OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

HSP-Studienarbeit

Erstellung eines Datentransformations- und Verteilungssystems für Software-as-a-Service-Anwendungen

eingereicht von:	Stephan Nunhofer Matrikelnummer: 3247646 Studiengang: Master Informatik OTH Regensburg
betreut durch:	Prof. Dr. Johannes Schildgen OTH Regensburg

Kallmünz, den 3. August 2020

Inhaltsverzeichnis

1 Aktuelle Bedeutung von Software-as-a-Service-Anwendungen

Mit *Software-as-a-Service*-Anwendungen (SaaS) werden Programme bezeichnet, welche dem Kunden als Dienstleistung angeboten, jedoch auf der IT-Infrastruktur des Dienstleisters betrieben werden. Der Kunde kann also auf den Dienst zugreifen, ohne eine eigenen ausreichende Umgebung für dessen Betrieb zu besitzen. Meist erfolgt dieser Zugriff über Web-Schnittstellen [1]. Durch die für den Kunden einfache Nutzung steigt der Bedarf nach der derartigen Angeboten. Beliebte Verwendungen sind Notizbuch- oder Kalender-Programme, da diese meist zentralisiert werden und von verschiedenen Orten erreichbar sein sollen. Generell ist diese Art von Dienst für die meisten Formen der zentralisierten Datenverwaltung geeignet. So braucht man auch beispielsweise bei einem neuen Rechner für ein Unternehmen nicht ein Verwaltungswerkzeug installieren, sondern kann direkt über einen Browser auf die Arbeitsdateien zugreifen. Ebenso muss der Anwender keine eigene Infrastruktur zum Betrieb des Dienstes bezahlen und versorgen, wodurch er auch Geld einsparen kann. Ein Nachteil von SaaS-Anwendungen ist hingegen die Abhängigkeit von der Verfügbarkeit durch den Anbieter. Jedoch können entsprechende Maßnahmen ergriffen werden, um längeren Ausfällen vorzubeugen. Dazu zählt ein redundantes System für den Dienst und eine stabile Internetanbindung des Kunden.

Das Beratungsunternehmen Gartner prognostiziert, dass Produkte auf Basis von SaaS auch in den folgenden zwei Jahren deutlich mehr Umsatz generieren werden [3]. SaaS-Anwendungen sind zudem der stärkste Umsatzzweig nach Gartner und konnten den Abstand zu den *Infrastructure-as-a-Service*-Anwendungen (IaaS) ausbauen.

Da allerdings eine Vielzahl an Diensten jeweils für spezielle Zwecke vorhanden ist, sind die Daten oft zwischen diesen verteilt. Möchte nun ein Kunde das Angebot wechseln und stellt der neue Anbieter keine Konvertierungsmöglichkeit für die Daten des anderen Anbieters bereit, so muss der Kunde entweder die Daten selbstständig übertragen oder auf diese verzichten. Dieses Problem kann jedoch durch Extrahierung und Injektion der Informationen durch die gegebenen Anwendungsschnittstellen und eine externe Konvertierung in einem neuen Anwendungsprogramm behoben werden. Dabei zeigt sich schnell, dass die größten Probleme die Diversität der Datenstrukturen und die Einschränkungen der Schnittstellen durch die Anbieter ist. Die Daten mögen vorhanden sein, jedoch gibt es oft schon bei der Angabe der letzten Änderungszeit in den Metadaten verschiedene Vorgehensweisen. Es muss also für jeden Dienst eine Konvertierung der Informationen auf eine gemeinsames Datenformat stattfinden. Falls die Anwendung eine Exportfunktion hat, kann man dabei diese für einen Konvertierung in ein einfaches Speicherformat nutzten. *Evernote* bietet

beispielsweise einen Export als *.enex*-Dateien (*.xml*-Format) an. Dienste wie *OneNote* von *Microsoft* schränken zusätzlich die Migrationsmöglichkeiten durch direkte Eingabe in die Schnittstelle ein. Beispielsweise können keine Metadaten beschrieben werden. Gleichzeitig bietet diese Anwendung aber auch verschiedene Import-Programme zur Datenmigration von den größten anderen Anbietern an. Beide Probleme müssen dabei für jeden Dienst einzeln beachtet und behandelt werden. Das in dieser Arbeit beschriebene Projekt mit Simon Hofmeister und Stephan Nunhofer unter der Aufsicht von Prof. Dr. Johannes Schildgen hat eine prototypische Zusammenführung und Vereinheitlichung der Konvertierungsvorgänge der einzelnen Anwendungen zum Ziel. Dadurch sind beispielsweise auch Datentransfers zwischen Notiz- und Kalender-Diensten möglich. Wichtigste Eigenschaft soll dabei die einfache Erweiterbarkeit sein, um einfach neue Dienste zu diesem Werkzeug hinzufügen zu können.

2 Umsetzung des Datentransformations- und Verteilungssystems

Die Hauptaufgaben dieses Werkzeuges ist die Gewinnung der Daten aus den verschiedenen Diensten. Diese Daten werden dann in einer Datenbank gespeichert und können bei einer Abfrage zur Übertragung in einen anderen Dienst verändert werden. Zur Ermöglichung dieses Ziels müssen die unterschiedlichen Informationen aus den Anwendungen auf eine gemeinsame Datenstruktur zusammengeführt werden. Darauf folgt die Festlegung der Klassenstruktur, eine Definition des Speichervorganges in die Datenbank und die Umsetzung der Rückführung in den selben oder einen anderen Dienst. Das Projekt fokussiert sich auf Dienstleister in den Bereichen Kalender- und Notiz-Verwaltung, wobei zu jedem Bereich jeweils drei Angebote eingebunden werden. Die Notiz-Verwaltung, auf welche sich diese Arbeit hauptsächlich konzentriert, wird dabei von *Keep* (Google), *OneNote* (Microsoft) und *Notion* (Notion) vertreten. Zur Umsetzung wird die Programmiersprache *Python* genutzt und die Verbindung mit den Diensten über entsprechende *Wrapper-Klassen* in dieser Sprache realisiert. Als Datenbank wird das Datenbanksystem *MongoDB* genutzt.

2.1 Zusammenführung der Daten und Metadaten

Als erster Schritt werden die Daten aus den verschiedenen Schnittstellen aufgelistet. Dafür werden alle möglichen Attribute, welche man aus den Diensten extrahieren kann, in einer Liste gesammelt und diese mit den Eigenschaften der anderen Dienste verglichen. Gibt es Übereinstimmungen in Bezeichnung oder Funktion, wir dieses Attribut für das gemeinsame Datenformat akzeptiert. Unter den gemeinsamen Daten gibt es dabei rein informative Informationen, wie beispielsweise den Titel oder einen Text, sowie für die Funktion des Dienstes strukturell notwendige Werte, wie die ID. Diese sind zwar für jede Anwendung vorhanden, unterscheiden sich allerdings nahezu jedes mal in ihren Werten. Bei der Verwendung dieser als gemeinsames Attribut muss noch eine Anpassung der Werte vorgenommen werden. Der Vorgang ist für die Notiz-Anwendungen in Abbildung 2.1 zu sehen.

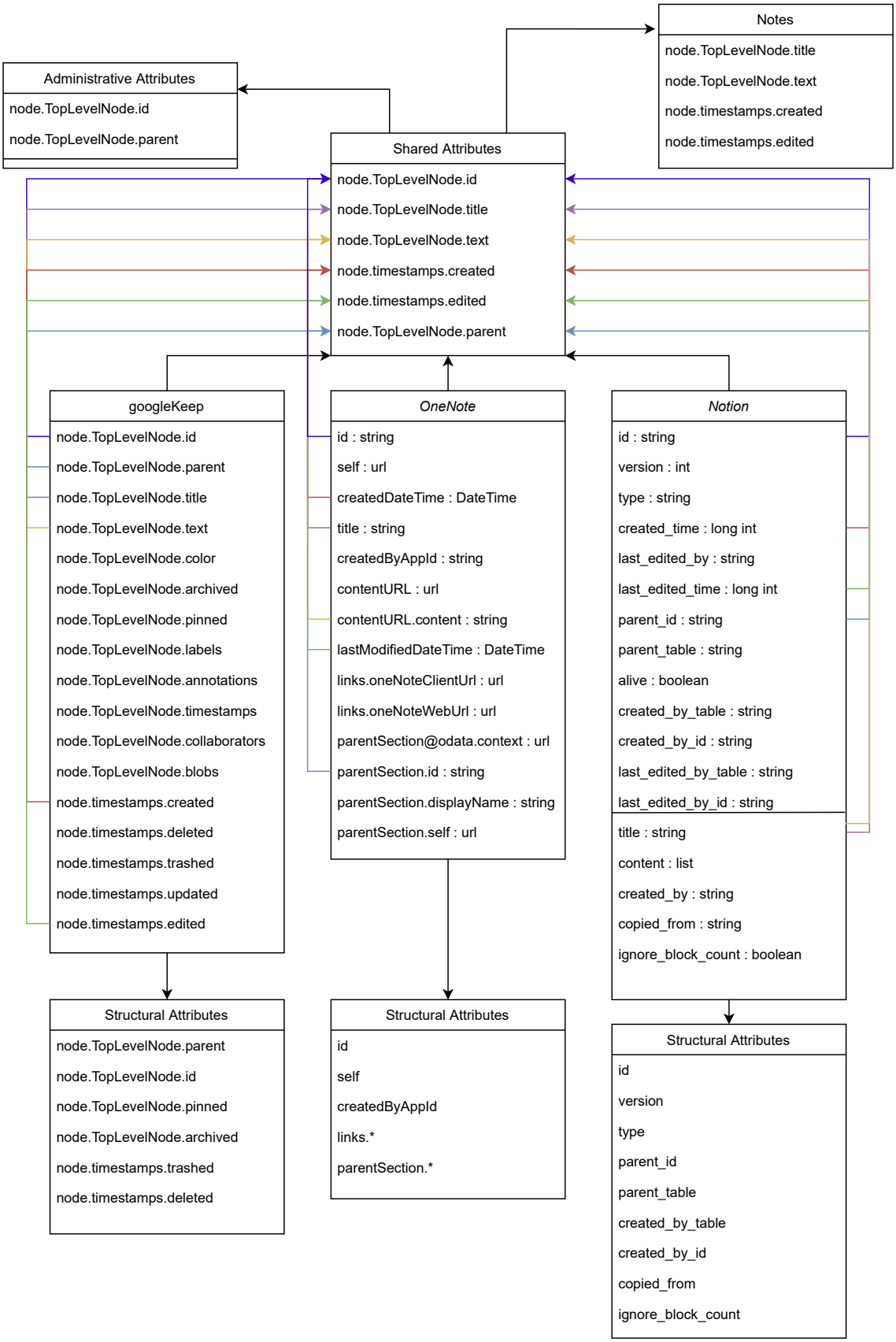

Abbildung 2.1: Darstellung des Attribut-Vereinigungsprozesses

Nach der Vereinigung dieser Attribute mit denen aus den Kalender-Diensten wurden folgende Attribute in Tabelle 2.1 für als feste, zwingend notwendige Eigenschaften eines jeden internen Datensatzes definiert.

Tabelle 2.1: Gemeinsame Attribute aller Dienste

Wert	**Datentyp**	**besondere Formatierung**
id	string	<Diensttyp>#<Dienstklassenname>#<Dienst-ID>
title	string	
text	string	
created	string	<J>-<M>-<T>T<S>:<M>:<S>.<ZZ>Z (UTC-Zeitformat)
edited	string	<J>-<M>-<T>T<S>:<M>:<S>.<ZZ>Z (UTC-Zeitformat)

Jeder Datensatz aus den Anwendungen muss diese Attribute enthalten, um in die Datenbank aufgenommen zu werden. Die übrigen Eigenschaften, welche nicht in allen anderen Diensten enthalten sind, werden ebenfalls in die Datenbank übertragen. Bei diesen muss jedoch bei der Rückführung des Datensatzes in den Dienst die Existenz des Attributes geprüft werden. Existiert dieses nicht, wird diese Information bei der Injektion ausgelassen, wodurch die Schnittstelle einen Standardwert einfügt.

2.2 Festlegung der Klassenstruktur

Um sicherzustellen, dass die festen Attribute vorhanden sind, werden die Daten in einer Klasse statt einer Datenstruktur gespeichert. Dabei gibt es eine Basisklasse, welche die festen Attribute enthält, und mehrere abgeleitete Klassen mit den diensteigenen, optionalen Werten. Diese Klassen besitzten keine Methoden, da sie als reine Speicherklassen genutzt werden. Objekte dieser Klassen werden durch die Dienst-Schnittstellenklassen erstellt. Deren Aufgabe ist die Kommunikation mit den Diensten über deren jeweilige Benutzerschnittstellen. Sie übernehmen also die Anmeldung, die Extrahierung von Daten aus der Anwendung und die Injektion der Daten aus der Datenbank zurück in den Dienst. Sie fungieren außerdem als Bedienungsschnittstelle für die Nutzeroberfläche. Dabei werden auch hier alle Dienst-Schnittstellenklassen von einer Basis-Klasse abgeleitet, wodurch man sicherstellt, dass alle Unterklassen auf die gleichen Resourcen zugreifen und die gleiche Funktionalität bereitstellen. Die letzte Klassengruppe stellen die Datenbank-Schnittstellenklassen dar. Diese ermöglichen den Datentransfer zu und von der Datenbank. Ebenso wie bei den Dienst-Klassen, wird auch hier eine Vererbungsstruktur genutzt, um eine Konsistenz zwischen den abgeleiteten Klassen sicherzustellen. Das gesamte Konzept ist dabei auf hohe Flexibilität und Erweiterbarkeit ausgelegt. So kann ein neuer Dienst einfach über das Hinzufügen einer neuen Dienst-Klasse eingebunden und die Datenbank über eine neue Datenbank-Klasse gewechselt werden. In beiden Fällen sind entweder keine oder nur wenige Zeilen an Änderungen in den vorhanden Klassen nötig. Durch das vereinheitlichte Datenkonzept ist zudem eine hohe Kompatibilität gegeben. Eine Übersicht über die Klassenstruktur ist in Abbildung 2.2 zu sehen. Die Dienst-Klassen sind dabei die

apiInterfaces, die Datenbank-Klassen die *datastores* und die Speicherklassen die *dataObjects*. Die Zeit-Attribute haben den Datentyp *string*, da zur Vereinheitlichung alle Zeitdatentypen in eine einheitlich formatierte Zeichenkette umgewandelt werden (zu sehen in Tabelle 2.1).

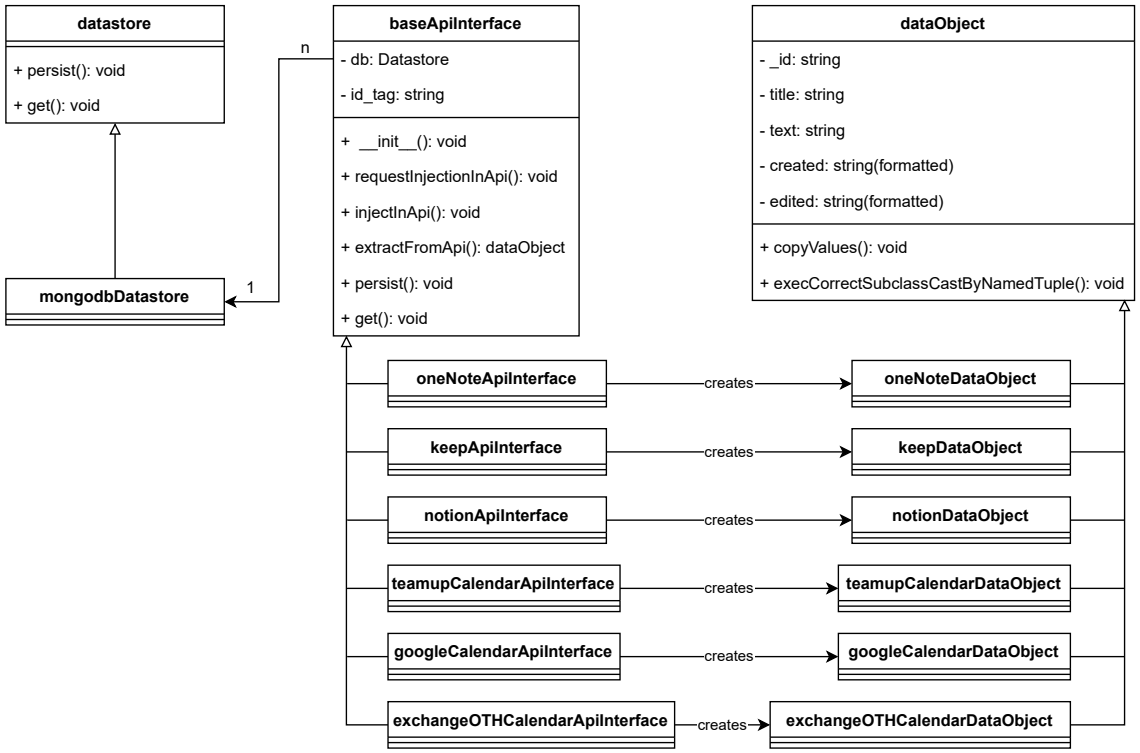

Abbildung 2.2: Vereinfachtes Klassendiagramm zur Darstellung der Klassenstruktur

2.3 Umsetzung des Speichervorganges

Um ein Element aus der Anwendung zu extrahieren, muss dessen Benutzerschnittstelle angesprochen werden. Zuerst meldet sich die Schnittstellenklasse bei dem Dienst an und erhält ein Zugriffsobjekt. Bei den Notiz-Diensten wird mit diesem Objekt zuerst eine Suche nach allen vorhandenen Elementen gestartet und deren Attributwerte in die jeweilige Speicherklasse übertragen. Dabei wird die *_id* mit einem Präfix aus dem Diensttypen und der Ursprungsklasse versehen. Dies ist dann beispielsweise der Zusatz *notes#keepApiInterface#* für den *Keep-Dienst* des Unternehmens *Google*. Zudem werden alle Zeitangaben auf das in Tabelle 2.1 dargestellte Zeitformat konvertiert. Die Speicherung der Daten-Klasse übernimmt die Methode *persist*. Diese nutzt das Attribut *db* aus der Basisklasse, um die *persist*-Methode aus der Datenbank-Klasse aufzurufen. Im Prototyp ist dies der *mongodb-Datastore*. Dieser wiederum stellt eine Verbindung zur Datenbank her und erhält dessen Verbindungsobjekt. Die Speicherklasse wird in eine *dict*-Speicherstrucktur umgewandelt und in den Dienst eingefügt. Es wird also eine Sammlung von Schlüssel-Werte-Paaren übergeben. Die Konvertierung ist dabei im Falle des *mongodbDatastores* nötig, da die Verbindungsklasse keine Klassen als Eingabeparameter akzeptiert und eine Konvertierung in ein *dict* leicht zu realisieren ist. Die Injektion in die Datenbank findet mithilfe der

replace_one()-Methode von *MongoDB* statt. Dabei ist die *upsert*-Option gesetzt, um sicherzustellen, dass nicht vorhandene Einträge eingefügt werden. Existieren diese bereits, ersetzt die Methode sie durch den neuen, übergebenen Eintrag, was einer Aktualisierung gleichkommt.

2.4 Definition des Extraktionsvorganges aus der Datenbank und der Datenrückführung in einen Dienst

Der Rückweg aus der Datenbank in die Dienste beginnt dabei ebenfalls in den Schnittstellenklassen. Die Notiz-Klassen bereiten in der Methode *requestInjection* die Ergebniszähler vor und rufen die Methode *requestInjectionInApi* auf. Dessen Parameter stammen dabei aus den übergebenen Werten der aufrufenden Methode. Diese haben dabei auch Standartwerte, welche dem Nutzer eine selektierte Angabe ermöglichen. Die Parameter sind in Tabelle 2.2 mit ihrem Datentypen und ihrer Bedeutung aufgelistet.

Tabelle 2.2: Parameter der Methoden zur Rückholung der Daten aus der Datenbank

Parameter	**Datentyp**	**Bedeutung**
substrIdTag	String	Stellt den Präfix dar, nach welchem die _id-Felder in der Datenbank gefiltert werden sollen.
filterOptions	Liste aus *Dictionaries*	Enthält die zusätzlichen Filteroptionen für die MongoDB Aggregationsmethode.
transformationOptions	Liste aus *Dictionaries*	Enthält die Datentransformationsinformationen für die MongoDB Aggregationsmethode. Beispielsweise werden Werte geändert oder Variablenwerte anderen Variablen zugewiesen.
addAggOptions	Liste aus *Dictionaries*	Damit können weitere MongoDB Parameter für die Aggregationsmethode angegeben werden. Beispielweise ist durch eine Angabe des *$unset*-Befehls eine Löschung eines Feldes möglich.

Die Methode *requestInjectionInApi* aus der Basis-Klasse leitet diese wiederum an die *get*-Methode der Datenbank-Klasse weiter. Jedoch kommt hier der Parameter *serviceObject* hinzu, welcher einfach eine Referenz auf das aufrufende Objekt - also die Schnittstellenklasse - ist. Die aufgerufene Methode des *mongodbDatastore* stellt dann zuerst eine Verbindung zur Datenbank her und definiert aus den übergebenen Werten eine *Aggregation Pipeline*. Damit werden die Daten nach den angegebenen Optionen gefiltert, transformiert oder anders verändert und an die Datenbank-Klasse als *Dictionary* zurückgegeben. Für jeden Eintrag in

dieser Wertesammlung wird nun die *injectInAPI*-Methode der aufrufenden Schnittstellen-Klasse über die übergebene Referenz aufgerufen. Diese meldet sich bei dem jeweiligen Dienst an und überprüft zuerst, ob die *id* aus dem Eintrag schon im Dienst vorhanden ist. Ist das der Fall, werden die Einträge aktualisiert. Wenn nicht, wird dieser als neuer Eintrag eingefügt. Die zusätzlichen Attribute, welche in den abgeleiteten Klassen von *dataObject* gespeichert werden, müssen dabei auf Existenz überprüft werden, da die Einträge auch von anderen Diensten stammen können. Eine weitere Aufgabe ist die Registrierung und das Abfangen von Fehlern bei der Injektion. Entsteht ein Fehler wird dieser abgefangen, ausgeben und die Variable *errorCount* inkrementiert. Kann der Eintrag problemlos verarbeit werden, erhöht sich die Variable *successCount*. In der Klasse *oneNoteApiInterface* ergibt sich dabei eine Besonderheit. Da der Dienst nur eingeschränkte Änderungen an den Einträgen zulässt, wird bei Erkennen eines Eintrages im Dienst die letzte Änderungszeit überprüft. Ist der Eintrag im Dienst jünger als der in der Datenbank wird die Variable *ignoredCount* inkrementiert und der Eintrag übersprungen. Andernfalls wird dieser neu eingefügt und als Erfolg gezählt. Nachdem alle Einträge eingefügt oder aktualisiert wurden, gibt die Ursprungsmethode *requestInjection* noch die Variablen *errorCount*, *ignoredCount* und *successCount* als Ergebnis aus. In Abbildung 2.3 ist die Ausgabe auf der Konsole für den gesamten Injektonsprozess zu sehen.

```
given filOps:  [{'updated': {'$gt': '2020-06-30T09:55:23.247000Z'}}]
filOp:  {'updated': {'$gt': '2020-06-30T09:55:23.247000Z'}}
given transOps:  [{'otitle': '$title'}, {'title': '$text'}, {'text': '$otitle'}]
given addOps:  [{'$unset': 'otitle'}]
pipeline:  [{'$match': {'_id': {'$regex': re.compile('^notes', re.IGNORECASE)}}}, {'$match': {'updated': {'$gt': '2020-06-30T09:55:23.2
47000Z'}}}, {'$set': {'otitle': '$title'}}, {'$set': {'title': '$text'}}, {'$set': {'text': '$otitle'}}, {'$unset': 'otitle'}]

{'_id': 'notes#keepApiInterface#1593510917375.1007550093', 'title': 'keep2', 'text': 'Keep2', 'edited': '2020-06-30T09:55:36.854000Z',
 'created': '2020-06-30T09:55:29.479000Z', 'parent_id': 'root', 'version': None, 'color': 'White', 'trashed': '1970-01-01T00:00:00.00000
0Z', 'updated': '2020-07-01T09:06:25.212000Z'}
Starting login
login successfull
Found preexisting Note
{'_id': 'notes#keepApiInterface#1593510930595.639083343', 'title': 'keep3', 'text': 'Keep3', 'edited': '2020-06-30T09:55:46.875000Z', '
created': '2020-06-30T09:55:39.504000Z', 'parent_id': 'root', 'version': None, 'color': 'White', 'trashed': '1970-01-01T00:00:00.000000
Z', 'updated': '2020-07-01T09:06:32.212000Z'}
Starting login
login successfull
Found preexisting Note
{'_id': 'notes#keepApiInterface#1593514952027.337224726', 'title': 'keep1', 'text': 'Keep1', 'edited': '2020-06-30T11:21:46.735000Z', '
created': '2020-06-30T11:21:34.722000Z', 'parent_id': 'root', 'version': None, 'color': 'White', 'trashed': '1970-01-01T00:00:00.000000
Z', 'updated': '2020-07-01T08:53:21.232000Z'}
Starting login
login successfull
Found preexisting Note

Results:
Notes failed to inject:  0
Notes successfully injected:  3
```

Abbildung 2.3: Konsolenausgabe für den Injektionsvorgang

2.5 Aufbau der grafischen Oberfläche

Bedient wird das System über eine einfache *Python* Benutzeroberfläche. Diese besteht aus der Auswahl der Dienste für Eingabe und Ausgabe der Daten, sowie der Selektion und Zuordnung von deren Attributen. In Zweiteren kann auch den Attributen des Eingabedienstes eine anderes Attribut des Ausgabedienstes oder ein fester, eigener Wert zugewiesen werden. Diese Konfigurationen stellen dann die Parameter aus Tabelle 2.2, welche für die *pipeline* in der Datenbankklasse genutzt werden. Die Auswahl in den einzelnen Felder findet über ein *Drop-Down* Menü statt. Bei den Attributen kann auch eine eigener Wert eingetragen werden. Zu sehen ist die Oberfläche in Abbildung 2.4.

Abbildung 2.4: Graphische Oberfläche des Systems

3 Mögliche zukünftige Anwendungen des Datentransformations- und Verteilungssystems

Der Prototyp an sich kann schon eine einfache Variante der gewünschten Funktionalität betreiben. Jedoch kann dieser noch weiter verbessert und ausgebaut werden, um eine angenehmere und umfangreichere Bedienung zu ermöglichen. Dabei gibt es einmal noch Verbesserungen für den jetzigen Funktionsstand und zum Anderen neue Möglichkeiten, welche das System gut ergänzen oder erweitern.

3.1 Mögliche Erweiterungen und Verbesserungen im jetzigen System

Eine gute Erweiterung des jetzigen Programms wäre die Beachtung der Attributwerte. So kann man die Wertebereiche überprüfen, um fälschliche Angaben in der Benutzeroberfläche herauszufiltern. Ebenso könnten aber auch die Filter detailliertere Argumente übernehmen. Es wäre beispielsweise möglich das Datum nicht als Zeichenkette sondern als tatsächliche Zeitpunkte zu vergleichen. Kennt der Nutzer nicht den Wertebereich eines Feldes und möchte einen eigenen Wert für ein Attribut in der grafischen Oberfläche setzen, ist eine Auswahlmöglichkeit aller Eingabemöglichkeiten von Vorteil. Diese ist nur bei kleinen Wertebereichen sinnvoll und kann über eine Abfrage des Datentyps im Dienst gewährleistet werden. Ein Beispiel hierfür ist eine Liste aller Farben für das *color*-Attribut des *Keep*-Notizdienstes, welches eine begrenzte Anzahl an Farben unterstützt. Die verschiedene Farben wären dann über ein *Drop-Down* Menü auswählbar für diese Attribut.
Eine weiter Verbesserung der Nutzbarkeit stellt eine Veränderungsmöglichkeit der verwendeten *pipeline* in der graphischen Oberfläche dar. Dabei wird die aus den Eingaben generierte Parameterkette dem Nutzer angezeigt und dieser kann diese nach Wunsch noch weiter verändern. Dies erfordert zwar Kenntnis der Datenbankdienstes, erlaubt erfahrenerer Nutzern jedoch eine deutliche Erweiterung der Anwendungsmöglichenkeiten des Systems. Die Eingabe des Nutzers muss dabei jedoch noch auf schädlichen Text oder unabsichtliche Löschvorgänge per Syntaxprüfung überprüft werden.
Soll es dem Nutzer nicht gestattet sein die *pipeline* direkt zu bearbeiten, aber ihm trotzdem deren gesamte Funktionalität zur Verfügung gestellt werden, kann man eine geheimes Eingabefeld für die *addAggOptions* aus Tabelle 2.2 einbauen. Dieses beinhaltet alle zusätzlichen Parameter der *pipeline* und ermöglicht somit Zugriff auf alle möglichen Funktionalitäten

der Datenbank. Normalerweise werden diese Optionen nur durch das System erstellt. Dem Kunden, welcher sich zusätzliche Funktionen wünscht, kann man so einfach eine Erweiterung anbieten, auch wenn er sich hierfür Wissen über die Funktionalität der *pipeline* aneigenen muss. Nützliche Links zu der Dokumentation der jeweiligen Datenbank können jedoch auch zur Verfügung gestellt werden.

3.2 Mögliche weiterführende Anwendungen und Erweiterungen des Systems

Das Programm setzt stark auf Flexibilität und Erweiterbarkeit. So kann man durch ein Hinzufügen von einer entsprechenden Dienst-Klasse und Speicherklasse einen neuen Dienst hinzufügen und ebenso über eine Datenbank-Klasse eine neue Datenbank einbinden. Theoretisch kann dieses Werkzeug also eine Schnittstelle für jede SaaS-Anwendung werden und den Datenaustausch zwischen ihnen gewährleisten. Wenn man nun den Transfer zu der Datenbank nicht über eine grafische Oberfläche, sondern über ein Event oder zeitgesteuert anstößt, kann die Datenbank auch als Backup für die Dienste fungieren. Die Informationen wären dadurch immer auf einen konfigurierbar neuen Stand. Wenn man nun zu bestimmten Zeiten Backups der Datenbank erstellt, kann man das System zu einem Versionsverwaltungssystem umbauen. Damit kann man beispielsweise auf alte Stände der Notizen zurückgreifen, falls die Einträge durch Unachtsamkeit gelöscht werden.
Möglich ist auch eine Analyse der gesammelten Einträge. Man kann dabei auch die Häufigkeiten der Aktualisierungen der einzelnen Dienste betrachten und so die aktivsten Programme herausfinden. Oder man betrachtet das Verhältnis der Einträge pro Typen des Eintrages. Gibt es also mehr Notizen als Kalender, kann man betrachten, in welchen Zeiten der Nutzer besonders aktiv war. Durch diese Analysen kann man dann die besten Dienste finden.
Nachteil des Systems sind jedoch die teilweise sehr eingeschränkte Rückführungsmöglichkeiten der Daten in die Anwendungen. Dadurch können eventuell wertvolle Metadaten in einigen Diensten nicht eingebunden werden. Dieses Problem kann dabei durch einen eigenen Dienst für das Werkzeug behoben werden, da dieser die Daten in ihrer internen Konvertierung darstellen kann. Er ist dann auch problemlos in der Lage, alle zusätzlichen Informationen aus den Diensten darstellen, da er nicht an ein spezielles Muster oder andere Vorgaben gebunden ist. Eine solche Anwendung könnte dann auf alle Daten der eingebunden Dienste zugreifen und diese manipulieren und auslesen. So ist beispielweise eine Export-Möglichkeit der Datenbank im *JSON*-Format denkbar. Auch können in diesen neue Funktionen wie das Analyseprogramm eingebunden werden. Man könnte nach dem Vorbild *Maven* ein *PlugIn*-System einbauen, womit neue Funktionalitäten einfach hinzugefügt werden können. Diese wurden dann in einen speziellen Ordner innerhalb des Programmverzeichnisses geladen und beispielsweise täglich ausgeführt. Eventuell stellt man dann auch, wie bei *Maven*, durch eine Verbindungen mit einem Versionsverwaltungssystem eine öffentliche Bibliothek für diese Teilprogramme bereit.

Literaturverzeichnis

[1] McNee, W: *SaaS 2.0.* Journal of Digital Asset Management, 3:209–214, August 2007. https://link.springer.com/article/10.1057/palgrave.dam.3650088#citeas.

[2] Gartner: *Umsatz mit Software-as-a-Service (SaaS) weltweit von 2010 bis 2018 und Prognose bis 2022 (in Milliarden US-Dollar)*, Juli 2020. https://de.statista.com/statistik/daten/studie/194117/umfrage/umsatz-mit-software-as-a-service-weltweit-seit-2010/.

[3] *Gartner Forecasts Worldwide Public Cloud Revenue to Grow 172020*, November 2019. https://www.gartner.com/en/newsroom/press-releases/2019-11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020.