufruf grundsätzlich fehlschlägt, wenn mindestens ein Client-Instanz-Aufruf in einem Fehler resultiert. Zudem sollte in diesem Fall auch kein Zustand die anderen Client-Instanzen betreffend geändert werden, zu denen Aufrufe fehlerfrei funktionieren würden.

Dieses Verhalten soll der Benutzer außerdem dynamisch zur Laufzeit abschwächen können, dass fehlerhafte Client-Instanz Aufrufe einfach ignoriert werden und das Ergebnis anhand der funktionierenden Client-Instanzen weiterhin berechnet und zurückgegeben wird.

3.2.3 Performance

Der Global Search muss Teilergebnisse von vielen Client-Instanzen abfragen und diese dann möglicherweise miteinander zu einem Ausgabeergebnis verschmelzen. Hierbei ist abzuwägen, wann mögliche Nebenläufigkeitstechniken angewandt werden können. Sofern es sinnvoll ist, sollen diese dann schließlich auch angewandt werden. Analog soll das auch für die Verschmelz-Algorithmen gelten.

3.2.4 Mindestens gleiche Schnittstelle

Diese Anforderung hier ist die mitunter wichtigste. Die WebUI erwartet wie auch die Drittsoftware eine bestimmte Schnittstelle zum Metadata-Hub. Der Global Search soll mindestens die gleiche Schnittstelle anbieten. Das hat dann zur Folge,

dass weder die WebUI noch die Drittsoftware angepasst werden beziehungsweise speziell für den Global Search erweitert werden muss. Somit kann die bereits existierende Drittsoftware einfach weiter verwendet werden. Der GLS wirkt so auf die Außenwelt als wäre er eine ganz normale Instanz mit einer viel größeren Datenmenge als es für eine normale Metadata-Hub Instanz üblich ist.

Dennoch soll der Global Search seine Schnittstellen erweitern dürfen, um so beispielsweise die Anforderung *Auswahlmöglichkeiten der Clients* auf Anfrage-Ebene umsetzen zu können. Möchte die WebUI oder Drittsoftware von dieser extra Funktionalität Gebrauch machen, ist folglich dann doch eine Anpassung dieser notwendig.

Diese Anforderung ist damit vergleichbar, wenn bei objektorientierten Programmiersprachen ein Elterntyp auf ein Kindtyp „gecastet“wird.

Abb. 3.1 soll diese Anforderung noch einmal visuell verdeutlichen.

Abbildung 3.1: Mindestens gleiche Schnittstelle

4 Architektur und Implementierung der Koordinierungs-Schicht

4.1 Programmierparadigma

Für die Architekturgestaltung als auch die Implementierung habe ich einen objektorientierten Ansatz gewählt, in dem auch einzelne Teile davon der funktionalen Programmierung entsprechen.

Ich bin jemand, der sehr viel Wert auf ein gutes Programmdesign legt und sich auch gut an verschiedenen Entwurfsmustern bedient, dennoch habe ich für diese Arbeit folgende Prinzipien verfolgt:

KISS (keep it simple): „Mache es einfach“, bedeutet, dass es, um ein Problem zu lösen, nicht immer „fünf“verschiedenen ineinander verschachtelten Entwurfsmuster bedarf, sondern das Problem sich auch mit wenigen Zeilen Code lösen lässt.(Degenmann, 2019)

YAGNI (you ain't gonna need it): „Du brauchst es nicht“, bedeutet in der Softwareentwicklung, dass man Dinge nicht programmieren sollte, weil man sie irgendwann einmal brauchen könnte. (Degenmann, 2019) Als Beispiel sei hier genannt, Klassen immer in Interface und Implementierung aufzuteilen. Klar, das könnte in Zukunft Vorteile mit sich bringen, sobald man für das gleiche Interface verschiedene Implementierungen anbieten möchte, aber oftmals ist es dann so, dass man dies so doch nicht benötigen wird und dies eine Wette für die Zukunft ist, deren Investition sich fast nie rechnen wird. Diese Meinung vertritt Professor Dirk Riehle. Der eben beschriebene Sachverhalt wurde so von mir und Herrn Riehle diskutiert.

DRY (Don't Repeat Yourself): „Wiederhole dich nicht“, bedeutet, Code, den ich schon einmal geschrieben habe, nicht nochmal (ähnlich) zu schreiben oder zu kopieren.Degenmann, 2019 Dafür ist es also wieder notwendig, Code den ich einem anderen Codeabschnitt sehr ähnlich wieder so schreiben würde, hier eine Abstraktion zu finden, damit ich dann den Code nur einmal habe, um ihn für

den ursprünglichen Einsatzzweck als auch für den neuen Einsatzzweck verwenden zu können. Das klingt zu dem Prinzip KISS konträr; was es bedingt auch ist. Von daher ist da ein guter Mittelweg zu finden zwischen gutem Design, aber nicht „übersteuertem“Design und dem „Mache es einfac“". Ich schrieb eben „was es bedingt auch ist“, damit meine ich folgendes: Code hundertfach zu kopieren mag zwar schnell und einfach gehen, macht das ganze dann aber sehr unübersichtlich, schwer wartbar, fehleranfällig und schlussendlich doch nicht mehr `KISS` entsprechend.

Mein Programmdesign folgt außerdem in Teilen den Konzepten aus Robert. C Martins „Clean Code“, ihn und dessen Buch möchte ich an dieser Stelle nicht unerwähnt lassen und ich weiterempfehlen. (Martin, 2008)

4.2 Anpassung der GraphQL-Schicht

4.2.1 Einführung

Abschnitt 3.2.4 fordert, dass der GLS mindestens die gleiche Schnittstelle haben soll wie eine normale Instanz. Deshalb bietet es sich an, statt den GLS als neue Software zu schreiben, die bereits für eine normale Instanz mit ihren Schnittstellen existierende um die GLS Funktionalitäten zu erweitern, die dann auf das bereits existierende Ökosystem sowie auf die Schnittstellen zurückgreifen können. Und genau für diesen Weg habe ich mich entschieden.

Um das zu realisieren ist insbesondere eine Anpassung und Erweiterung der GraphQL-Schicht notwendig.

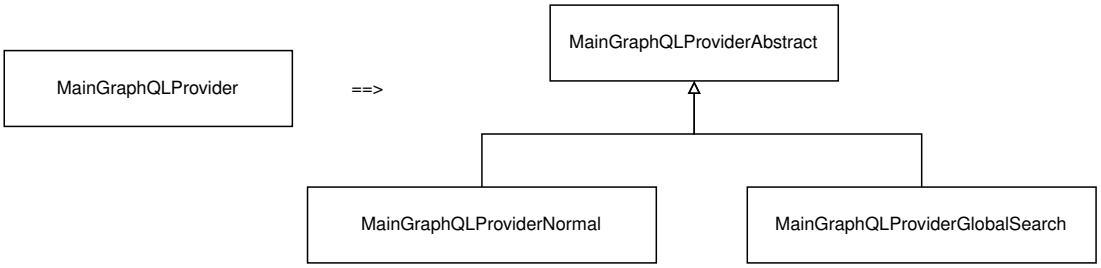

Abbildung 4.1: Änderung des GraphQL Providers

Wie bereits in Abb. 2.8 gezeigt ist der `MainGraphQLProvider` in Abb. 4.1 die Klasse, die mit Hilfe des Java GraphQL-Frameworks das zentrale `GraphQL-Objekt` baut und dazu außerdem die in den implementierenden Klassen von `Abstract-GraphQL` hinterlegten DataFetcher verwendet. Da aber nun für Global Search andere DataFetcher verwendet werden sollen, solche, die eben die Global-Funktionali-

täten (später noch von mir erklärt) in sich kapseln, benötigt es hier eines anderen `Providers`. Deshalb habe ich aus der `MainGraphQLProvider`-Klasse zwei gemacht, deren sich überschneidene Funktionalität und sich überschneidende Schnittstelle ich in `MainGraphQLProviderAbstract` zentralisiere. Die Auswahl, welcher konkrete `Provider` in Abhängigkeit, wie ich die Anwendung starte, verwendet werden soll, übernimmt das in Abschnitt 2.4.2 kurz vorgestellte Dependency-Injection-Framework `Google Guice`. Der konkrete `Provider` könnte sich die nun für ihn relevanten `DataFetcher-Container` (Kindklassen von `AbstractGraphQL`) selbst erstellen, aber gemäß der flexibleren Natur des `Dependency-Inversion-Prinzips` lasse ich auch diese in den konkreten `Provider` injizieren.

Warum aber bleibe ich so dann nicht bei einem Provider und injiziere dort einfach die unterschiedlichen `DataFetcher-Container`? Das hat den Grund, dass der `Container` hier noch weitere Aufgaben übernimmt, denn neben anderen DataFetchern, auf die ich im folgenden Unterkapitel noch einmal etwas näher eingehe, gibt es auch noch weitere Möglichkeiten, das Verhalten des MdH anzupassen, und zwar die Möglichkeiten der `Instrumentierung`, auf die ich dann in den darauf folgenden Unterkapiteln näher eingehe. Diese Möglichkeiten bedürfen einer gesonderten Konfiguration, die ebenso von `MainGraphQLProviderGlobal-Search` übernommen wird. Zudem gibt es im Global Search auch noch zusätzliche DataFetcher, die es im normalen MdH nicht gibt. Diese werden an Felder im GraphQL-Schema geknüpft, die so auch nur im Global Search existieren und über erweiternde Schema-Dateien dem Global Search bekannt gemacht werden. Damit deckt der MdH Funktionalitäten ab, die nur im Kontext des GLS wichtig sind und auf die ich im Kapitel Abschnitt 4.3 noch etwas näher eingehen werde.

4.2.2 Global-Search Datafetcher

Dass das Austauschen der normalen DataFetcher im Global Search einer der zentralen Punkte ist, um den GLS zu realisieren, habe ich bereits im vorhergehenden und im Grundlagenkapitel erklärt. Auch das „Wie“habe ich dort erklärt. In diesem Kapitel soll es jetzt aber darum gehen, mit welchen Möglichkeiten ich so einen DataFetcher genau konstruieren kann. An der Stelle sei noch einmal darauf hingewiesen, dass die DataFetcher als Lambda-Funktionen in einer von `AbstractGraphQL` abgeleiteten `Container`-Klasse (wie z.B `DashboardGraphQL`) definiert sind. Und wenn ich von „Austauschen von DataFecher“ schreibe, ist damit auch das Austauschen dessen `Container`-Klasse gemeint.

Mittels der exemplarischen `DashboardGraphQL`-Klasse zeigt Abb. 4.2 zwei architektonische Möglichkeiten, diesen DataFetcher zu konstruieren. Links in der Abbildung erbt die neue `GlobalSearch`-Klasse alles von der ursprünglichen und der nach wie vor für einen normalen MdH Core eingesetzten `GraphQL`-Klasse. Darin sind einerseits die konkreten DataFetcher enthalten, die dann in der `Global-Search`-Klasse überschrieben werden müssen, andererseits ist darin Code enthal-

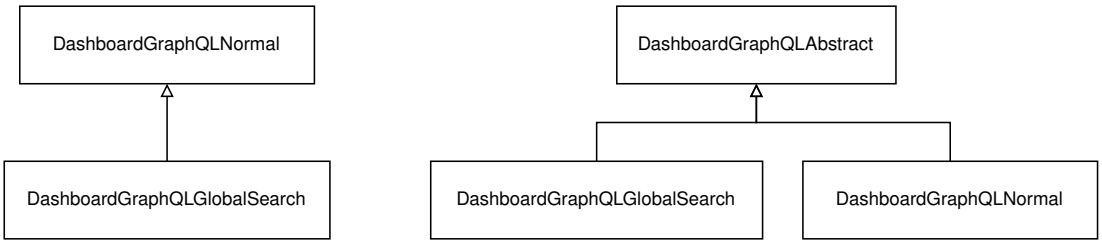

Abbildung 4.2: GLS Datafetcher Architekturmöglichkeiten

ten, der später das `GraphQL-Framework` wissen lässt, welches GraphQL-Schema Feld mit welchem dieser DataFetcher verbunden werden soll. Da ich diese Schnittstelle gemäß der gestellten Anforderungen nicht ändern darf, muss dieser Teil auch in der `GlobalSearch`-Klasse nicht angepasst werden. Es reicht hierbei über die Vererbung aus, die Lambda-Methoden polymorph zu überschreiben. Das Problem mit der Architektur links ist jedoch, dass sich hier leichter Programmierfehler einschleichen können. Beispielsweise füge ich einige Zeit später einen neuen DataFetcher in `DashboardGraphQLNormal` ein und vergesse diesen in der `GlobalSearch`-Klasse zu überschreiben. Dafür bietet sich die Architektur rechts in der Abbildung an. Dabei bleibt der eben erwähnte `Wiring-Code` („Schema zu Datafetcher Verbindungscode“) in der abstrakten Oberklasse und damitzwangsweise auch die abstrakten Schnittstellendefinitionen in dieser, die dann in den jeweiligen Klassen überschrieben werden müssen. Abschließend sei hier noch zu erwähnen, dass der MdH beide Lösungen umsetzt.

Denn es sollen nicht alle DataFetcher überschrieben werden. DataFetcher, die bspw. für das „Neustarten“ des Dockercontainers gedacht sind, sollen hierbei dann später keinen Fernaufruf zur Folge haben, sondern diesen ausschließlich lokal durchführen. Hierbei bleibt das Verhalten einer normalen Instanz und der einer Global-Search-Instanz gleich.

Es gibt außerdem Datafetcher, die (vorerst) im Global-Search grundsätzlich nicht ausgeführt werden sollen. Dazu zählen beispielsweise diese, die den Harvester betreffen. Der eben beschriebene Mechanismus bleibt aber auch für diese gleich, nur dass so die überschriebenen Lambda-Funktionen eine „Nicht-Verfügbar-Exception“ werfen. Dies kann je nach Sichtweise gegen das Liskovsche Substituionsprinzip verstoßen, ist hier aber explizit so gewollt. (Liskov & Wing, 1994) Dazu in der Auswertung der Anforderungen mehr.

4.2.3 GraphQL Instrumentierung

Dass Funktionalität, die es im normalen MdH, aber nicht im GLS geben soll, das kann statt des Überschreibens der entsprechenden DataFetcher auch noch anders gelöst werden. GraphQL Java bietet mit seinem Framework umfangreiche

Möglichkeiten, bereits geladene und auch schon mit den DataFechern verbundene Schemas noch nachträglich anzupassen. So können hierbei beispielsweise GraphQL-Schema-`Directiven` ausgelesen werden und basierend darauf entsprechender Code eingeschoben werden.

Leider sind diese Möglichkeiten nicht alle und auch nicht sehr detailliert seitens GraphQL Java dokumentiert. Durch gezieltes Nachfragen in GithHub-Diskussions und den anschließend sehr freundlichen und sehr detaillierten Antworten von den GraphQL Java Verantwortlichen kam ich diesbezüglich weiter.

Es gibt hierbei mehrere Möglichkeiten den Code zu instrumentieren. Ich habe mich für den `TypeVisitor` entschieden. Dieser Besucher iteriert über alle GraphQL-Typen und deren Felder, die im Schema definiert sind. Währenddessen kann er auf GraphQL-`Directiven` zugreifen, mit denen die einzelnen Typen und Felder annotiert werden können.(„GraphQL Java GitHub“, 2021) Somit hat der `TypeVisitor` nun die Möglichkeit, wenn er bspw. @`onGlobalSerachDeactivated` bei `getHarvestHistory` sieht, dieses Feld zu deaktivieren. Das macht er, indem er den an dieses Feld gekoppelte ursprünglichen DataFetcher durch einen neuen ersetzt. Und zwar einen, der die „Nicht-Verfügbar-Exception“ wirft. Grundsätzlich wäre es hier auch möglich, nur einen DataFetcher dazwischen zu schalten, der den ursprünglichen aufruft und selbst bspw. „Loggingcode“ dazwischen einfügt. Der `Visitor` hat zudem die Möglichkeit, die Beschreibung des `Types` beziehungsweise `Feldes` anzupassen („GraphQL Java GitHub“, 2021) und damit auch die Möglichkeit der Beschreibung direkt hinzufügen, dass diese Methode im Global Search „nicht ausgeführt“ werden kann. Dies ist für etwaige GraphQL-Dokumentationsgeneratoren interessant, die die Dokumentation anhand des `GraphQL-Introspection-Query` erzeugen. Diese Anfrage gibt hauptsächlich das auf dem Server hinterlegte GraphQL-Schema in einer der GraphQL Spezifikation zur Grunde liegenden wohldefinierten Form aus.(Facebook, 2018) Über das Instrumentieren, kann diese aber, wie eben beschrieben, angepasst werden. Ich kann so mit dem MdH, je nach Rolle, ob er als normale Instanz oder als GLS ausgeführt wird, zwei unterschiedliche Dokumentationen generieren lassen.

4.2.4 API Verdoppler

Der `Typevisitor` bietet neben dem Einfügen von eigenen speziellen DataFetchern außerdem auch noch die Möglichkeit, das GraphQL-Schema grundlegend zu ändern. Somit kann er bspw. weitere Typen und Felder hinzufügen, die es so vorher in der Original-Schema-Datei gar nicht gegeben hat.(Facebook, 2018)

Damit dupliziere ich für Abschnitt 3.1.3 jede API-Methode, also jedes Feld in den GraphQL Wurzeln `Query` und `Mutation` und hänge an deren Namen das Postfix `AsListCall` an. Außerdem ändere ich für jede dieser duplizierten Methoden den Rückgabewert zu einer Liste von „ursprünglicher Typ“ ab. `getDashboard` bspw.

gibt als Rückgabe einen Wert vom Typ `Dashboard` zurück. `getDashboardAsListCall` gibt nun als Rückgabe eine Liste zurück, deren Elemente vom Typ `Dashboard` sind.

Jedes dieser Elemente repräsentiert dabei das Ergebnis einer einzelnen Client-Instanz, die der Global Search anfragt. Da diese Methoden wirklich für jede in einer normalen Instanz verfügbaren Methode generiert werden, ist darüber über den Global Search auch Funktionalität verfügbar, die über die regulären Methoden nicht verfügbar ist. Der Instrumentierungscode muss hierfür nur einmal geschrieben werden, eine weitere Abstimmung mit den DataFetchern, die mit den Original-Methoden verbunden sind, ist hier nicht erforderlich.

Der `Instrumentierungs-DataFetcher` nimmt auf dem GLS die GraphQL Anfrage entgegen und modifziert sie leicht. In der Anfrage an den GLS steht in der Anfrage eine GraphQL-Anfrage mit der Methode und dem Postfix. Dieser Postfix entfernt der eben erwähnte DataFetcher und reicht das und alles andere unverändert an die Client-Instanzen weiter. Die Ergebnisse schreibt er nun in eine Liste, die er dann schlussendlich an den anfragenden Client zurück gibt.

Was ist jetzt aber der Unterschied zwischen einer originalen API-Methode zu der der DataFetcher überschrieben werden kann und der gepostfixten? Die API-Methoden mit `AsListCall` haben eine veränderte Schnittstelle und sind somit zu den originalen nicht kompatibel. Das sind sozusagen komplett neue API-Methoden, die nicht eins zu eins ersetzt werden können. Außerdem geben diese die einzelnen Teilergebnisse der jeweiligen Client-Instanzen zurück. Die originalen nicht. Diese geben stattdessen **ein** Ergebnis zurück, sodass der Aufrufer erst einmal gar nicht erkennen kann, ob das Ergebnis nun von verschiedenen Client-Instanzen zusammengesetzt worden ist, oder nur von einer ganz normalen Instanz kommt. Dafür muss der überschriebene DataFetcher aktiv werden und die Teilergebnisse zu einem zusammenführen. Als Beispiel sei hier die Methode `getMetadataTag(name:Megapixels)count` genannt. Diese gibt dann sowohl im Global Search als auch in einer normalen Instanz die (aufsummierte) Anzahl an `Megapixeln` zurück, während die Methode `getMetadataTagAsListCall(count)` die Anzahl der `Megapixels` je Instanz als Listenelemente zurück gibt. Das sind zwei verschiedene Anwendungsfälle.

Wie lässt sich so aber zuordnen, welches Listenelement zu welcher Instanz gehört? Grundsätzlich hätte ich die gepostfixten Methoden auch noch anderweitig gestalten können, ihnen in der Ausgabe bspw. noch weitere Felder hinzufügen können. Dazu hätte ich aber dann auch die Ergebnistypen grundsätzlich duplizieren und die entstandenen Duplikate modifizieren müssen. Das wollte ich aber aus verschiedenen Gründen nicht. Das hätte die Dokumentation unnötig aufgebläht, außerdem müsste ich als anfragender Benutzer auch eine ganz andere Anfrage stellen. So bleiben wenigstens die Anfragen zwischen der originalen Methode und der gepostfixten gleich und es unterscheidet sich so nur im Ergebnis. Um an die

Zuordnung zu kommen, welches Listenelement zu welcher Instanz gehört, dafür habe ich mir einen anderen Mechanismus einfallen lassen.

Diesen Mechanismus beschreibe ich im folgenden Kapitel Abschnitt 4.3. Dort erkläre ich außerdem, wie die überschriebenen DataFetcher, aber auch die gepostfixten überhaupt auf die Client-Instanzen zugreifen können.

4.3 Global Coordinator Framework

Das Global-Coordinator-Framework ist neben der Anpassung der GraphQL Schicht der zentrale Punkt in der Arbeit, um all die Anforderungen umsetzen zu können.

4.3.1 Aufbau

Der Global Search muss wissen, welche Client-Instanzen es gibt und wie deren Zugriffsdaten sind. Ich habe mich hierbei aus Flexibilitätsgründen und aus Gründen, die ich später noch etwas genauer ausführen werde für einen gruppenbasierten Ansatz entschieden. Das heißt, dass der Global Search verschiedene Gruppen haben kann, in denen die Informationen der Client-Instanzen hinterlegt werden können. Dabei kann eine Client-Instanz auch Teil mehrerer Gruppen sein. Eine Gruppe erhält genau wie eine normale Instanz ein Gruppen-User-Zugriffspasswort als auch ein Gruppen-Admin-Zugriffspasswort. Diese Passwörter müssen sowohl hier als auch über alle Gruppen hinweg verschieden sein, denn sie dienen beim Login dafür, zu bestimmen, in welche Gruppe sich der Benutzer einloggt beziehungsweise für welche Gruppe er dann eine API-Anfrage senden möchte. Ein Administrator hinterlegt genau wie für eine normale Instanz die diesbezügliche Konfiguration im JSON-Format in der Konfigurationsdatei. Abb. 4.3 illustriert dieses Design nochmal als UML-Diagramm.

Wie auf diesem zu sehen ist, bietet der `GlobalCoordinator` eine Methode an, die mit dem entsprechenden Passwort die dazugehörige Gruppe zurück gibt. Die Gruppe, zu der ein Name vergeben werden muss, enthält dann die Informationen als Liste zu den Client-Instanzen. Hierbei muss der Name exakt mit dem `Instanz-Namen` der entsprechenden CLient-Instanz übereinstimmen. Über Daten wie `URL`, `Port`, `userPassword` und `adminPassword` hat der GLS nun die Möglichkeit, sich auf dieser einzuloggen und an diese eine GraphQL Anfrage zu senden. Dafür bietet die Klasse `Instance` die Methode `sendGraphQLRequest()` an. Mit dem Flag `asAdmin`, das zur Laufzeit festgelegt werden kann, bestimmt der Methoden-Aufrufer, ob diese API-Methode im Kontext eines `Admins` oder eines `normalen Users` gesendet werden soll. Die eigentliche Anfrage ist in `GraphQL-Bundle`, ein Datentransferobjekt, gekapselt. Darin ist die eigentliche `GraphQL-Query` mit den `GraphQL-Variablen` und weiteren `HTTP-Header-Daten` die darüber übertragen werden können. Das Ergebnis wird als `CompletableFuture`, das

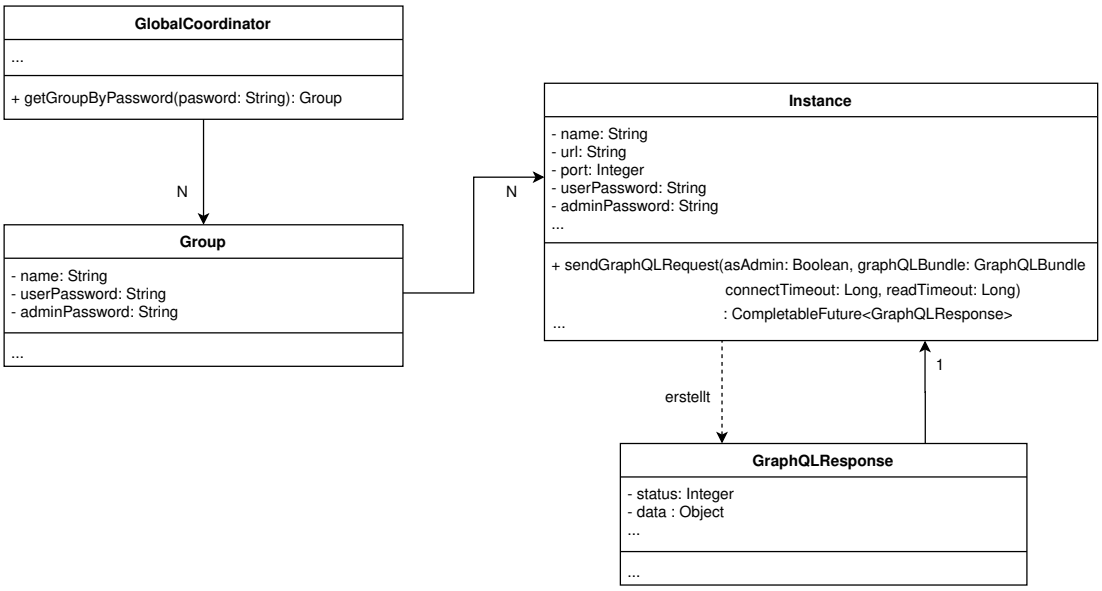

Abbildung 4.3: Übersicht Global Coordinator

ich bereits in Abschnitt 2.5 vorgestellt habe, zurückgegeben. Damit lässt sich das eigentliche Ergebnis nun asynchron holen. Außerdem können so gleich mehrere Anfragen parallel versendet werden, da man bei einer zweiten zu sendenden Anfrage nicht auf das Ergebnis der ersten warten muss.

Ein Objekt der Klasse `GraphQLResponse` stellt das eigentliche Ergebnis dar. Im Erfolgsfall kann ich über entsprechende Methoden über ein `Json-Deserialisierer`, der mir `data` in ein `Java-Objekt` oder eine `Map` transformiert, zugreifen. Im Fehlerfall kann ich den Fehlerstatus auslesen, den ich mir aber wie die Abbildung fälschlicherweise vermuten lässt nicht nur als Integer zurückgeben lassen kann, sondern auch in Textform, die dann genauere Angaben zum Fehler macht.

Folgende Fehlerklassen gibt es:

- **Verbindungsfehler**

  Diese Fehlerklasse beinhaltet Fehler wie `Socket Errors`, `Read Timeouts`, `Write Timeouts` oder sonstige `HTTP Fehler`. Diese Fehler treten meist dann auf, wenn die andere Instanz gerade nicht zur Verfügung steht oder gerade unter Last stehend nicht schnell genug antworten kann.

- **Nicht übereinstimmende Zusammenarbeit der Instanzen**

  Fehler dieser Klasse treten bspw. dann auf, wenn die Version der Client-Instanz nicht mit der des Global Search übereinstimmt. Außerdem treten sie auf, wenn erwartete `Header`-Daten nicht gelesen werden können, weil ein Proxy dazwischen diese heraus gefiltert hat.

- **Authentifikationsfehler**

  Wenn sich der GLS in bei der Client-Instanz nicht einloggen kann, tritt dieser Fehler auf.

- **Sonstige Fehler und GraphQL-Fehler**

  Fehler dieser Klasse sollten normalerweise nicht auftreten, es sei denn es liegt ein Programmierfehler vor, der dazu führt, dass eine Client-Instanz unerwartet einen GraphQL-Fehler zurück gibt, obwohl stattdessen eine gültige Antwort erwartet wird. Auch fallen in diese Fehlerklasse sämtliche weiteren Fehler, die Java als `Exception` fangen kann, aber nicht geworfen werden sollten.

Der Benutzer kann wie bei einer normalen Instanz auch, den GraphQL-Befehl nun an den GraphQL-Endpunkt des Global Search schicken und gibt auch hier das Passwort über das entsprechende Header-Feld an.

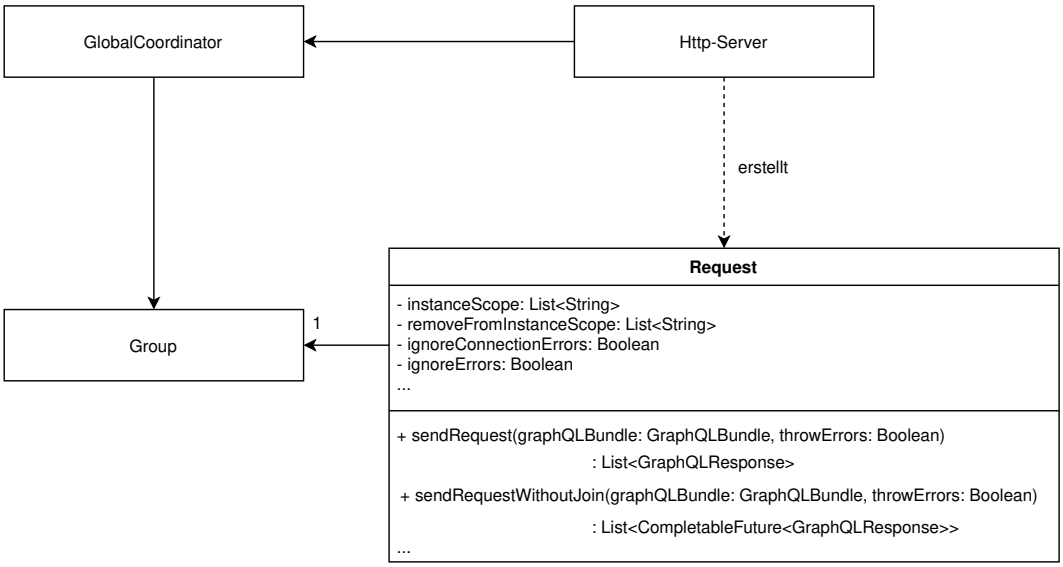

Abbildung 4.4: Anfrageklassendiagramm

In Abb. 4.4 fragt der HTTP-Server schließlich den `GlobalCoordinator` die zu diesem Passwort zugehörige Gruppe an und erstellt damit ein Objekt der Klasse `Request`, das dann diese Gruppe beinhaltet. Dieses Objekt packt der `HTTP-Server` in das `GraphQL-Kontext`-Objekt, sodass dieses jetzt allen DataFetchern zur Verfügung steht. Diese können mit dem `Request`-Objekt gesammelte Anfragen an die in der Gruppe hinterlegten Client-Instanzen schicken. Die Abbildung zeigt nicht alle die über dieses Objekt zur Verfügung stehenden Methoden. Es gibt weitere, mit denen ich auch Anfragen gezielt an eine einzelne Client-Instanz schicken kann. Die beiden angegeben Methoden unterscheiden sich darin, dass die erste eine Liste von den vorhin vorgestellten `GraphQLReponse`-Objekten zurück

gibt und zwar so, dass die Ergebnisse schon alle vorliegen, während die zweite nicht darauf wartet, bis alle Ergebnisse vorhanden sind, sondern stattdessen eine Liste von `CompletableFutures` zurückgibt. Je nach späteren Anwendungsfall profitiere ich davon, die Ergebnisse bereits asynchron, also bspw. mit weiteren Stufen weiterverarbeiten zu können, da sie bis zu einem bestimmten Punkt nicht voneinander abhängen, oder ich profitiere nicht davon, da ich direkt alle Ergebnis benötige, um diese dann miteinander zu verrechnen.

Das `throw-Errors`-Flag jeweils in den angegeben Methoden signalisiert, ob sich die Methode an möglicherweise auf `TRUE` gesetzte Objektvariablen bei *ignoreConnectionErrors* und `ignoreErrors` halten soll, oder nicht. Sofern das `Flag` auf `FALSE` gesetzt ist, gibt es ganz normal die Liste mit den `GraphQL-Response` oder in der `Future`-Form zurück, andernfalls wirft sie oder später das `Future` eine Exception falls der `GraphQLResponse-Status` keinen Erfolgsfall anzeigt, wobei auch nur dann, wenn die eben erwähnten Objektvariablen je nach Fehlerfall auf `TRUE` gesetzt sind. Neben dem Werfen der Exceptions machen die Methoden der `Request`-Klasse hier noch etwas mehr und zwar unabhängig von den `Flag`-Variablen. Sie loggen den Vorgang mit, auch in welcher Reihenfolge die Ergebnisse dann schlussendlich vorliegen und speichern dies intern in `Request` ab. Dieser Logspeicher wird später dann vom `HTTP`-Server ausgelesen und im `HTTP-Response-Header` zurück an den Aufrufer übertragen.

Und genau um diesen Seitenkanal `HTTP-Header` möchte ich im nächsten Unterkapitel etwas genauer beschreiben. Da ich die GraphQL-Schnittstellen nicht ändern möchte und darf, also von den gepostfixten `asListCall`-Methoden abgesehen, und aber selbst da möchte ich mich relativ nahe am original bewegen, muss ich relevante Zusatzdaten an den Globa Search anderweitig übertragen. Und hierbei habe ich mich für den `HTTP-Header` entschieden, sowohl was die Anfrage vom Benutzer an den Global Search angeht, als auch die Antwort von letzterem.

Mit dem Gruppenmechanismus kann ein Administrator zur Konfigurationszeit schon einmal statisch mehrere verschiedene Sichten auf die Client-Instanz-Welt festlegen. Das ist aber nicht so flexibel, wie es in den Anforderungen gefordert ist. Deshalb soll die Menge an CLient-Instanzen, die dem Benutzer über das Gruppenpasswort zur Verfügung steht von diesem dynamisch noch weiter eingeschränkt werden können.

4.3.2 Seitenkanal HTTP-Header

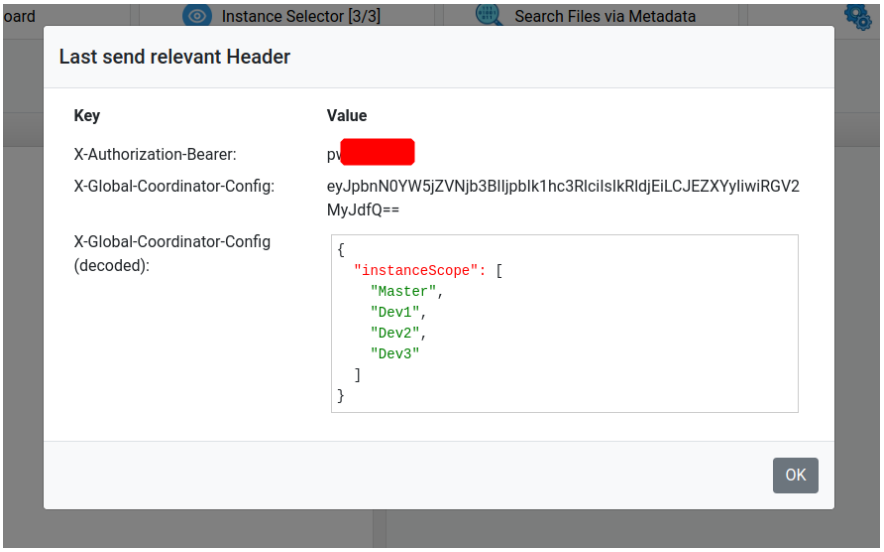

Abbildung 4.5: HTTP-Header-Felder *Benutzer -> GLS* Anfrage

Abb. 4.5 zeigt eine Visualisierung der zuletzt gesendeten Header-Daten der GraphiQL-Konsole, die wir in unsere WebUI integriert haben. Das Plugin für die HTTP-Header-Visualisierung habe ich geschrieben. Zu erkennen ist darauf das Header-Feld, über das das Passwort gesendet wird und eines über das diese für den Global Search relevanten zusätzlichen Daten gesendet werden. Diese müssen `Base64 enkodiert` übertragen werden. `Dekodiert` liegen die Daten im Json-Format vor.

Dieses Json-Objekt kann kann vier weitere Json-Objekte aufnehmen, und zwar:

- **instanceScope**

  Dieses Attribut kann eine Liste mit Instanz-Namen enthalten, oder null sein. Sofern dieses Attribut gesetzt ist, schränkt der GLS die verfügbare Menge für diese Anfrage auf die dort angegebenen Instanzen ein. Auch kann hier die Global-Search-Instanz selbst mit angegeben werden. Auch wenn diese Instanz selbst keine Metadaten abgespeichert hat, die sie im Fall, dass sie hier angegeben wird oder nicht, ausliefern könnte oder nicht, ist das in einem anderen Kontext relevant. Und zwar bei den abzuspeichernden Daten in der Suchmaske. Denn diese kann der GLS selbst auch abspeichern und wieder ausliefern. Dazu später aber mehr.

- **removeFromInstanceScope**

  Dieses Attribut funktioniert analog zu eben beschriebenem, nur anders herum. Damit lassen sich Instanzen aus der Gruppe entfernen, die man in dieser Anfrage nicht dabei haben möchte.

- **ignoreConnectionErrors**

  Ist dieses Attribut auf `TRUE` gesetzt, ignoriert der GLS mögliche Verbindungsprobleme mit den Client-Instanzen. Es wird wie vorhin schon beschrieben keine Exception geworfen, der DataFetcher bekommt in dem Fall dann auch im Verbindungsfehlerfall das Ergebnis bei einem `sendRequest()` zurück. Darin enthalten sind aber auch die fehlerhaften `GraphQLResponses`. Der Methodenaufrufer muss in dem Fall die Antworten herausfiltern, deren Status nicht dem „Erfolgreich-Status“ entsprechen.

  Das `Request`-Obekt loggt mögliche Fehler mit.

- **ignoreErrors**

  Dieses Attribut funktioniert genauso wie `ingoreConnectionErrors`, nur dass dieses grundsätzlich alle Fehler zulässt. Hier führen Client-Instanzen, die einen `Authentifikationsfehler`, genau wie Instanzen, die beispielsweise aufgrund eines Programmierfehlers einen `GraphQL-Error` zurückgeben, nicht zum Abbruch.

Diese JSON kodierte Daten im Header überträgt der HTTP-Server in die Objektvariablen des `Request`-Objekts. Siehe Abb. 4.4.

Abbildung 4.6: HTTP-Header-Felder als Ergebnis einer GLS Anfrage

Abb. 4.6 zeigt das dekodierte Header-Feld in einer Antwort auf eine an den GLS gesendete Anfrage. Auch darin ist ein JSON-Objekt enthalten. Dieses wiederum enthält Attribute, die den `instanceScope` aufgeteilt in `selectedInstanceList` und `globalCoordinatorSelfInScope` wieder zurück gibt. Interessanter sind hier aber noch die beiden anderen Felder. Das `ordered...` Feld gibt die Instanzen zurück, die der Global Search erfolgreich hat erreichen können und deren Ergebnisse keinen Fehlerstatus ergeben haben. Außerdem kodiert dieses Listenfeld die Reihenfolge, wie die Ergebnisse sortiert sind. Diese Reihenfolge ist für bspw. eine

aufsummierte `getMetadataTag`-Anfrage irrelevant, für die gepostfixte Variante, also `getMetadataTagAsListCall` aber sehr wichtig. Da die GraphQL Antwort an sich die Instanz Namen nicht mit zurück gibt, hat der Aufrufer über die Daten im Header die Möglichkeit zu den über die gepostfixte-Methoden zurückgegebene Liste, Instanzen zu deren Elemente zuzuordnen.

`instanceExecutionResultList` gibt allgemein Informationen darüber zurück, ob eine bestimmte Instanz ein valides Ergebnis zurückgeliefert hat. Das kann sowohl für Anfragen bei denen Fehler unterdrückt werden, als auch für Anfragen bei denen Fehler nicht unterdrückt werden, man aber erfahren möchte, weshalb der komplette Aufruf gescheitert ist, von Bedeutung sein.

4.3.3 Instance Selector in der WebUI

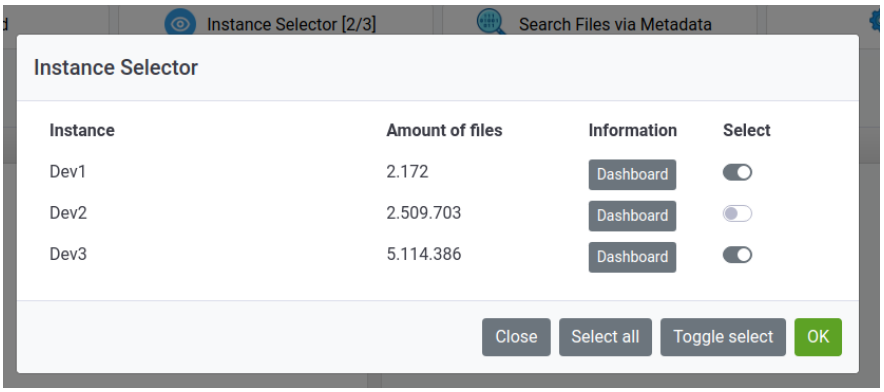

Abbildung 4.7: Instance Selector

Über den `Instance Selector` (siehe Abb. 4.7) kann er Instanzen mit in den `instanceScope` nehmen oder sie aus diesem entfernen. Über den Dashboard-Button kann er sich das Dashboard einer einzelnen Instanz direkt in der GLS-WebUI anzeigen lassen. Dort stehen ihm auch die Statistiken dieser Instanz zur Verfügung. Ist eine Client-Instanz nicht erreichbar, zeigt ihm der `Instance Selector` statt des Dashboard-Buttons einen `Info`-Button an. Darüber erhält der Benutzer Informationen, weshalb diese Instanz gerade nicht Verfügbar ist, z.B. *weil der GLS gerade keine Verbindung zu dieser aufbauen kann.*

Sobald sich, während der User den Global Search über die WebUI bedient, beispielsweise gerade eine Suche vorbereitet, die Verfügbarkeit der Client-Instanzen verändern sollte, dann informiert die WebUI den Benutzer darüber, dass sich die verfügbare Menge an Client-Instanzen geändert hat, außerdem deaktiviert die WebUI die nicht verfügbaren Client-Instanzen im `Instance Selector`. Würde dies die WebUI nicht tun, dann würde der Benutzer beim Suchen einen Fehlermeldung zurückbekommen, da die `ignoreError`-Flags von der WebUI nicht

verwendet werden. Die Informationen darüber, welche Instanzen gerade verfügbar sind und welche nicht, erhält die WebUI über eine GraphQL-Query-Methode, die ich explizit für diese Statusabfrage dem Global Search bezüglich GraphQL erweiternd hinzugefügt habe. Ich habe dafür keine anderen Methoden abgeändert. Auch könnte ich an diese Information über ein `getDashboardAsListCall` gelangen, zusätzlich mit den in den Headern übertragenen Daten, aber da dies meiner Ansicht nach eine relevante und häufig benötigte Geschäftsfunktionalität ist, habe ich für diesen Anwendungsfall eine eigene GraphQL-Methode hinzugefügt.

4.3.4 Beispielaufruf

In diesem Unterkapitel gehe ich nun eine Abfrage exemplarisch durch.

Ein Benutzer sendet eine GraphQL-Query mit `getDashboard...` an den Global Search. Dabei verwendet er das Passwort *xyz* und definiert den `instanceScope` mit *dev1*, *dev2*, *dev5* und *dev7*. Auf die `ingoreError`-Flags verzichtet er. Der HTTP-Server nimmt die Anfrage entgegen und fragt den `GlobalCoordinator`, zu welcher Gruppe das Passwort xyz gehört. Dieser gibt die Gruppe *Picture-Group* zurück, die insgesamt 15 Client-Instanzen hat. Diese Gruppe packt der HTTP-Server nun zusammen mit der `instanceScope`-Information in ein von ihm erstelltes `Request`-Objekt und dieses wiederum in das `GraphQL-Kontext`-Objekt. Schließlich gelangt dieses `Kontext`-Objekt in den vom Global Search bereitgestellten und überschreibenden DataFetcher `getDashboard()`. Dieser DataFetcher hat nun Zugriff auf das `Request`-Objekt und kann darüber Anfragen an die Client-Instanzen schicken. Und das macht er über die Methode `sendGraphQLRequest(...)`, die das `Request`-Objekt bereitstellt. Dieses Objekt prüft jetzt vorab mittels einer anderen Anfrage die Erreichbarkeit der Client-Instanzen, an welche die eigentliche Anfrage geschickt werden soll. Anschließend sendet das `Request`-Objekt die eigentliche Anfrage parallel an alle Instanzen weiter, die in `instanceScope` definiert sind, also an *dev1*, *dev2*, *dev5* und *dev7*. Sobald diese Methode die Ergebnisse aller Instanzen zurückerhalten hat, gibt sie diese zurück an den DataFetcher. Dieser kann diese Daten jetzt entsprechend verarbeiten und eine Antwort an den Benutzer zurückgeben. Diese sollte im Erfolgsfall ein GraphQL-Typ von `Dashboard` enthalten, außerdem stehen im Header wie vorhin gezeigt weitere Angaben zur Ausführung. Wie der überschreibende DataFetcher `getDashboard` diese Daten genau verarbeitet und wie es die anderen überschreibenden DataeFetcher machen, erkläre ich in Kapitel 5.

4.3.5 Konsistenzgarantien

Dieses Kapitel prüft die Konsistenzgarantien, die mein Framework hat.

Solange die `ignoreError`-Flags nicht gesetzt sind, verfolgt der Global Search eine *Alles oder Nichts* Strategie. Das heißt, wenn der GLS in einer Gruppe von drei Client-Instanzen, eine Anfrage an Instanz *eins* sendet und diese dort ausgeführt wird, dann soll sie auch in den Instanzen *zwei* und *drei* ausgeführt werden. Gilt das auch für die Zustands ändernde GraphQL-Methoden in `Mutation`, dann ist das System stark konsistent. (Singh, 2019) Ganz erreiche ich dieses ideal aber (bewusst) nicht, wie ich im folgenden näher ausführe.

Bevor der GLS die eigentliche Anfrage an die Client-Instanzen schickt, prüft er erst einmal vorab, ob diese alle überhaupt erreichbar sind. Ist das nicht der Fall, dann bricht der GLS schon vorher ab und gibt einen Fehler an den Benutzer zurück. Sind alle Client-Instanzen verfügbar, dann nehme ich an, dass sie das auch noch sehr wahrscheinlich wenige Millisekunden bis Sekunden später sind. Die eigentliche Anfrage sollte dann so auf allen Instanzen ausgeführt werden, dem vorausgesetzt, dass nicht eine währenddessen abstürzt. Auch in den eben erwähnten Millisekunden könnte sich die Verfügbarkeit einer Instanz ändern. Und das ist der Grund weshalb der GLS doch nicht stark konsistent arbeitet.

Das ist bei nicht zustandsändernden Anfragen nicht weiter schlimm. Bei diesen Anfragen kann es vorkommen, dass schlimmstenfalls ein Teil der Instanzen die Anfrage unnötig berechnen, da sie dann letztendlich verworfen wird, weil eine der Instanzen in der Zwischenzeit nicht mehr verfügbar geworden ist, dem zur Folge der GLS einen Fehler an den Benutzer zurückmeldet. Problematischer ist das schon bei Anfragen, die den Zustand ändern. Hierbei kann es passieren, dass der Zustand auf Instanz *eins* und *zwei* geändert wurde, aber auf Instanz *drei* nicht, weil letztere nach der Verbindungskontrolle vom Netzwerk getrennt worden ist. In dem Fall würde der Benutzer das aber über das Log-Protokoll, das über den Header zurück an den ihn gesendet wird, bemerken. Dieser muss sich dann aber selbst darum kümmern, dass das System aus seiner Sicht wieder in einen konsistenten Zustand kommt.

Der Global Search bietet mit der vorherigen Verbindungskontrolle etwas mehr Konsistenz als ein trivial programmiertes System, das darauf verzichtet. Für ein stark konsistentes System müsste der GLS beispielsweise auf Paxos (Lamport, 1998) oder Raft (Ongaro & Ousterhout, 2014) zurückgreifen, das aber nicht gefordert wurde. Der Einsatz dieser Protokolle würde den MdH Performance kosten bei bislang eher unkritische GraphQL-`Mutation` Befehle.

5 Architektur und Implementierung der Global-Search-Funktionen

5.1 Allgemeine Einführung

Dieses Kapitel beschreibt den Aufbau der wichtigsten überschreibenden DataFetcher des Global Search. Dazu gehören der `Dashboard`-, `MetadataInfo`, `QueryStorage`- und `Search`-DataFetcher.

All diese DataFetcher erhalten wie in den Kapiteln zuvor als auch in den Grundlagenkapiteln beschrieben ein Objekt, das `DataFetchingEnvironment`-Objekt `DFE` übergeben, worüber sie auf den GraphQL-Query zugreifen können. Damit können sie die Parameter abfragen und die Felder sehen, an denen der Client interessiert ist. Außerdem bekommen sie vom `DFE`-Objekt das zuvor vom HTTP-Server gesetzte `Kontext`-Objekt. In diesem steckt - wird der MdH als GLS ausgeführt - das `Request`-Objekt, über das sie die Anfragen an die Client-Instanzen schicken können. Darin außerdem enthalten ist auch der GraphQL-Query sowie dessen zugehörige optionale Variablen in normaler Textform.

Leider bietet GraphQL Java keine Möglichkeit, aus den Daten, die unter anderem in einer `Map` in `DFE` vorliegen, ein (bzw. den ursprünglichen) GraphQL-Query zu generieren, wie ihn der Benutzer an den Server gesendet hat. Diese Einschränkung umgehe ich damit, dass ich den `Query` mit in das `Kontext`-Objekt packe. Jedoch umgehe ich diese Einschränkung nur zum Teil. Denn ich kann auf diese Art die Daten in `DFE` nicht anpassen und mir dann für die angepassten Daten einen neuen GraphQL-Query generieren lassen. Dafür habe ich mir selbst einen kleinen *DFE zu GraphQL-Query-Konverter* programmiert, der aber natürlich nur einen kleinen Sprachumfang GraphQLs beherrscht und damit nur für eine kleine Menge der in `DFE` enthaltenen Daten ein GraphQL-Query erzeugen kann.

Zwar benutzt der Global Search nun größtenteils einfach die GraphQL-API, welche die Client-Instanzen ohnehin auch für Dritte anbieten, aber nicht immer reicht das aus, oder der eben erwähnte Konverter deckt den benötigten Funktionsumfang nicht ab. Um diese Einschränkungen zu umgehen, habe ich in diesem

Möglichkeit geschaffen, dass der Global Search an GraphQL vorbei über einen Art Seitenkanal auch noch Informationen übertragen kann. Dafür verwende ich auch HTTP-Felder.

In den folgenden Kapiteln meine ich mit *angefragte Client-Instanzen* bzw. *ausgewählten Client-Instanzen*, die Client-Instanzen, die das `Request`-Objekt für seine `sendRequest()`-Methoden vorhält. Das sind also die Client-Instanzen, die sich in der mit dem `Request`-Objekt verknüpften Gruppe befinden und außerdem dem über den HTTP-Header übertragenen `instanceScope` entsprechen.

5.2 Dashboard

Der GLS-Datafetcher soll gemäß der Anforderung in Abschnitt 3.1.4 die Dashboard-Daten der ausgewählten Client-Instanzen anfragen und zu einem Ausgabe Dashboard zusammenfügen.

Da dieses Dashboard schnell geladen werden können soll, muss ich die Funktionalität gegenüber eines Dashboard einer normalen Instanz etwas einschränken.

Ich verzichte beim GLS Dashboard auf die Angaben `different Filetypes` und `different Metadata tags`. Denn diese kann ich nicht aus den Einzelwerten der jeweiligen Teilergebnisse der Client Instanzen Dashboards berechnen. Um sie doch berechnen zu können, müsste ich von jeder ausgewählten Client-Instanz deren Filetypenliste und Metadaten-Tags Liste anfordern, was aber nicht mehr so performant wäre, wie gefordert. Das heißt ich gebe für diese beiden Felder `NULL` zurück.

Das GraphQL Feld zum `Treewalkstate` ist vom Typ *String*. Das heißt ich kann dort hinein völlig freien Text schreiben. Ich habe mich entschieden, hier beispielsweise `Running [3/5]` anzugeben, falls drei der fünf ausgewählten Client-Instanzen gerade einen Treewalk ausführen, ansonsten steht da nur `All ready`.

Die GraphQL Felder der zuletzt ausgeführten `Treewalks` und des nächst-auszuführenden `Treewalks` setze ich auf den Wert, den die Feldnamen implizieren, nur dass ich beispielsweise den zuletzt ausgeführten Treewalk aus der Menge aller ausgewählten Instanzen bestimme.

Angaben wie `Instance name`, `User Role`, `Installation date` und `last reboot` sind Angaben die den GLS selbst betreffen und nicht aus den Daten berechnet werden, die von den Client-Instanzen kommen.

Abb. 5.1 zeigt als Beispiel das Dashboard eines Global Search

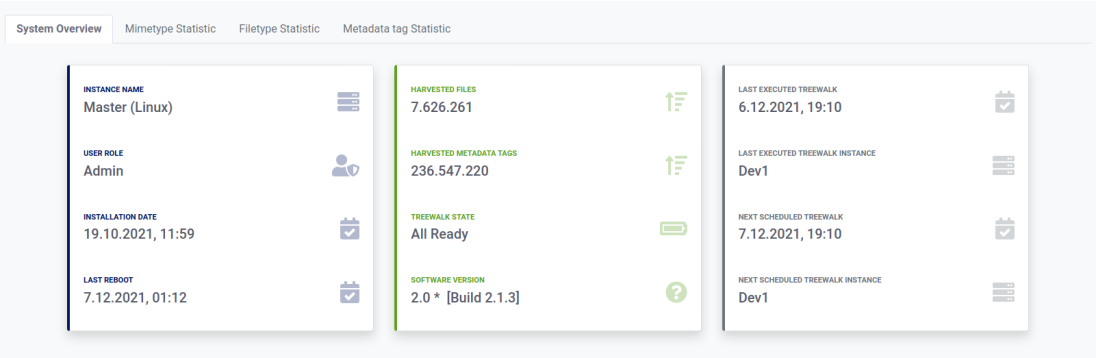

Abbildung 5.1: Global Search Dashboard

5.2.1 Umsetzung

Der Dashboard DataFetcher sendet den über das Kontext-Objekt erhaltenen Graphql-Query unverändert über das `Request`-Objekt an die ausgewählten Client-Instanzen und erhält das JSON-Ergebnis zurück. Die erhaltenen Werte führt der DataFetcher wie im letzten Unterkapitel ausgeführt in ein neues Ausgabeobjekt zusammen bzw. fügt in dieses dann noch die Informationen der Global Search Instanz selbst ein. Schließlich gibt der DataFatcher das Ausgabeobjekt zurück und der Benutzer des GLS erhält seinerseits ein einziges Dashboard-Objekt in JSON-Form.

Für die Umsetzung habe ich zudem das GraphQL-Schema bezüglich des Dashboards nur für den Global Search erweitert. Hier sind die Felder `lastExecutedTreewalkInstanceName` und `nextScheduledTreewalkInstanceName` dazugekommen, um die oben erwähnten zusätzlichen Informationen zurückgeben zu können. Weiterhin habe ich sowohl für den normalen MdH als auch für den GLS die *Nicht-NULL-Bedingungen* bei `differentFiletypes` und `differentMetadataTags` zurückgenommen, damit diese nun auch NUlL-Werte transportieren dürfen.

5.2.2 Auswertung

Das Global Search Dashboard summiert verschiedene Daten von den Dashboards der Client Instanzen auf, an anderer Stelle nimmt er die größten bzw. kleinsten Werte, wie z.B bei `lastExecutedTreewalk` und für die GraphQL-Felder, die nur den GLS betreffen können, nimmt er die Werte von sich selbst. Insgesamt führt der GLS die Dashboardinformationen sinnvoll zusammen, auch wenn er aus Performance-Gründen zwei Werte auf NULL setzt.

Damit gilt die Anforderung *Dashboard* aus Abschnitt 3.1.4 als erfüllt.

5.3 Metadata-Info

Der GraphQL-DataFetcher-Container `MetadataInfoGraphQL` vereint die überschreibenden DataFetcher in sich, die für die Metadaten der Metadaten da sind. Dazu gehören die sechs GraphQL Wurzelfelder `getMetadataTag(s)`, `getFileType(s)` und `getMimeType(s)`. Diese sind wie aus den Grundlagenkapiteln bekannt vorhanden für die `Statistiken`, den `Metadatatag Explorer`, Felder in der Suche, deren Eingabetyp auf Metadadaten-Tag-Typen abgestimmt werden kann.

5.3.1 Umsetzung

Die Umsetzung von `getMetadataTag`, also der Methode, die als Ergebnis nur ein Element zurück gibt, analog bei `getFileType` und `getMimetype`, ist trivial.

Auch hier reicht der entsprechende DataFetcher den vom Aufrufer über das `Kontext`-Objekt erhaltenen und unveränderten GraphQL-Query an die ausgewählten Client-Instanzen weiter. Die Ergebnisse kann er ohne größeren Schwierigkeiten zusammenführen, den Namen des Metadaten-Tags beibehalten, die Zähler aufsummieren und an den Aufrufer ein zusammengeführtes Objekt zurückgeben.

Die Implementierung der Methoden, die statt eines einzelnen Elements, eine sortierte Liste zurück geben, die über Limit auch noch begrenzt werden können, ist schon nicht mehr so trivial.

Anfangs nahm ich irrtümlich an, dass es doch ausreiche, auch hier den den GraphQL-Query vom Benutzer einfach an die Client-Instanzen weiterzuleiten, was so aber nicht funktioniert.

`getMetadataTags(limit:10)` liefert bei einer normalen Instanz die zehn Metadaten-Tags zurück, die am häufigsten vorkommen, also nach dem Zähler sortiert. Darunter ist beispielsweise bei der Client-Instanz *eins* der Metadaten-Tag *Megapixels*, bei Instanz *zwei* ist dieser Wert aber erst auf Position 257. Und soll nun auch der GLS die zehn meisten Metadaten-Tags über die Instanzen *eins* und *zwei* zurück liefern, dann muss er von diesen Client-Instanzen folglich alle Metadaten-Tags anfragen, um dann eine Aussage darüber treffen zu können, was global gesehen die *TOP 10* Metadaten-Tags sind.

Das bedeutet nun, dass der GLS hier die Benutzeranfrage modifizieren muss, er muss aus dieser die Parameter `limit` und `offset` streichen und diese neue Anfrage dann an die Client-Instanzen senden. Die Ergebnislisten die er zurückbekommt, muss er alle zu einer gesamten Liste *ListeGesamt* zusammenführen und diese dann erneut nach Häufigkeit sortieren. Schließlich erstellt er eine weitere Liste, indem er von der *ListeGesamt* gemäß den Parametern `limit` und `offset` vom originalen `Query` die entsprechenden Elemente in diese packt. Diese Liste gibt er dann an den Aufrufer zurück.

Auch wenn es nicht ganz so performant ist, sich von jeder Client-Instanz deren vollständigen Metadaten-Tags geben zu lassen, macht der GLS hier an dieser Stelle etwas, was die schlechte Performance zumindest in Teilen wieder etwas relativiert. Er führt die ganzen Operationen parallel aus. Die Anfragen an die Client-Instanzen werden schon vom `Global Coordinator Framework` parallel gestellt, die nicht *gejointe* Liste dann aber vom `getMetadataTag`-DataFetcher parallel weiterverarbeitet. Das heißt, sobald der DataFetcher von zwei Client-Instanzen Ergebnisse hat, beginnt er sofort diese zusammenzuführen, sobald ein drittes Ergebnis dazu kommt wird auch dieses direkt in die eben zusammengeführte Liste hinein verwoben. Weiterhin, sobald beispielsweise gleichzeitig acht Teilergebnisse zur Verfügung stehen (weil die Instanzen ihre Ergebnisse ziemlich zeitgleich ausliefern), werden diese nicht nur so nach und nach zusammengeführt, sondern immer in „zweier Paketen“ parallel. Also *eins* und *zwei*, *drei* und *vier*, *fünf* und *sechs*, *sieben* und *acht*. Daraus resultieren dann vier weitere Listen, die analog wie eben beschrieben weiter zusammengeführt werden, bis am Ende jede Client-Instanz seine Ergebnisse abgeliefert hat und auch alle Ergebnisse in die *ListeGesamt* zusammengeführt worden sind.

Hierbei verwende ich von den `CompletableFutures` deren Stufenmechanismus und deren `anyOf`- Klassenmethoden. Damit bekomme ich von den `Completable-Futures` das zurück, das als erstes fertig wird, und dieses kann ich dann mit einem weiteren Aufruf von `anyOf` und einer weiteren Rückgabe eines anderen `CompletableFuture` direkt zusammenführen. Das Ergebnis packe ich dann wieder in ein neues `Completable-Future` und stelle dies dem Pool der zusammenführenden `CompletableFutures` zur Verfügung. Sobald darin dann nur noch eines ist, bin ich fertig.

Diese Liste vom letzten `CompletableFuture` speichert der DataFetcher für 60 Minuten zwischen, um erneute Anfragen, von diesem Cache aus beantworten zu können. Aus dieser (zwischengespeicherten) *ListeGesamt* entnimmt, wie oben beschrieben, der DataFetcher gemäß `limit` und `offset` die entsprechenden Elemente und gibt diese an den Aufrufer zurück.

Die DataFetcher `getFileTypes` und `getMimetypes` funktionieren analog.

5.3.2 Auswertung

Ich habe die DataFetcher-Schnittstellen in `MetadataInfo` nicht einschränken müssen. Dass hierbei aber mitunter die vollständigen Daten von den Client-Instanzen angefordert werden müssen, obwohl der Benutzer nur eine Teilmenge dieser Daten angefragt hat, ist zwar nicht ganz so performant, wird aber durch paralleles Verarbeiten und Zwischenspeichern des Endergebnisses aus dieser Perspektive wieder etwas relativiert. Insgesamt ist die Funktionalität so gegeben, wie in der Anforderung vorgesehen.

Und damit gilt die Anforderung *Metadata-Info* aus Abschnitt 3.1.5 als erfüllt.

5.4 Query-Storage

5.4.1 Einführung und Besonderheiten

Der `Query-Storage` ist kein Begriff aus GraphQL sondern ist Bestandteil des MdH. In diesem kann ein Benutzer der WebUI, seine in der Suchmaske getätigten Eingaben abspeichern und später wieder laden. Dazu stellt der MdH über das GraphQL-Schema mehrere Methoden zur Verfügung, deren überschreibenden DataFetcher Gegenstand dieses Kapitels sind.

Der Benutzer soll sich über das `Query-Storage` `Load-Modal` alle abgespeicherten Eingabemasken aller ausgewählten Client-Instanzen laden können, aber diese auch aus dem GLS heraus löschen dürfen.

Ein Sonderfall stellt das Abspeichern dar. Wenn der Benutzer eine Suchanfrage-Eingabe abspeichern möchte, dann wird diese im Global Search selbst abgespeichert und im `Load-Modal` zusätzlich angezeigt.

Über den `Instance-Selector` kann der Benutzer die Menge ausgewählter Client-Instanzen verändern und damit auch die möglichen zu ladenden abgespeicherten Einträge, die ich im folgenden als `Queries` bezeichnen werde.

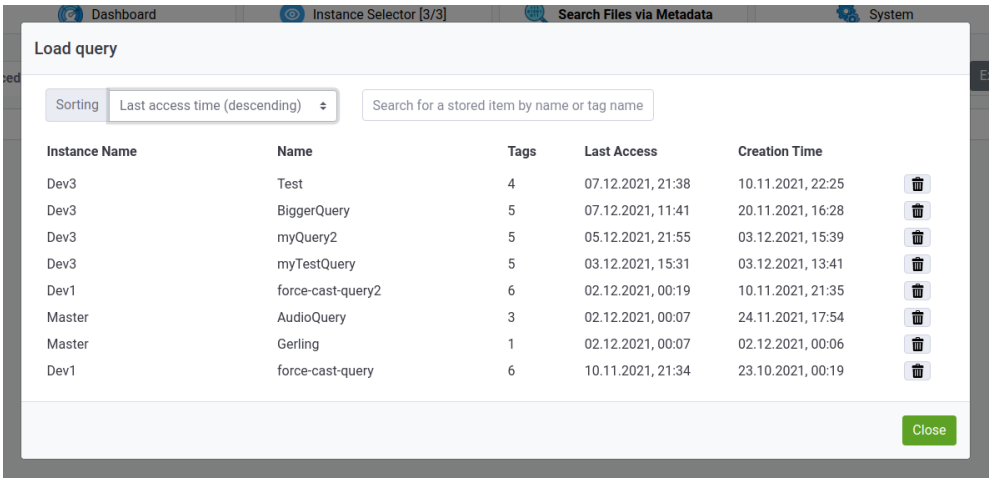

Abbildung 5.2: Query-Storage Load-Modal

Als Beispiel zeigt Abb. 5.2 das `Load-Modal` eines GLS. `Master` ist hierbei die Global-Search-Instanz selbst, während *Dev1-3* dessen ausgewählten Client-Instanzen repräsentieren.

5.4.2 Umsetzung

Den DataFetcher, der für das Abspeichern der `Queries` zuständig ist, habe ich nicht überschrieben. Anfragen die diesen DataFetcher erreichen, können genauso abgearbeitet werden, wie wenn sie einen DataFetcher von einer normalen Instanz erreichen. Er speichert den Inhalt dann in seiner eigenen Datenbank ab.

Beim Laden eines bestimmten `Queries` gibt der Benutzer eine `ID` des zu ladenden `Query` an. Diese war ursprünglich genau die `ID`, welche die Datenbank mittels eines *Autoincrement*-Mechanismus beim Abspeichern selbst bestimmt hat. Diese `IDs` sind aber über alle Client-Instanzen, die der GLS haben kann nicht einzigartig. Um diese Einzigartigkeit zu bekommen, habe ich sie aber nicht durch einen *Universally Unique Identifier (UUID)* ersetzt, welche dies garantieren würde, sondern die ID in der Datenbank unverändert gelassen. Stattdessen fügt der Server dieser `ID` beim Ausliefern seinen Instanznamen und ein Trennzeichen hinzu. Das hat nun den Vorteil, dass der GLS direkt bestimmen kann, an welche seiner ausgewählten Client-Instanzen er den `Load-Request` weitersenden muss, um an die abgespeicherte `Query` zu gelangen. Im Falle des `UUID` müsste er alle Client-Instanzen fragen, welche davon ihm den `Query` mit der vom Benutzer angefragten `UUID` hat. Sobald eine Anfrage mit einer ID ankommt, die den GLS selbst adressiert, übergibt er diese an den DataFetcher, den er selbst überschrieben hat mittels eines `super()`-Aufrufs, um die angefragte `Query` selbst aus seiner Datenbank zu laden. Analog funktioniert der `Delete`-DataFetcher

Die DataFetcher, die nicht nur einen Eintrag zurückgeben, sondern eine Liste von abgespeicherten `Queries`, funktionieren ähnlich zu denen, die ich in Abschnitt 5.3 beschrieben habe. Sie laden alle `Queries` und das `limit` und `offset` wird dann nachträglich auf die gesamte Menge angewandt. Weitergehende Optimierungen sind hier aber nicht vorgesehen, da nicht zu erwartet ist, dass hier ähnlich viele Daten wie bei `getMetadataTags` zu verarbeiten sind.

5.4.3 Auswertung

Der GLS stellt sämtliche Funktionalität zur Verfügung, die hierzu gefordert ist. Er kann `Queries` in seiner Instanz selbst abspeichern und wieder laden und löschen, sowie er `Queries` auch von seinen ausgewählten Client-Instanzen laden und löschen kann.

Damit gilt die Anforderung *Query-Storage* aus Abschnitt 3.1.6 als erfüllt.

5.5 Search

Für den Search, die zentrale Funktion auf der Ausgabeseite der Metadatatags gibt es im normal ausgeführten MdH nur einen DataFetcher und so für den GLS

auch nur einen zu überschreibenden DataFetcher.

Über diesen, der sehr viele Parameter hat und sehr eine große Auswahlmöglichkeit an Ausgabefeldern, kann der Benutzer die Suche vollständig steuern.

Wie der Global Search diese Suche auf seine Client-Instanzen aufteilt, welche Randfälle dabei zu beachten sind und auch wie ich eine Suche nach den Instanz-Namen selbst ermögliche (siehe Abb. 5.3), die so als Metadaten-Tag nicht in der Datenbank stehen, erkläre ich im folgenden Unterkapitel.

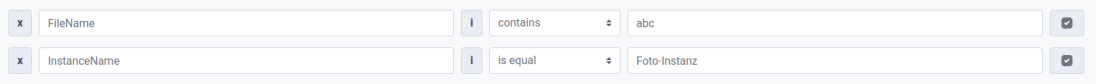

Abbildung 5.3: Beispielsuche mit InstanceName

5.5.1 Umsetzung

Zuerst gehe ich näher auf den sogennanten virtuellen Metadaten-Tag `InstanceName` ein. Diesen fügt der der MdH automatisch zu jeder Metadaten-Tag Ergebnismenge hinzu und sendet sie zurück an den Benutzer, sofern dieser nach `InstanceName` fragt. Da dieser `Tag` nicht in der Datenbank existiert, wird dieser auf Server Ebene hinzugefügt und zwar so, dass alle Regeln die für die Metadaten-Tags in der Datenbank gelten auch hier gelten. Das heißt, möchte der Benutzer nach diesem `Tag` suchen oder nach Textbestandteilen in diesem `Tag`, gewährleistet dies der Server. Damit kann er diesen `Tag` nicht nur als technischen `Tag` in der Ausgabe verwenden, sondern diesen auch als Bestandteil der Metadaten-Tag Domäne sehen, sofern er diese Client-Instanz als Teil seiner Domäne sieht. Es sei angenommen, dass dem so ist und er die Instanz `Foto-Instanz` als Bestandteil seiner Domäne sieht. Somit kann er eine Abfrage über den GLS senden, die wie folgt lautet: *FileName CONTAINS abc AND InstanceName EQUALS Foto-Instanz*. Damit sucht er dann nach Dateinamen die `abc` enthalten und irgendetwas mit Fotos zu tun haben, da dies sein Domänenmodel `Foto-Instanz` impliziert.

Baut der Benutzer seine MdH-Landschaft nicht nach derartigen Regeln, und sieht seine Unterteilung der Metadaten auf verschiedene MdH Instancen eher mehr aus einer technischen Perspektive, so reicht oftmals auch das Aktivieren bzw. Deaktivieren auszuwählender Instanzen über den `instanceScope` bzw. in der WebUI über den `Instance Selector` aus, den er für die gleiche Abfrage unterschiedlich konfigurieren kann.

Wenn der Benutzer eine Suchanfrage an den GLS `mdhSearch`-DataFetcher sendet, dann macht dieser nun folgendes: Nehmen wir an, dass es eine Suchanfrage ist, in der nach `Megapixels` sortiert werden soll und außerdem ein `offset` von *12* und `limit` von *18* eingestellt ist.

Anders als bei `MetadataInfo` muss der GLS hier jetzt nicht alle Daten von den Client-Instanzen anfordern, aber einfach nur den `Query` vom Benutzer an diese weiterzuleiten funktioniert auch hier nicht. Den `Offset` muss er weglassen und mit dem `Limit` muss er 12 + 18 = 30 Elemente von jeder ausgewählten Client-Instanz anfragen. An der Stelle sei als Beispiel angenommen, dass die Elemente von der Client-Instanz *eins* diejenigen sind, die gemäß der Sortierreihenfolge vor denen der anderen Client-Instanzen liegen. Das heißt, der GLS überspringt die ersten 12 Einträge von dieser Instanz und gibt die nächsten 18 an den Benutzer zurück. Die angefragten Elemente der anderen Client-Instanzen würde er so verwerfen. Für diesen Sonderfall, den der GLs aber vorab nicht wissen kann, wäre es ihm möglich tatsächlich an die Client-Instanz *eins* genau `Offset` *12* und `Limit` *18* zu schicken.

Im Normalfall ist es aber so, dass die Sortierung ergibt, dass der GLS ein paar Einträge von Client-Instanz *eins*, ein paar von Client-Instanz *zwei* usw. mit in seine Ausgabemenge der Größe 18 übernehmen muss. Und das funktioniert wie folgt: Der GLS hat nun alle 30 Einträge jeder ausgewählten Client-Instanz erhalten. Diese sind von den jeweiligen Instanzen bereits vor-sortiert, das heißt jede Instanz bezogene Liste mit den Elementen ist bereits richtig sortiert. Der GLS schaut sich nun alle Listenkopfelemente der jeweiligen Instanzen an und nimmt sich dann das, was der Sortierung entsprechend von diesen am weitesten oben steht und packt es in seine Ausgabeliste. Bei der Instanz, von der er eben das Kopfelement entnommen hat, rückt das nächste Element nach und ist dessen neues Kopfelement. Und jetzt beginnt der Vorgang von vorne. Der GLS sucht wieder das Element, das am weitesten oben steht und so weiter und so fort, bis er 18 Elemente entnommen hat.

Dieser Vorgang lässt sich leider in keiner Hinsicht parallelisieren. Der GLS kann mit diesem Vorgang erst beginnen, wenn er von jeder ausgewählten Instanz deren vor-sortierte Listen bekommen hat. Andererseits sei hier nochmal zu erwähnen, dass diese Listen maximal `offset`+`limit` groß sein müssen und keinesfalls wie bei `MetadataInfo` alle Listenelemente beinhalten müssen. Den Prozess den der GLS an der Stelle machen muss, habe ich etwas vereinfacht dargestellt. Dieser Sortiervorgang lässt sich zumindest algorithmisch noch etwas optimieren. So verwende ich hier eine `TreeMap` in welche die Kopfelemente, beziehungsweise deren kompletten Listen anhand ihrer Kopfelemente anfangs einmal einsortiert werden. Sobald ein Element entnommen wurde, die Listenreferenz in Folge dessen aus der `TreeMap` entfernt worden ist und das nächste Kopfelement nachgerückt ist wird diese Liste mit seinem neuen Kopfelement erneut in die `TreeMap` einsortiert. Im Gegensatz zu dem vorhin beschriebenen trivialen Ansatz, dessen Aufwand bei O(N) liegt, ist der Aufwand nun nur noch bei O(log(n)), da die Einsortierung über eine TreeMap einer binären Suche gleichkommt. (Júnior, 2013)

Statt eine `Query` ohne `offset` und verändertem `limit` an die Client-Instanzen zu

schicken, sende ich an diese den original `Query`, da sich das Zusammenbauen einer derartigen Query über die `DataFetchingEnvironment`-Daten einfach zu komplex wäre, da der Query an sich komplex sein kann. Deshalb habe ich mich dafür entschieden, hier diese Daten über einen Seitenkanal über den Header zu übertragen, der dann von den normalen Instanzen erkannt wird und in ihren DataFetchern zugreifbar über das `Kontext`-Objekt vor den `limit`- und `offset`-Angaben im GraphQL-Query (kodiert in deren `DataFetchingEnvironment`-Objekten) bevorzugt wird.

Eine Warnung bezüglich der Benutzung sei hier noch zu nennen. Große `Offset`- und kleine `Limit`-Werte bedeuten wie gezeigt nicht, dass nur der GLS von jeder ausgewählten Client-Instanz nur eine kleine Teilmenge laden muss, sondern er muss `offset` + `limit` viele Elemente laden. Dementsprechend sind große `Offset`-Werte zu vermeiden. In der WebUI ist der `Pagination`-Button, mit dem der Benutzer auf die letzte Seite springen können soll und somit ein sehr großes `Offset` erzeugen würde, deaktiviert. Stattdessen erscheint der Hinweis, so der Benutzer an der letzten Seite interessiert ist, einfach die Sortierung um-zudrehen. Darüber kann er dann auch auf die „letzte Seite“ springen, aber ohne großen Offset und vor allem so, dass der Server hier nicht Millionen von Einträgen von verschiedenen ausgewählten Client-Instanzen laden und sortieren muss.

Damit der Benutzer in der WebUI erkennen kann, welcher Eintrag zu welcher Client-Instanz gehört, aktiviert die WebUI automatisch den Metadaten-Tag `InstanceName` zur Ausgabe, sofern die WebUI im Kontext eines GLS läuft. Diese Information teilt ihr der Server beim erstmaligen Laden der *Single-Page-Application* direkt mit. Darüber in Kenntnis gesetzt kann sie nun auch statt des `Harvest`-Reiters den `Instance Selector` anzeigen. Die WebUI bedarf ansonsten keinen größeren weiteren Änderungen, da sich die Schnittstellen grundsätzlich, und damit auch die Schnittstellen zum `Search` nicht geändert haben. Sie muss nur die Daten im `InstanceSelector` bei jedem API-Aufruf mitsenden, damit der GLS dem Wunsch ausgewählter Client-Instanzen entsprechen kann.

5.5.2 Auswertung

Die Suche über alle Instanzen ist möglich. Dabei kann er sowohl bei einer normalen als auch einer GLS Instanz nach Instanz-Namen (als Teil eines Domänenmodels) suchen. Aus einer eher technischen Sicht oder eher dynamisch anwendbaren Domänenmodels kann er diese Suche auch über den `instanceScope` (in der WebUI über den `Instance Selector`) beeinflussen. In der WebUI wird der Instanz Name automatisch mit in die Ergebnistabelle aufgenommen.

Damit gilt die Anforderung *Search* aus Abschnitt 3.1.7 als erfüllt.

6 Auswertung

6.1 Funktionale Anforderungen

6.1.1 Auswahlmöglichkeit der Clients

Der Administrator übergibt die Informationen, welche Client-Instanzen der GLS verwenden und wie er darauf zugreifen kann über ein der normalen MdH-Instanz ähnlichen Konfigurationsdatei. Darüber hinausgehend, kann er diese hier in Gruppen einteilen, wobei die Gruppe und damit auch die verfügbaren Client-Instanzen zur Laufzeit mittels des Gruppenpassworts bestimmt wird, auf deren Grundlage der GLS die Sicht auf die ihm zur Verfügung stehenden Instanzen hat.

Diese Sicht kann der Benutzer auch noch während der Laufzeit für jede einzelne Anfrage über bestimmte Variablen im HTTP-Header abändern, hierbei angeben, welche Instanzen der GLS aus der ausgewählten Gruppe verwenden soll, oder welche er aus dieser entfernen soll. Die daraus resultiernde Menge von Client-Instanzen habe ich in Kapitel 5 auch als *ausgewählte Client-Instanzen* bezeichnet. Ein Teil von Abschnitt 4.3.1 beschreibt, wie ich diese Anforderung umgesetzt habe.

Die Anforderung *Auswahlmöglichkeit der Clients* aus Abschnitt 3.1.1 gilt somit als erfüllt.

6.1.2 Client Instanzen in der WebUI

Ein Benutzer der WebUI, muss sich in dieser einloggen. Über das eingegebene Passwort bestimmt der GLS die Gruppe und damit eine Vorauswahl dem Benutzer möglicher zur Verfügung stehender Instanzen. Über den in der WebUI nun verfügbaren `Instance Selector` hat der Benutzer die Möglichkeit, sich Informationen über die eben erwähnten Instanzen anzeigen zu lassen. Dazu gehören auch deren Dashboard Daten und somit auch deren Statistiken. Weiterhin zeigt ihm der `Instance Selector` im Fehlerfall, die Fehlerinformationen an. Über die *HTML-Select-Boxen* kann der Benutzer Instanzen, mit denen der GLS arbeiten soll, aktivieren und deaktivieren.

Damit ist die Anforderung *Client Instanzen in der WebUI* aus Abschnitt 3.1.2 erfüllt.

6.1.3 Einfache Proxy-Aufrufe für alle GraphQL-Schnittstellen

Der GLS stellt für jede originale Methode eine weitere zur Verfügung. Diese versieht er mit dem Postfix `asListCall`. Und diese machen genau das, was in der Anforderung steht. Diese Methode gibt die Einzelergebnisse der ausgewählten Client-Instanzen unverarbeitet in einer Liste zurück. Damit der Client die Anfrage, dazu gehören die Parameter und die GraphQL-Auswahlliste der Felder, die zurückgegeben werden muss, nicht anders als bei der originalen Methode aufbauen muss, bleibt zumindest dieser Teil schematisch in GraphQL gleich. Das hat aber auch zur Folge, dass der Aufrufer, sofern die Original-Methode keine Möglichkeit bietet, den Instanz-Namen über die Auswahlliste anzufordern, diesen so nicht bekommen kann. Dafür bietet mein Global-Search-Coordinator-Framework aber einen Ersatz. Im Header wird die Zuordnung der Listenelemente zu den Instanzen mit übertragen (siehe Abb. 4.6) . Die entsprechende Umsetzung findet sich in Abschnitt 4.2.4

Damit ist die Anforderung *Einfache Proxy-Aufrufe für alle GraphQL-Schnittstellen* aus Abschnitt 6.1.3 erfüllt.

6.1.4 Dashboard

Die Auswertung zu dieser Anforderung (Abschnitt 3.1.4) habe ich bereits in Abschnitt 5.2.2 geschrieben und gilt als erfüllt

6.1.5 Metadata-Info

Die Auswertung zu dieser Anforderung (Abschnitt 3.1.5) habe ich bereits in Abschnitt 5.3.2 geschrieben und gilt als erfüllt

6.1.6 Query-Storage

Die Auswertung zu dieser Anforderung (Abschnitt 3.1.6) habe ich bereits in Abschnitt 5.4.3 geschrieben und gilt als erfüllt

6.1.7 Search

Die Auswertung zu dieser Anforderung (Abschnitt 3.1.7) habe ich bereits in Abschnitt 5.5.2 geschrieben und gilt als erfüllt

6.2 Nicht-funktionale Anforderungen

6.2.1 GraphQL für Kommunikation

Der Global Search bietet genau wie eine normale Instanz auch eine GraphQL Schnittstelle, die unter Berücksichtigung der Anforderung Abschnitt 3.2.4 sogar mindestens den gleichen Umfang haben muss wie die einer normalen Instanz. Auch die Kommunikation zwischen GLS und seinen ausgewählten Client-Instanzen basiert auf GraphQL.

Sowohl für die Kommunikation von *GLS zu Benutzer*, als auch *GLS zu Client Instanz*, findet ein Teil der Kommunikation auch noch über einen Seitenkanal statt, über HTTP-Header. Über diesen werden zusätzliche Informationen ausgetauscht. Das wäre auch direkt über GraphQL möglich gewesen, auch unter Einhaltung der eben genannten Anforderung, hätte aber zum Nachteil gehabt, dass die Abwärtskompatibilität nicht mehr gegeben wäre. Das heißt, eine speziell für den GLS geschriebene Software, die von diesen Zusatzinformationen (wie z.b von `removeFromScope`, aus Abschnitt 4.3.2) Gebrauch macht, könnte auf GraphQL-Ebene nicht mehr so einfach mit einer normalen Client Instanz direkt kommunizieren.

Die Anforderung *GraphQL für Kommunikation* aus Abschnitt 3.2.1 verlangt aber nur, dass die grundlegende Kommunikation auf GraphQL basieren soll. Dies ist auch mit den Seitenkanälen über HTTP-Header Felder gegeben und damit ist die Anforderung erfüllt.

6.2.2 Fehlerbehandlung

Treten bei der Kommunikation zwischen dem GLS und einer Client-Instanz Fehler (siehe Abschnitt 4.3.1, Fehlerklassen) auf, dann gibt der Global Search diese an den Benutzer weiter, wie in Abschnitt 4.3.2) gezeigt.

Die außerdem in dieser Anforderung verlangten Konsistenzgarantien habe ich in Abschnitt 4.3.5 beschrieben. Eine möglichst starke Konsistenz wird erreicht, wenn auch nicht eine vollständige. Das war aber auch nicht gefordert. Der Global Search prüft dazu vorab, ob er die ausgewählten Client-Instanzen erreichen kann, und sendet erst dann die eigentliche Anfrage.

Dieses Verhalten kann der Benutzer über den Seitenkanal HTTP-Header abschwächen, indem er die zwei vorhandenen *Error-Flags* setzt. In dem Fall bricht der GLS die Anfrage nicht ab, sondern berechnet das Ergebnis anhand der funktionierenden beziehungsweise erreichbaren anderen ausgewählten Client-Instanzen.

Damit gilt die Anforderung *Fehlerbehandlung* aus Abschnitt 3.2.2 als erfüllt.

6.2.3 Performance

Über das Global-Search-Coordinator-Framework sendet der GLS die Anfragen grundsätzlich parallel und empfängt diese auch parallel. Ob die DataFetcher diese schließlich auch parallel weiterverarbeiten können, hängt von deren Verarbeitungsprozesse ab. Bei `MetadataInfo` beispielsweise können immer zwei Ergebnisse parallel weiterverarbeitet werden, bis am Ende der Zusammenführung nur noch eines übrig bleibt. Andere DataFetcher können die Ergebnisse erst dann weiterverarbeiten, sobald sie alle Teilergebnisse der Client-Instanzen haben, so z.B. der `Search`. Denn um die Teilergebnisse in ein Endergebnis sortiert überführen zu können, müssen diese für die Sortierung alle vorhanden sein.

Somit wägt die Umsetzung gut ab, wann sie Parallelisierung verwenden kann und wann nicht. Die Anforderung *Performance* aus Abschnitt 3.2.3 gilt dem zur Folge als erfüllt.

6.2.4 Mindestens gleiche Schnittstelle

Der Global Search bietet mindestens die selbe Schnittstelle an wie eine normale Client Instanz. Das erreicht er dadurch, indem er keine neue eigene Software ist, sondern auf das Ökosystem „Client-Instanz“ aufsetzt und dessen Schnittstellen mitverwendet. Um dann die notwendigen GLS-Funktionalitäten bereitzustellen, überschreibt er die jeweiligen für die Schnittstellen relevanten DataFetcher. Außerdem erweitert der GLS die Schnittstellen, um so über die HTTP-Header-Felder dem Benutzer zu erlauben, Zusatzinformationen zu übertragen. Damit kann der Benutzer die Fehlertoleranz bestimmen und außerdem die möglichen Client-Instanzen beschränken.

Drittsoftware muss damit also nicht angepasst werden, da sie mindestens die gleiche Schnittstellen vorfindet wie bei einer normalen Client Instanz auch. Sie kann aber angepasst beziehungsweise viel mehr erweitert werden, und zwar um beispielsweise die Funktionalität die ich eben erwähnt habe wie z.B die Client-Instanz Menge pro Anfrage festzulegen. Dass diese Anforderung diesbezüglich gilt, zeigt auch die WebUI. Diese hätte nicht angepasst werden müssen und hätte damit genauso weiter funktioniert. Aber um diese Zusatzfunktionalität auch in der WebUI zugänglich zu machen, waren doch ein paar wenige Anpassungen notwendig.

Der GLS stellt aber nicht für jede Schnittstelle eine funktionierende Implementierung zur Verfügung. Die `Harvest`-Methoden beispielsweise überschreibt der GLS zwar, deaktiviert sie dann sogleich aber indem er in diesen überschreibenden Methoden eine *Nicht-Verfügbarkeits-Exception* wirft. Stellt der GLS damit etwa doch nicht *mindestens die gleichen Schnittstellen* zur Verfügung? Diese Frage ist nicht ganz trivial zu beantworten. Es kommt hier auf die Perspektive an. Ist es so, dass

der Aufrufer grundsätzlich erwarten kann, dass jegliche API-Methoden Exceptions werfen dürfen? Dann gilt die Anforderung diesbezüglich als erfüllt. Ist es aber so, dass die Methoden nur eine vorher festgelegte Menge an Exceptions werfen dürfen. Und ist diese Menge auf eine normale Client-Instanz vorher festgelegt, dann wäre eine *Nicht-Verfügbarkeits-Exception* ein Verstoß gegen das Liskovsche Substitutionsprinzip, da es hier die Nachbedingungen aufweicht, also mehr Exceptions zulässt als in einer normalen Client-Instanz erlaubt ist. Da es aber eine solche Festlegung bei einer normalen Client-Instanz nicht gibt und der Aufrufer immer darauf gefasst sein muss, dass ein Fehler irgend einer Art zurückkommen kann (auch bei einer normalen Client-Instanz), stellt das hier keine Verletzung der Anforderung *Mindestens gleiche Schnittstelle* dar. Analog ist das mit den Fehler zu sehen, die der GLS daraus resultierend zurück gibt, wenn Client-Instanzen nicht Verfügbar sind, oder hierbei ein anderer Fehler auftritt.(Liskov & Wing, 1994)

Problematischer ist aber noch etwas anderes. Ich habe beim Dashboard das GraphQL Schema dahingehend angepasst dass bei `diffMetadatatags` und `diffFile-`

`types` NULL zurückgegeben werden darf. Dies habe ich deshalb getan, dass der GLS hier NULL zurückgeben kann. Nur leider hatten diese beiden Felder zuvor eine *NOT-NULL-Bedingung* (Das Ausrufezeichen in GraphQL). Hier habe ich also ganz klar in die Schnittstelle einer normalen Client-Instanz eingegriffen. Eine Software die bisher angenommen hat, dass diese zwei Felder keine NULL-Werte zurück liefern und jetzt doch NULL-Werte bekommen, funktionieren nicht mehr richtig.

Gerade des letzten von mir geschriebenen Abschnitts wegen gilt die Anforderung *Mindestens gleiche Schnittstelle* aus Abschnitt 3.2.4 als nur teilweise erfüllt.

7 Fazit

7.1 Rückblick und Ergebnisse

Mit dem MdH Global Search steht dem Benutzer jetzt ein mächtiges Werkzeug zur Verfügung, mit dem er gesammelt auf viele einzelne Client-Instanzen, die zum eigenen Unternehmen gehören, als auch bei anderen Unternehmen bereitgestellt sind, zugreifen kann. Somit kann er beispielsweise über all diese Instanzen gleichzeitig suchen, deren Metadaten-Statistiken sich aufsummiert zurückgeben lassen, als auch grundsätzlich auf jegliche Funktionalität dieser Instanzen über den zentralen GLS zugreifen.

Die WebUI ist bis auf kleine auf den GLS zugeschnittene Funktionen die gleiche wie bei einer normalen Instanz und auch bereits geschriebene Software, welche die GraphQL-API einer normalen Instanz verwendet, kann ohne weitere Anpassung mit der API vom GLS kommunizieren. Das gilt bis auf eine kleine Ausnahme im Dashboard, bei der es im Zuge der GLS Umsetzung zu einer kleinen Anpassung der GraphQL-API gekommen ist, die auch eine normale Instanz betrifft.

Zur Umsetzung habe ich die GraphQL-Schicht der bisherigen MdH Software anpassen müssen. Dort habe ich es ermöglicht, DataFetcher, in denen die Programmlogik zu den jeweiligen GraphQL-Methoden enthalten ist, mit den für den GLS spezifischen DataFetchern austauschen zu können. Mit der Implementierung des Global-Search-Coordinator steht den DataFetchern ein umfangreiches Framework zur Verfügung, über das sie Anfragen an ausgewählte Client-Instanzen schicken können. Darüber hinaus bietet dieses Framework eine umfangreiche Fehlerbehandlung an, verstärkte Konsistenzgarantien, falls es vorkommen sollte, dass eine der ausgewählten Client-Instanzen nicht so funktioniert, wie sie funktionieren sollte. Außerdem sind in diesem einige Nebenläufigkeitsmechanismen verbaut, wie z.B. dass Anfragen parallel abgeschickt und empfangen werden können. Auch intelligente Algorithmen und der Einsatz weiterer Parallelisierungstechniken in den DataFetchern selbst runden die Umsetzung des GLS gut ab.

Alle gestellten Anforderungen habe ich erfüllt, bis auf eine, die ich nur teilweise erfüllt habe. Dabei handelt es sich um die vorhin erwähnte Ausnahme im Dash-

board. Hierbei wird eine einmalige Anpassung bereits auf den MdH abgestimmte Software notwendig sein und danach funktioniert alles wie gewünscht, dass der Benutzer es gar nicht mehr merken muss, ob er jetzt mit einer normalen MdH Instanz arbeitet oder mit dem GLS. Er kann beim Global Search dann natürlich noch die spezfischen Zusatzfunktionalitäten verwende, die zu ihm gehören.

Dennoch, der GLS wirkt nun so wie eine normale MdH-Instanz, nur dass der Benutzer darüber dann auf ein Vielfaches der Daten und deren Metadaten im Vergleich zu einer normalen Client-Instanz Zugriff hat. Es wirkt damit so, als wären die mit dem GLS verbundene ausgewählte Client-Instanzen aus einem Guss. Der Global Search ist somit eine sehr gut gelungene weitere Softwarekomponente des MdH.

7.2 Ausblick

Wie in Abb. 1.1 gezeigt gibt es mehrere Möglichkeiten, den MdH zu skalieren. Das, was dort auf der linken Hälfte dargestellt ist, war der GLS und somit Teil dieser Masterarbeit. Die rechte Seite zeigt wie ein MdH mittels mehreren Harvestern weiter verstärkt werden kann. Dies wäre ein nächster Schritt den MdH noch weiter zu verbessern.

Die Global-Search-Implementierung setzt auf einige Nebenläufigkeitstechniken auf, diesen performanter zu machen. Eine detaillierte Evaluation zwecks der Performance steht aber noch aus. Dies könnte bspw. dadurch erreicht werden, indem man beispielsweise 100 Instanzen in einer Cloud-Umgebung auf verschiedenen physikalischen Rechnern startet und diese dann mit dem GLS verknüpft und Messungen durchführt. Dabei sollte die Anzahl der Instanzen und der jeweilige Datenbankinhalt variiert werden, um hinterher eine genaue Aussage darüber treffen zu können, ab wann es sinnvoll ist bei einer physikalischen Maschine X keine weiteren Daten mehr zu *harvesten* sondern weitere Maschinen gleichen Typs dazu zunehmen, um dadurch dann bezüglich der Suche Geschwindigkeitsvorteile zu bekommen und den GLS diesbezüglich möglicherweise weiter optimieren zu können

Eine weitere Verbesserung des GLS wäre, die bislang noch als *Nicht-Verfügbar-Exception* werfende überschreibende Methoden auszuprogrammieren, um auch in dieser Hinsicht einen GLS zu bekommen, dessen Schnittstelle so verwendet werden kann, wie wenn es die Schnittstelle einer normalen Client-Instanz ist und es der Benutzer auch bei den `Harvest`-Methoden die Möglichkeit zu bieten, diese über den GLS zu steuern als wäre es nur eine Instanz, statt viele, die der GLS im Hintergrund vereint. Das ganze hätte dann auch zum Vorteil, dass ein Benutzer, egal ob für die Suche oder für die Steuerung des Harvesters nur noch mit einer Instanz arbeiten müsste, nämlich der GLS-Instanz. Ob dies aus der Mar-

ketingsicht auch so gewollt ist, stellt sich in einer anderen Frage. Hier kann es durchaus so sein, dass man davon ausgeht, dass der Kunde den MdH voluminöser wahrnimmt, wenn er später neben dem GLS auch noch weiterhin mit den einzelnen Client-Instanzen interagieren kann, vielmehr für den Harvester sogar muss.

Als letzter Punkt in *Ausblick* wäre noch folgendes zu nennen. Ein Global-Search kann bislang ausschließlich normale Instanzen als Client-Instanzen verwenden. Eine Weiterentwicklung wäre, wenn ein GLS auch andere GLS-Instanzen als Client-Instanzen verwenden kann. Damit wäre unter anderem eine weitere performancetechnische Optimierung möglich. Außerdem wären hier auch auf logischer Ebene weitere Anwendungsfälle möglich. So müsste eine Abteilung oder ein Unternehmen im Falle einer Kollaboration seinen GLS nicht direkt mit den Client-Instanzen des anderen Unternehmen verbinden, sondern könnte seinen GLS mit deren GLS verbinden, was dies aus administrativer Sicht vereinfacht.

Literaturverzeichnis

baeldung. (2021). *Guide To CompletableFuture*. Verfügbar 12. Dezember 2021 unter https://www.baeldung.com/java-completablefuture

Byron, L. (2015). *GraphQL: A data query language*. Verfügbar 12. Dezember 2021 unter https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/

Class CompletableFuture<T>. (n. d.). Verfügbar 12. Dezember 2021 unter https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html

Degenmann, K. (2019). *YAGNI-, DRY- und KISS-Prinzip: Die Zauberformeln der Programmierung!* Verfügbar 12. Dezember 2021 unter https://entwickler.de/agile/yagni-dry-und-kiss-prinzip-die-zauberformeln-der-programmierung/

Dependency Injection vs Factory Pattern. (2009). Verfügbar 12. Dezember 2021 unter https://stackoverflow.com/a/557790/2182302

Facebook, I. (2018). *GraphQL Specification*. Verfügbar 12. Dezember 2021 unter https://spec.graphql.org/June2018/

Foundation, T. G. (2021). *GraphQL*. Verfügbar 12. Dezember 2021 unter https://graphql.org/

Fowler, M. (2004). *Inversion of Control Containers and the Dependency Injection pattern*. Verfügbar 12. Dezember 2021 unter https://martinfowler.com/articles/injection.html

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (2015). *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. MITP-Verlags GmbH & Co. KG. https://books.google.de/books?id=lYyXBgAAQBAJ

Gopal, K. (2019). *IO vs CPU operations*. Verfügbar 12. Dezember 2021 unter https://jkl.gg/b/io-cpu-bound-threads/

GraphiQL. (n. d.). Verfügbar 12. Dezember 2021 unter https://github.com/graphql/graphiql

GraphQL is the better REST. (2021). Verfügbar 12. Dezember 2021 unter https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

GraphQL Java GitHub. (2021). Verfügbar 12. Dezember 2021 unter https://github.com/graphql-java/graphql-java

GRAU-DATA. (2021). *Grobgranulare Architektur des MdH*. Verfügbar 12. Dezember 2021 unter https://metadatahub.de/documents/static/mdh/animation.gif

Guice. (2021). Verfügbar 12. Dezember 2021 unter https://github.com/google/guice

Júnior, A. (2013). *treemap vs arraylist: perfomance and resources while iterating/adding/editing values*. Verfügbar 12. Dezember 2021 unter https://stackoverflow.com/a/16725503/2182302

Kress, D. (2020). *GraphQL: Eine Einführung in APIs mit GraphQL*. dpunkt.verlag. https://books.google.de/books?id=QPsGEAAAQBAJ

Lamport, L. (1998). The Part-Time Parliament. *ACM Trans. Comput. Syst.*, *16*(2), 133–169. http://dblp.uni-trier.de/db/journals/tocs/tocs16.html#Lamport98

Liskov, B. H. & Wing, J. M. (1994). A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.*, *16*(6), 1811–1841. https://doi.org/10.1145/197320.197383

Marek, A. (2021). *GraphQL Java*. Verfügbar 12. Dezember 2021 unter https://www.graphql-java.com/

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson.

Ongaro, D. & Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, 305–320.

Singh, V. K. (2019). *Eventual Consistency vs Strong Consistency*. Verfügbar 12. Dezember 2021 unter https://medium.com/system-design-blog/eventual-consistency-vs-strong-consistency-b4de1f92534d