Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

KARL LUDWIG WERNER
MASTER THESIS

DESIGN UND IMPLEMENTIERUNG EINES KONZEPTS ZUR LIVE-KONFIGURATION

Eingereicht am 20. Dezember 2019

Betreuer:
Prof. Dr. Dirk Riehle, M.B.A.
Andreas Bauer, M.Sc.
Georg Schwarz, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

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, 20. Dezember 2019

License

Erlangen, 20. Dezember 2019

Abstract

Nowadays a variety of systems generate and handle more and more data. This data is often presented in different formats and has to be retrieved from various locations. The Open Data Service (ODS) presents a system to conquer those tasks. The ODS allows the definition of data manipulation pipelines to process and convert data. Those data transformations are defined by code snippets executed on the data.

This thesis presents a concept for the live configuration of those transformations. With live configuration the user receives instant feedback about his code, similiar to using an integrated development environment (IDE). A live preview of the resulting data is also shown. The result is a faster and more efficient development of the transformation snippets. The thesis describes the creation of the live configuration concept and documents an exemplary implementation of the concept as a web application.

Zusammenfassung

In der heutigen Zeit werden immer mehr Daten erzeugt, welche von zahlreichen Systemen verarbeitet und ausgewertet werden. Diese Daten liegen häufig in verschiedenen Formaten vor und müssen von verschiedenen Orten abgerufen werden. Mit dem Open Data Service (ODS) wird ein System angeboten, dass diese Aufgaben übernehmen kann. Der ODS ermöglicht die Definition von Datenverarbeitungspipelines, um Daten zu verarbeiten und zu konvertieren. Diese Transformationen enthalten kleine Codeabschnitte, die durch den Transformationsdienst ausgeführt und auf die Daten angewendet werden.

Diese Arbeit präsentiert ein Konzept zur Live-Konfiguration dieser Transformationen. Bei Live-Konfiguration erhält der Nutzer, ähnlich wie in integrierten Entwicklungsumgebungen, sofortige Rückmeldung über die Korrektheit des geschriebenen Codes. Zusätzlich wird eine Live-Vorschau des Ergebnisses angezeigt. Hierdurch ist es möglich, die Transformationen schneller und effizienter zu entwickeln. Die Arbeit beschreibt die Erstellung des Live-Konfigurations-Konzepts und dokumentiert die beispielhafte Implementierung des Konzepts als Webanwendung.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

BSD Berkeley Software Distribution

CSS Cascading Style Sheets

CSV Comma-seperated values

FTP File Transfer Protocol

HTTP Hypertext Transfer Protocol

ID Identifier

IDE integrated development environment

JSON JavaScript Object Notation

LSP Language Server Protocol

MVC Model-View-Controller

MVP Minimum Viable Product

MVVM Model View ViewModel

OCP Open-Closed-Prinzip

ODS Open Data Service

REST Representational State Transfer

RPC Remote Procedure Call

SRP Single-Responsibility-Prinzip

TDD Test-Driven Development

XML Extensible Markup Language

1 Einleitung

Die Wissenschaft der Informatik beschäftigt sich mit der „maschinellen Informationsverarbeitung“ [BG11]. Heutzutage erzeugen Verkaufsanalysen oder Sensornetze kontinuierlich Unmengen an Daten, die von zahlreichen verschiedenen Systemen verarbeitet und ausgewertet werden. Unter dem Namen „Big Data“ hat sich ein neuer Forschungszweig entwickelt, der neue Techniken zur Analyse von großen Datenmengen präsentiert [BCD16].

Parallel dazu setzt sich die „Open Data“-Bewegung für eine offene Bereitstellung von Daten jeglicher Art ein. Hier sind zumeist Daten von öffentlichen Ämtern oder Wissenschaftlern gemeint, der Begriff Open Data ist darauf aber nicht beschränkt [Wes+17].

Eine besonderes Augenmerk ist auf das Problem des standardisierten Zugriffs und der Verarbeitung der Daten zu richten. Derzeit entwickeln alle Institutionen, welche solche Daten anbieten, seien es Regierungen, Wissenschaft oder Industrie, ihr eigenes „Ökosystem“ zur Bereitstellung und Präsentation [Wes+17]. Ein Softwareprodukt, das diese Daten nutzen möchte, muss folglich verschiedene Abrufmechaniken unterstützen und eine Möglichkeit zur Konvertierung dieser Daten implementieren. Dies muss gleichartig in jedem Projekt wiederholt werden.

Eine Möglichkeit, diesen wiederholten Arbeitsaufwand zu vermeiden, ist die Benutzung des *JValue Open Data Service*. Der Open Data Service (ODS) ist eine Open-Source-Software, die an der Professur für Open-Source-Software der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt wird. Das Projekt stellt ein einfach zu konfigurierendes System bereit, das die Aufgaben des Datenabrufs, der Datenaufbereitung und das Bereitstellen der Daten übernehmen kann. Entwickler können hierbei als Anwender auf die vorhandene Infrastruktur des Dienstes zurückgreifen und müssen sich nur um ihre eigene Konfiguration kümmern.

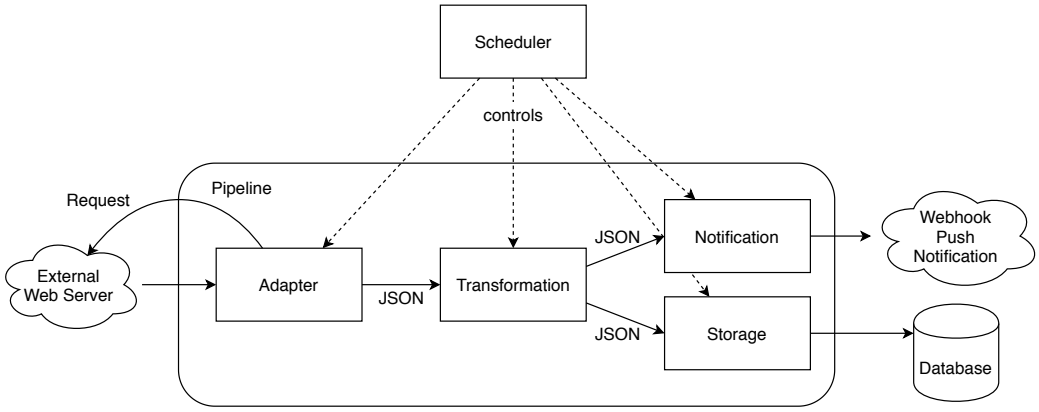

Abbildung 1.1: Bestandteile und Interaktionen des Open Data Service

1.1 Konzept des Open Data Service

Abbildung 1.1 beschreibt den Aufbau des ODS. Im ODS können sogenannte *Pipelines* angelegt werden, welche eine automatisierte Datenverarbeitung beschreiben. Die Pipelines werden entsprechend ihrer Konfiguration periodisch durch den *Scheduler* ausgeführt. Der Scheduler kommuniziert hierzu mit dem Adapterdienst, dem Transformationsdienst sowie dem Benachrichtigungs- und Speicherdienst. Diese Dienste sind jeweils als eigene separate *Microservices* implementiert.

Adapter stellen den Kontakt zur Außenwelt her. Der Adapterdienst übernimmt den Abruf der Daten von anderen Webservern oder -diensten. Der ODS unterstützt die Verwendung von verschiedenen Adapterimplementierungen für unterschiedliche Transportprotokolle und Datenformate. Neben der Unterstützung des Hypertext Transfer Protocols (HTTP) ist auch eine Implementierung für das File Transfer Protocol (FTP) möglich. Im Bereich der Datenformate werden derzeit die JavaScript Object Notation (JSON) und die Extensible Markup Language (XML) unterstützt. JSON wird auch als internes Datenformat im ODS benutzt. Die Verwendung von Comma-seperated values (CSV) ist ebenfalls bereits implementiert.

Als nächster Schritt innerhalb des ODS werden *Transformationen* konfiguriert, welche die Aufbereitung und Bereinigung der Daten implementieren. Der Anwender hinterlegt hierfür Codeabschnitte in einer Scriptsprache. Bei der Ausführung der Pipeline sendet der Scheduler diese zusammen mit den Daten an den Transformationsdienst, der den Code ausführt und die transformierten Daten zurückliefert.

Den Abschluss der Kette stellen *Storage* und *Notification* dar. Jedes Ergebnis

eines Pipelinedurchlaufs wird im Storage-System gespeichert. Diese Resultate können später einzeln oder auch gesammelt abgerufen werden. Mithilfe von *Notifications* ist es möglich, direkt aus dem ODS heraus Nachrichten an Slack oder Mobilgeräte über Push-Benachrichtigungen zu senden.

In der Anfangsphase des ODS-Projekts wurden die Pipelines nur manuell über eine REST-Schnittstelle konfiguriert. Dies war umständlich, fehleranfällig und wenig benutzerfreundlich.

Im Rahmen der weiteren Entwicklung des ODS ist eine Weboberfläche entstanden, mit der die Pipelines und ihre Bestandteile schrittweise konfiguriert werden können (siehe Abbildung 1.2).

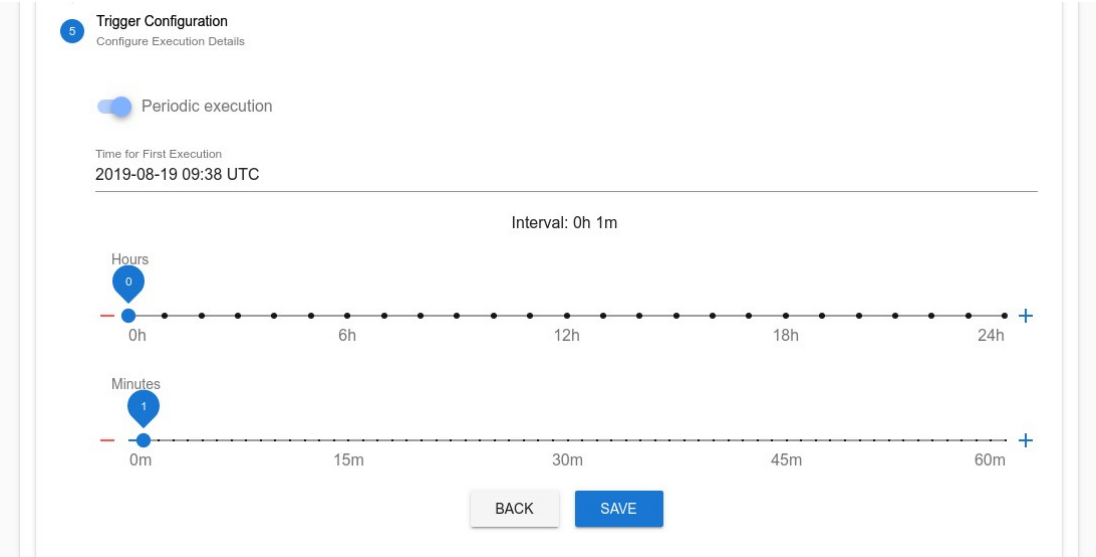

Abbildung 1.2: Teil der Weboberfläche der Pipelinekonfiguration

1.2 Ziel der Arbeit

Die bereits bestehende Weboberfläche zur Konfiguration der Pipelines ist zwar ein deutlicher Fortschritt gegenüber der manuellen Konfiguration per REST-Schnittstelle, jedoch besteht hier noch viel Verbesserungspotential. Insbesondere die Transformationen sind essentieller Teil der Pipelines und stellen derzeit eine erhebliche Barriere für Entwickler dar, da ihre Konfiguration bislang nicht sehr benutzerfreundlich sind.

Ziel dieser Arbeit ist es, die Konfiguration des Transformationsdienstes im Webinterface im Stil einer Live-Konfiguration zu konzeptionieren und zu implementieren.

Mit Live-Konfiguration bezeichnen wir die Idee, die Eingaben des Nutzers möglichst schnell zu verarbeiten. Die Rückmeldung auf Erfolg oder Fehler erfolgt zeitnah und ermöglicht so einen guten Arbeitsfluss. Dieses Prinzip hat sich auch in anderen Domänen bewährt, wie etwa bei testgetriebener Entwicklung, englisch Test-Driven Development (TDD). Ein essentieller Bestandteil von TDD ist die schnelle Rückmeldung des Status der Tests an den Entwickler.

Die Vorteile von Live-Konfiguration sollen nun auch dem ODS zugute kommen. Das aktive Feedback während der Erstellung der Codeabschnitte soll sicher stellen, dass nur syntaktisch korrekte Transformationen gespeichert werden können. Außerdem soll aktiv auf Fehler hingewiesen werden und eine Vorschau der Transformationsergebnisse die gewünschte Manipulation der Quelldaten während der Konfiguration gewährleisten. Mithilfe des Echtzeit-Feedbacks soll zusätzlich die aufgewendete Zeit für die Konfiguration der Transformationen verringert werden.

1.3 Aufbau der Arbeit

Nach dieser Einleitung werden in Kapitel 2 die *Anforderungen* an das im Laufe dieser Arbeit entstehende Konzept und dessen Implementierung festgelegt. Neben den funktionalen und nicht-funktionalen Anforderungen wird auch ein Bewertungsmodell für die Evaluation definiert.

In Kapitel 3 wird der Prozess erläutert, durch welchen das Konzept zur Live-Konfiguration entstanden ist.

Die eigentliche *Umsetzung* und Implementierung dieses Konzepts werden in Kapitel 4 beschrieben. Hierbei wird auf die Entwicklung sowohl der Weboberfläche als auch des zugrunde liegenden Transformationsdienstes eingegangen.

Die Erfüllung der in Kapitel 2 aufgelisteten Anforderungen wird in Kapitel 5 überprüft. Dies geschieht mithilfe des vorher definierten Bewertungsmodells.

Abschließend bietet Kapitel 6 eine Zusammenfassung der Arbeit und einen Ausblick auf weitere Möglichkeiten zur weiteren Entwicklung des ODS.

2 Anforderungen

2.1 Funktionale Anforderungen

2.1.1 Transformationsdienst

A1: Ausführungsmechanismus

Der Transformationsdienst des Open Data Service (ODS) soll, wie in Kapitel 1.1 beschrieben, die Ausführung von durch den Nutzer verfassten Operationen zur Datenverarbeitung unterstützen. Dabei werden dem Transformationsdienst ein Codeabschnitt und Daten übergeben, der Code wird dann auf diese angewendet und das Ergebnis zurückgeliefert.

A2: Fehlerüberprüfung der Transformationsfunktionen

Bei der Ausführung der Codeabschnitte ist die robuste Fehlerbehandlung essentiell. Sowohl Syntaxfehler im Code als auch Fehler bei der Ausführung des Codes im Transformationsdienst (Runtime-Errors) sollen abgefangen werden. Die Fehlermeldung muss eine aussagekräftige Nachricht sowie einen Hinweis auf die Lokalität des Fehlers bereitstellen (Methode, Codezeile).

A3: Laufzeitverhalten und Metadaten

Der Transformationsdienst soll geeignete Metadaten wie Ausführungszeitpunkt, Ausführungsdauer oder Nutzung von Ressourcen erfassen und zurückliefern, damit der Nutzer die Ressourcenkosten der Transformation überwachen kann.

A4: Schnittstelle zu anderen Microservices

Die Schnittstelle des Transformationsdienstes soll als Representational State Transfer (REST)-Schnittstelle realisiert werden. Hierbei soll ein Endpunkt angeboten werden, welcher die Eingabedaten als JSON und den Transformationscode als Zeichenkette annimmt. Dieser Endpunkt liefert die Metadaten sowie das Ergebnis oder Fehler der Transformation im JSON-Format zurück.

2.1.2 Benutzeroberfläche

A5: Testen der Transformation

Die Weboberfläche muss die Möglichkeit bieten, Beispieldaten und Transformationscode in einem Codeeditor anzugeben und einen Test der Transformation anzustoßen. Das Resultat des Aufrufs des Transformationsdienstes muss in der Benutzeroberfläche angezeigt werden. Neben der manuellen Interaktion soll das System das Ergebnis der Transformation nach Änderungen auch automatisch anzeigen, ohne dass ein manuelles Bestätigen durch den Nutzer erforderlich ist. Durch die automatische Ausführung der Transformation soll der Arbeitsfluss besser erhalten bleiben.

A6: Syntax-Hervorhebung im Codeeditor

Das Eingabefeld für den Code in der Weboberfläche soll Syntax-Hervorhebung unterstützen, wie es etwa aus integrierten Entwicklungsumgebungen, engl. IDEs, auf dem Desktop bekannt ist.

A7: Anzeige der Fehlermeldungen im Code-Editor

Die vom Transformationdienst zurückgegebenen Fehlermeldungen (siehe Anforderung A2) sollen direkt in der entsprechenden Zeile im Codeeditor angezeigt werden.

A8: Anzeige der Metadaten in der Weboberfläche

Die in Anforderung A3 in den Transformationsdienst eingebauten Metadaten müssen auch in der Weboberfläche dargestellt werden, um einen Überblick über die Dauer und Ressourcennutzung der Transformation bereitzustellen.

A9: Template-Mechanismus

Wiederverwendbare Skripte sollen als Templates abgelegt werden können. Diese können in den Transformationscode eingefügt werden, um oft auftretende Probleme wie z.B. Konvertierung von Datumsangaben zu übernehmen.

2.2 Nicht-funktionale Anforderungen

2.2.1 Gesamtes Konzept

A10: Portabilität und Anpassbarkeit

Das entwickelte Konzept sowie dessen Umsetzung sollen anpassbar und änderbar sein, insbesondere in Bezug auf die verwendete Transportmethode (vergleiche Anforderung A4) und die verwendete Programmiersprache zur Datenverarbeitung.

2.2.2 Transformationsdienst

A11: Sicherheit (Ausführung)

Der Transformationsdienst muss die Ausführung von Schadcode verhindern. Innerhalb der Codeabschnitte dürfen keine Bibliotheken nachgeladen werden und der Zugriff auf Systemfunktionen muss beschränkt sein. Auch ein Zugriff oder Manipulation von Pipelines anderer Nutzer soll ausgeschlossen sein.

A12: Sicherheit (Dauer)

Die Ausführungszeit eines Skriptes muss zeitlich begrenzt sein, um ein Blockieren des Dienstes für andere Nutzer zu verhindern.

A13: Verfügbarkeit

Es darf nicht möglich sein, den Transformationsdienst durch die Ausführung von schadhaften Codeabschnitten zu blockieren oder zum Absturz zu bringen.

2.2.3 Benutzeroberfläche

A14: Usability

Die Benutzeroberfläche der Webkomponente soll übersichtlich, ansprechend und benutzerfreundlich sein.

2.3 Bewertungsmodell

In Vorbereitung auf die Auswertung in Kapitel 5 wird das Bewertungsmodell festgelegt, auf dessen Grundlage das in der Umsetzung erstellte Projekt im Hinblick auf die Anforderungen überprüft wird.

Alle funktionalen Anforderungen werden einzeln daraufhin überprüft, ob sie vollständig, teilweise oder nicht erfüllt wurden.

Die *Usability* des Systems soll anhand der ISO Norm 9241-110 „Dialoggestaltung“ überprüft werden. Diese Norm beschreibt sieben verschiedene Grundsätze der Dialoggestaltung und die dazugehörigen Empfehlungen. Da nicht alle Empfehlungen auf auf den Anwendungsfall der Live-Konfiguration zutreffen, werden je Grundsatz ein bis zwei Empfehlungen ausgewählt und mit der Implementierung abgeglichen.

Die *Portabilität* des Konfigurationskonzepts soll anhand einer beispielhaften Implementierung für eine andere Programmiersprache evaluiert werden. Insbesondere die Anzahl der notwendigen Änderungen an der Weboberfläche sind ein Maß für die Anpassbarkeit.

Die *Sicherheit* und Sicherstellung der *Verfügbarkeit* des Transformationsdienstes wird überprüft, indem schadhafter Code übergeben wird, der Endlosschleifen oder Zugriffe auf Systemaufrufe enhält. Dieser Code darf nicht ausgeführt werden und darf den Transformationsdienst nicht blockieren. Es darf zu keinem Absturz kommen, stattdessen müssen entsprechende Fehlermeldungen zurückgeliefert werden.

3 Konzeption

Dieses Kapitel beschreibt die Erstellung eines Konzepts, um die Anforderungen aus Kapitel 2 zu erfüllen. Abschnitt 3.1 betrachtet die Generalisierung des konkreten Problems hin zu der Entwicklung eines generischen Konzepts zur Live-Konfiguration. Dieses generische Konzept wird in Abschnitt 3.2 erläutert und auch im Hinblick auf die Codearchitektur beschrieben. Schließlich wird in Abschnitt 3.3 die generische Lösung auf den konkreten Fall der Konfiguration des Transformationsdienstes im ODS angewendet und weiter ausgearbeitet.

3.1 Rahmenbedingungen

Zur Erstellung eines generischen Konzepts zur Live-Konfiguration kann es hilfreich sein, das bisherige Konfigurationssystem, welches im Rahmen des ODS benutzt wird, zu betrachten.

Bisher werden Pipelines mithilfe einer Weboberfläche konfiguriert, welche als Clientseitige JavaScript-Anwendung im Browser realisiert ist. Die Verwendung einer Weboberfläche ermöglicht eine Verfügbarkeit ohne zusätzliche Installation eines Clients und Portierbarkeit durch Plattformunabhängigkeit [MP13].

Die Kommunikation mit den Diensten des ODS erfolgt hierbei über HTTP-Aufrufe, welche dem REST-Prinzip folgen. Generell ist jedoch nur relevant, dass der Aufruf der Transformations-API über das Netzwerk möglich gemacht wird. Dies kann auch mithilfe von anderen Protokollen wie beispielsweise Remote Procedure Calls (RPCs) oder WebSockets geschehen. Es ist also wünschenswert, über das verwendete Transportsystem zu abstrahieren und die Verwendung von verschiedenen Transportmöglichkeiten zu unterstützen.

Auf Seite des Transformationsdienstes ist zu beachten, dass dieser, wie alle anderen Dienste im ODS, als Microservice gestaltet ist. Die Benutzung von Microservices ist ein Architekturstil, der verwendet wird, um verschiedene Bestandteile eines Softwaresystems logisch zu trennen und getrennt betreiben zu können [LF14]. So sind Microservices dem Namen entsprechend meist klein und erfüllen

eine spezielle Aufgabe. Ihre Schnittstellen sollen hierbei die dahinter liegende Implementierung nach dem Prinzip der Kapselung verbergen. Eine Eigenschaft von Microservice-basierten Systemen ist die Skalierbarkeit. Zustandslose Implementierungen haben hier einen entscheidenden Vorteil: sie können durch das Hinzufügen von Instanzen des Dienstes skaliert werden [New15].

Dementsprechend ist zu bevorzugen, dass der zu konfigurierende Dienst generell, und auch der Transformationsdienst im ODS im Speziellen, zustandslos implementiert werden. Im ODS wurde ein funktionaler Ansatz gewählt, bei dem die auszuführende Transformation und die Daten an die REST-API übergeben werden. Wie bei funktionaler Programmierung [CM98] ist die Testmethode somit eine pure, seiteneffektfreie Methode, welche bei wiederholtem Aufruf stets dieselben Ergebnisse zurückliefert (abgesehen von Metadaten wie dem Ausführungszeitpunkt). Dies erleichtert das Testen von verschiedenen Konfigurationen.

3.2 Generelles Konzept

Das generelle Konzept der Live-Konfiguration soll also auf die Verwendung mit einem zustandslosen Microservice zugeschnitten sein.

Der Microservice muss dafür mehrere Anforderungen erfüllen. Um dem Nutzer während dem Konfigurationsprozess Feedback zu geben, ist es notwendig, dem zu konfigurierendem Dienst alle Konfigurationsoptionen sowie Beispieldaten in einem Aufruf zu übergeben. Der Dienst testet diese und meldet entweder einen Erfolg und das Ergebnis oder Fehlermeldungen zurück. Der dafür verwendete Methode sollte idempotent gestaltet sein. Ein wiederholter Aufruf dieser API mit denselben Eingabedaten soll jeweils dasselbe Ergebnis zurückliefern.

Wichtig ist zudem die Ausgabe von aussagekräftigen Fehlermeldungen, welche auch in Bezug zu den übergebenen Konfigurationselementen stehen. Dies ist notwendig, um den Fehler in der Konfiguration zu lokalisieren und letztendlich möglichst schnell beheben zu können.

Zuletzt ist bei der Erstellung des Microservice noch zu beachten, dass die Rückgabe des Ergebnisses beziehungsweise der Fehlermeldung zeitnah geschieht, maximal innerhalb von ein paar Sekunden. Nur so kann eine kurze Feedback-Loop und ein flüssiges Arbeiten sichergestellt werden, welches dem Konzept der „Live“-Konfiguration gerecht wird [Nah03].

Beim Erstellen der Weboberfläche ist zu beachten, dass alle Konfigurationselemente auf einen Blick sichtbar und konfigurierbar sind. Bei jeglicher Änderung an diesen oder an anderen zu übergebenden Daten sollte eine Auswertung der Konfiguration automatisch angestoßen werden, ohne eine weitere Eingabe des

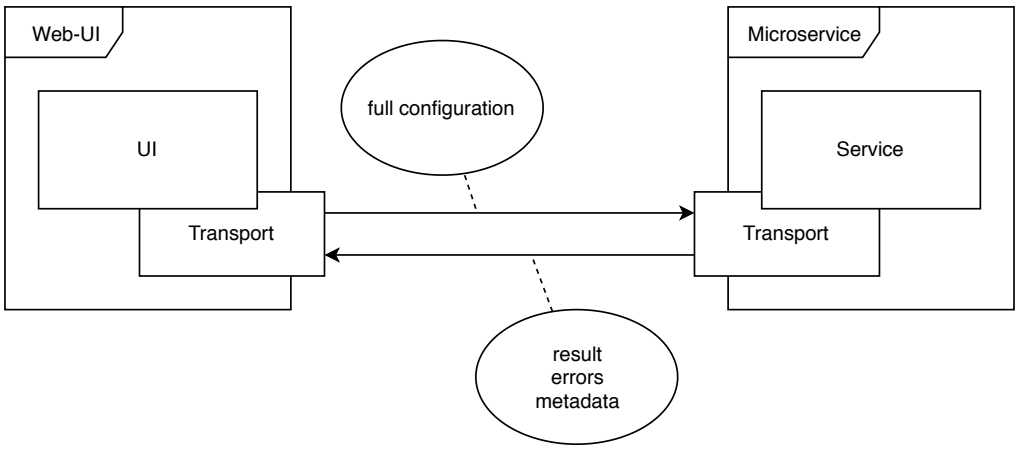

Abbildung 3.1: Generelles Konzept zur Live-Konfiguration eines Microservice

Nutzers abzuwarten. Das Ergebnis dieses Aufrufs muss daraufhin ebenfalls sofort sichtbar angezeigt werden, wobei etwaige Fehlermeldungen hervorgehoben und ihr Ursprungsort markiert werden muss. Diese Verknüpfung zwischen Resultat, Fehlermeldung und Konfigurationselement ermöglicht es dem Nutzer, sofort eine Verbindung zwischen diesen herzustellen.

Die Verwendung der Weboberfläche sollte bei der Aktualisierung des Ergebnisses nicht blockiert werden. Dazu ist es notwendig, die Kommunikation mit dem Dienst asynchron zu gestalten. Im Falle der Weboberfläche wird dadurch die Verwendung von Formularfeldern, welche ihre Anfrage über HTTP-Post übermitteln, ausgeschlossen; stattdessen müssen Techniken wie Asynchronous JavaScript and XML (AJAX) oder WebSockets benutzt werden. Die konkrete Transportmethode ist jedoch implementationsabhängig, das Konzept sollte hierüber abstrahieren.

Abbildung 3.1 zeigt schematisch den Aufbau des generellen Konzepts zur Live-Konfiguration.

3.2.1 Softwarearchitektur

Bei dem Erstellen der Softwarearchitektur für das generelle Konzept der Live-Konfiguration werden folgende Designprinzipien bedacht, um die Erweiterbarkeit und Wartbarkeit des entstehenden Codes zu gewährleisten.

Kopplung und Kohäsion

Bereits 1979 beschrieben Edward Yourdon und Larry L. Constantine die Prinzipien der Kopplung und Kohäsion [YC79]. Beides sind Maße, welche die Komplexität der Beziehungen zwischen Komponenten beschreiben.

Mehrere Komponenten einer Software sollten nur lose miteinander gekoppelt sein. Komponenten sollten nur über dafür vorgesehene Schnittstellen miteinander kommunizieren, um nicht auf etwaigen Implementierungsdetails zu basieren. Durch lose Kopplung wird die Wiederverwendbarkeit des Codes erhöht und Veränderungen von Anforderungen erfordern nur Änderungen an einzelnen Modulen.

Die Trennung des Softwaresystems in mehrere Microservices entspricht dem Designprinzip der losen Kopplung. So können die Implementationen der Weboberfläche oder des zu konfigurierenden Dienstes leicht ausgetauscht werden, solange sich die verwendeten Schnittstellen nicht ändern.

Innerhalb einer Komponente sollte darauf geachtet werden, dass diese nach außen eine logische Einheit darstellt und somit eine hohe Kohäsion besitzt. Dafür sollte sie nur für bestimme Aufgaben verantwortlich sein (strikter ist hier das Single-Responsibility-Prinzip, siehe nächster Abschnitt).

Single-Responsibility-Prinzip und Open-Closed-Prinzip

Das Single-Responsibility-Prinzip (SRP) besagt, dass es nie mehr als einen Grund geben sollte, eine Klasse zu ändern [Mar09]. Später änderte der Autor seine Definition auf „Ein Modul sollte nur einem, und nur einem, Akteur gegenüber verantwortlich sein.“ [Mar17]. Softwaresysteme sollten also aus mehreren kleinen Komponenten bestehen, und nicht aus wenigen Großen. Jede Klasse sollte eine einzelne Verantwortung haben, einen einzigen Grund sich zu ändern und mit anderen Klassen zusammenarbeiten.

Mit dem Thema der Veränderung beschäftigt sich auch das Open-Closed-Prinzip (OCP). Es besagt, dass Module sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein sollten [Mey88]. Ein Beispiel ist die Verwendung von Vererbung, um mehr Möglichkeiten von Verhalten hinzuzufügen, ohne bisherigen Code ändern zu müssen.

3.2.2 Anwendung auf die Softwarearchitektur

Das Single-Responsibility-Prinzip, das Open-Closed-Prinzip und die Konzepte der Kopplung und Kohäsion werden in der Softwarearchitektur des Live-Konfigurations-Konzepts angewendet (siehe Abbildung 3.2 und 3.3). Mehrere Bereiche

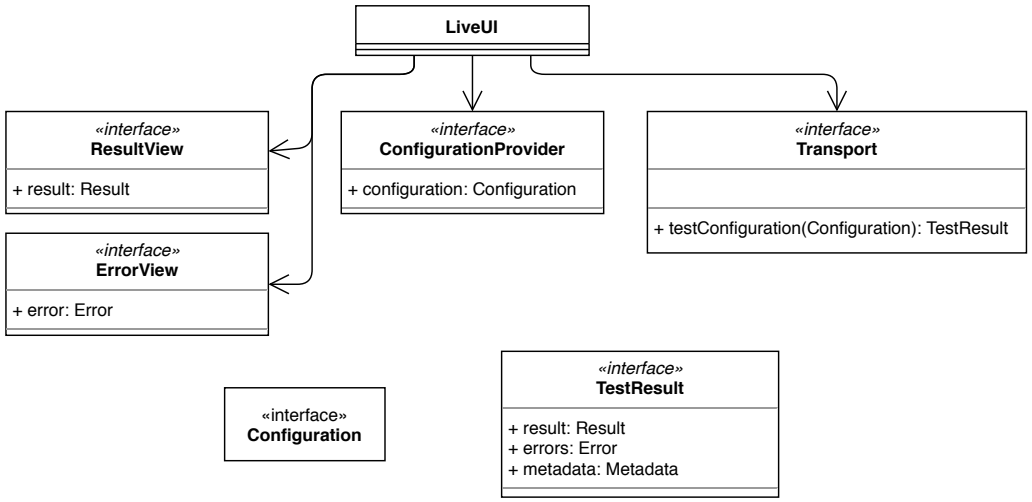

Abbildung 3.2: Codestruktur der Weboberfläche

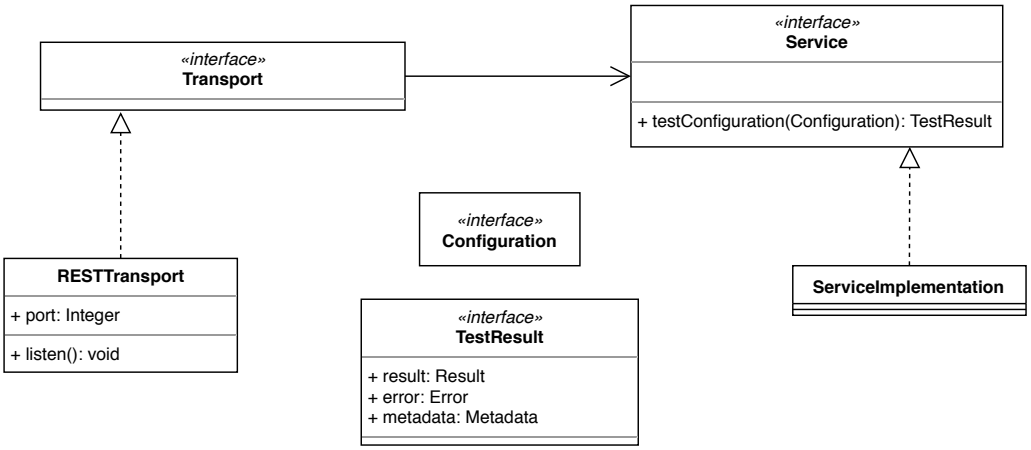

Abbildung 3.3: Codestruktur des Dienstes

werden modularisiert, um eine Erweiterung zu vereinfachen. So wird sowohl in der Weboberfläche als auch in der Implementation des Dienstes die Transportschicht ausgelagert. Hierdurch können verschiedene Varianten implementiert werden, ohne eine Änderung des restlichen Codes zu erfordern. Ein Beispiel hierfür ist die ursprüngliche Verwendung von HTTP-REST-Aufrufen, welche zum Beispiel durch Websockets ersetzt werden kann.

In der Weboberfläche wird, angelehnt an das Entwurfsmuster Model-View-Controller (MVC), die Steuerung der Applikation und die Anzeige und Änderung der Konfigurationselemente voneinander getrennt. Die Bereitstellung der zu testenden Konfiguration ist über die Schnittstelle *ConfigurationProvider* gekapselt. Auch hierbei sind mehrere Varianten der Implementierung denkbar. Außerdem beschränkt dies die Änderungen auf diese Komponente, wenn Konfigurationselemente hinzukommen oder entfernt werden.

Die Schnittstellen *ResultView* und *ErrorView* repräsentieren die Anzeigen des Resultat oder der Fehlermeldung des Testaufrufs. Diese Aufgaben werden von der eigentlichen Live-UI-Komponenente an Subkomponenten delegiert. Auch dies beschränkt die Anzahl der Klassen, die bei einer Änderung am Format des Ergebnisses oder der Fehler geändert werden müssen. Generell wird, soweit möglich, gegen Schnittstellen programmiert und nicht gegen konkrete Implementierungen.

3.3 Anwendung auf den konkreten Transformationsfall

Für die Live-Konfiguration des Transformationsdienstes des ODS kann das generelle Konzept einer Live-Konfiguration angewendet und konkretisiert werden. Hierfür muss sowohl auf die in 3.1 aufgestellten Rahmenbedingungen als auch auf die Anforderungen (Kapitel 2) an den Transformationsdienst und die Weboberfläche eingegangen werden.

3.3.1 Transformationsdienst

Der Transformationsdienst ist wie alle Dienste innerhalb des ODS als Microservice konzipiert. Er ist somit ein eigenständiges Modul, das getrennt von der Weboberfläche und dem Rest des Systems betrieben und ausgeliefert werden kann. Da jeder Microservice einen eigenen Lebenszyklus besitzt, haben diese auch eigene fortlaufende Versionsnummern.

Zur Kommunikation mit dem Transformationsdienst werden, wie bei den weiteren Diensten des ODS, REST-Aufrufe über HTTP genutzt. Dies ist somit die

Konkretisierung des Transportsystems aus dem generellen Konzept und benötigt eine Implementierung. Hierfür muss ein HTTP-Server bereitgestellt werden, welcher einen entsprechenden Endpunkt zum Anstoßen der Transformation anbietet. Auch ein Endpunkt für die aktuelle Version des Transformationsystems soll implementiert werden. Wie auch im generellen Konzept soll es dennoch die Möglichkeit geben, die verwendete Transportmethode etwa durch Websockets auszutauschen.

Der Transformationsdienst muss übergebene Codeabschnitte zur Datenverarbeitung ausführen. Aufgrund der einfachen Syntax und großen Verbreitung vor allem im Bereich der Webtechnologien wird im Rahmen dieser Arbeit als Programmiersprache für die Transformationsscripte JavaScript ausgewählt. Diese Implementierung ermöglicht Datenmanipulation durch eine interpretierte Programmiersprache, eine Kompilierung ist nicht notwendig.

Bei der Ausführung der Transformationen sind die Anforderungen „Sicherheit“ (A11/A12) und „Verfügbarkeit“ (A13) zu gewährleisten. Hierfür ist es notwendig, die Ausführung der Transformation mithilfe von *Sandboxing* vom Rest des Dienstes zu trennen.

Sandbox bezeichnet hier einen abgetrennten Bereich innerhalb eines Softwaresystems, in dem kritischer Code sicher ausgeführt werden kann [Gol+96]. So wird sichergestellt, dass der Transformationscode isoliert läuft und keinen Zugriff auf kritische Systemfunktionen erhält.

Je nach gewählter Implementationssprache gibt es verschiedene Technologien für Sandboxing. Deswegen soll diese im Rahmen des Konzepts austauschbar sein. Die Wahl der konkreten Sandbox-Lösung für den Transformationsdienst diskutiert der Abschnitt 4.2 im Kapitel Umsetzung.

Ablauf eines Transformationsaufrufs

In Abbildung 3.4 wird mit einem Sequenzdiagramm beschrieben, wie die verschiedenen Objekte innerhalb des Transformationsdienst zusammenarbeiten. Nachdem der Client eine Anfrage an den Microservice abgesendet hat, wird diese vom Endpunkt behandelt. Dieser prüft zuerst, ob die Anfrage syntaktisch korrekt ist und alle nötigen Daten enthält. Daraufhin ruft der Endpunkt die entsprechende Methode am Transformationsdienst auf, und verpackt die Antwort entsprechend dem verwendeten Transportsystem (hier HTTP-REST). Im Falle einer auszuführenden Transformation benutzt die Instanz des Transformationsdienst die spezifizierte Sandbox-Implementierung und übergibt dieser die Daten und den Codeabschnitt. Da die Bestimmung der Metadaten, wie etwa Zeitmessungen, unabhängig von der gewählten Sandbox-Lösung ist, wird diese vom Transformationsdienst selbst übernommen. So ist ein Austausch des Sandboxmechanismus möglich, ohne die Erfassung der Metadaten neu implementieren zu müssen.

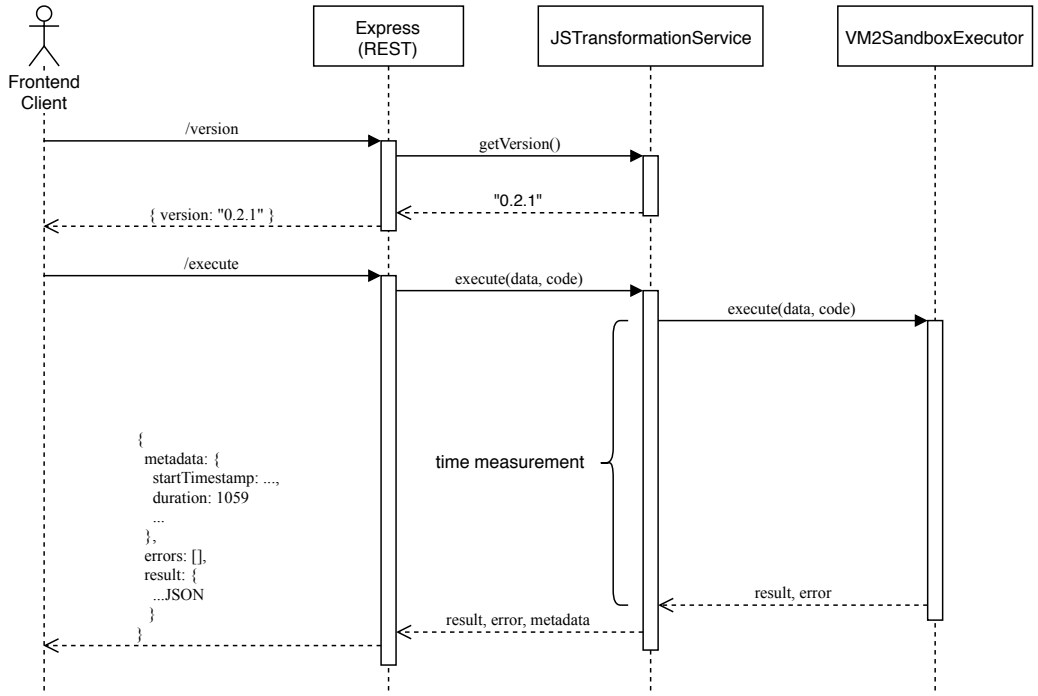

Abbildung 3.4: Ablauf von API-Aufrufen an den Transformationsdienst

Softwarearchitektur

Die Codearchitektur des Transformationsdienstes (siehe 3.5) entspricht in großen Teilen der Struktur des Dienstes im generellen Konzept. Hinzugekommen ist die konkrete Implementierung des Transportsystems und die benötigten Klassen für den Sandboxmechanismus. Dieser wird wie schon der Transportmechanismus zuvor gekapselt, um eine Änderung des Sandboxmechanismus zu vereinfachen und auf eine Klasse zu beschränken. Der Service wird nur gegen die allgemeine Schnittstelle programmiert.

Da die Bestimmung der Metadaten innerhalb der Klasse `JSTransformationService` erfolgt, wird für die Rückgabe der Ergebnisse der Sandbox eine weiteres Interface namens `ExecutionResult` benötigt.

3.3.2 Weboberfläche

Bei der Erstellung des Konzepts für die Weboberfläche ist zu beachten, dass die generischen Konfigurationselemente aus dem Transformationscode und den Beispieldaten bestehen.

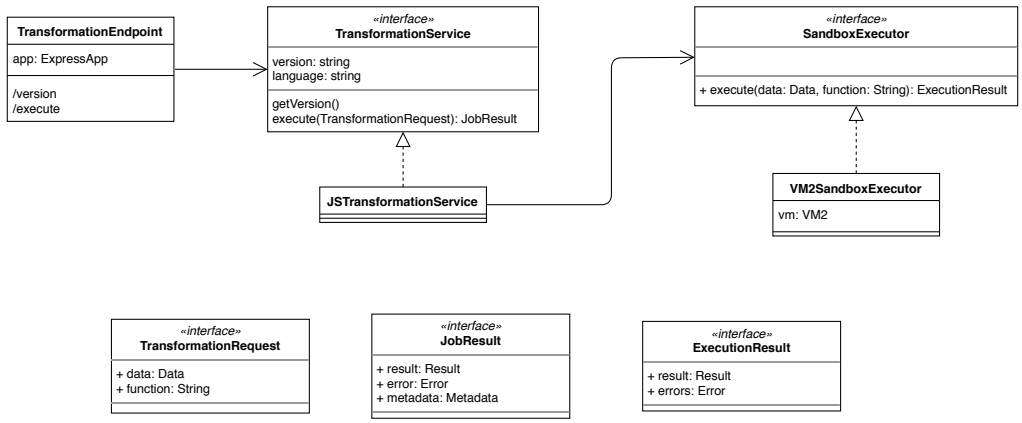

Abbildung 3.5: Geplante Codestruktur des Transformationsdienstes des ODS

Aufbau der Weboberfläche

Wie im generellen Konzept vermerkt, sollen sowohl die Beispieldaten als auch der eigentliche Transformationscode sowie das Ergebnis auf einem Blick zu sehen sein. Die Oberfläche wird daher in drei Teile gegliedert (siehe auch Abbildung 3.6:

- Datenquelle
- Codeeditor
- Ergebnis- bzw. Fehleranzeige

Im ersten Teil werden die zu transformierenden Beispieldaten festgelegt. Dies geschieht über eine simple Texteingabefläche, in welche JSON-Daten geschrieben werden können. Wenn auch in der Anforderungen nicht aufgeführt, kann es auch hier wünschenswert sein, aus IDEs bekannte Features wie Syntax-Hervorhebung oder Codevervollständigung zu unterstützen. Alternativ zur direkten Eingabe der Beispieldaten sind auch andere Implementierungen denkbar. Wie in Abschnitt 1.1 beschrieben, werden die Daten bei der Ausführung der Pipeline durch den ODS von Adaptern bereitgestellt, die diese beispielsweise von einem Webservice abrufen. Diese Funktionalität könnte auch bei der Konfiguration des Transformationsdienstes simuliert werden, um ein realitätsnahes Testen der Transformation mit echten Daten zu ermöglichen.

Die Eingabe der Datenmanipulationsschritte erfolgt in einem Codeeditor. In einer einfachen Version der Oberfläche kann dieser auch als HTML-TextArea realisiert sein. Um die Anforderungen nach Codevervollständigung und Syntaxhervorhebung zu erfüllen, wird jedoch eine externe Bibliothek verwendet, welche diese

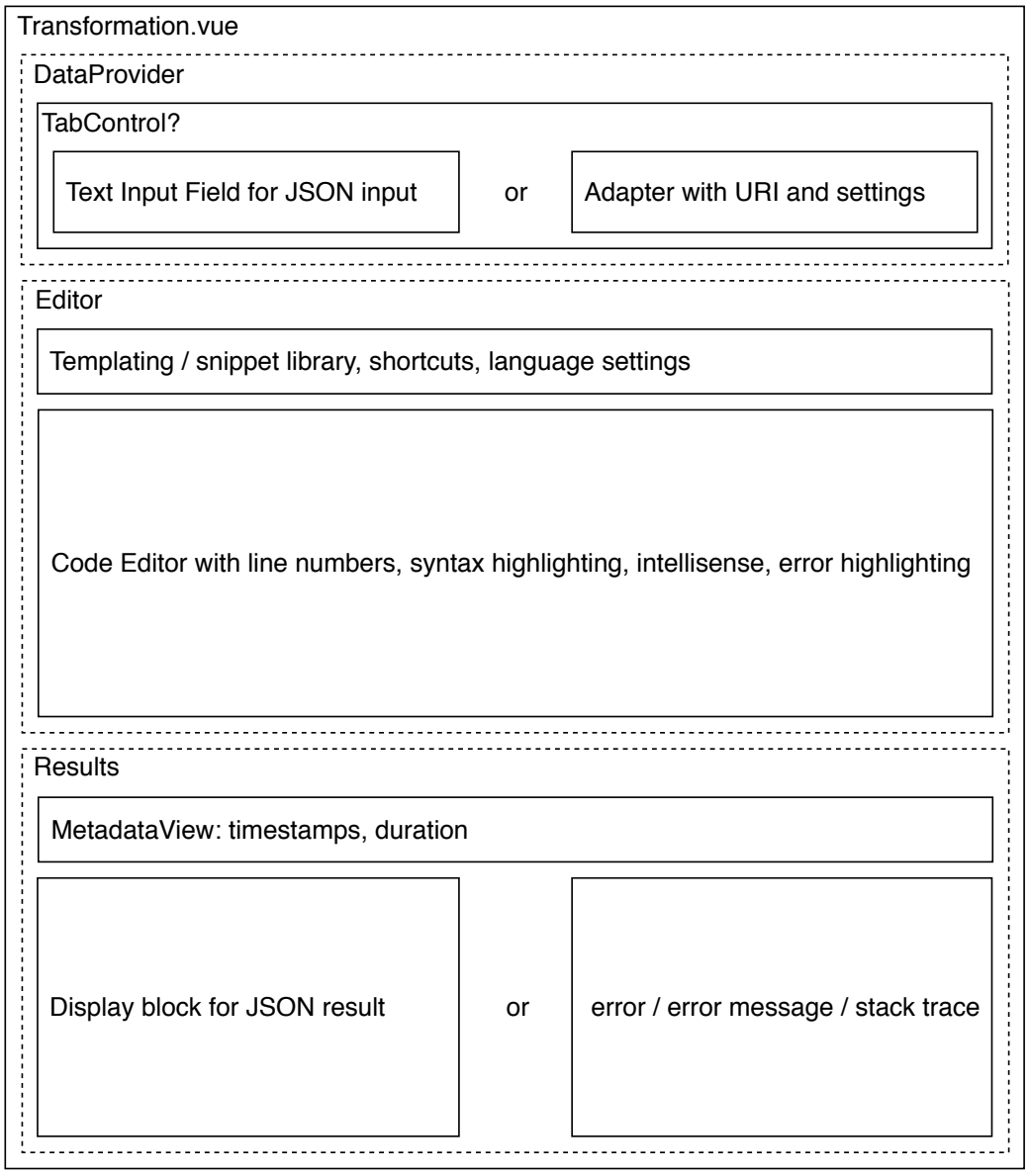

Abbildung 3.6: Geplanter Aufbau der Weboberfläche

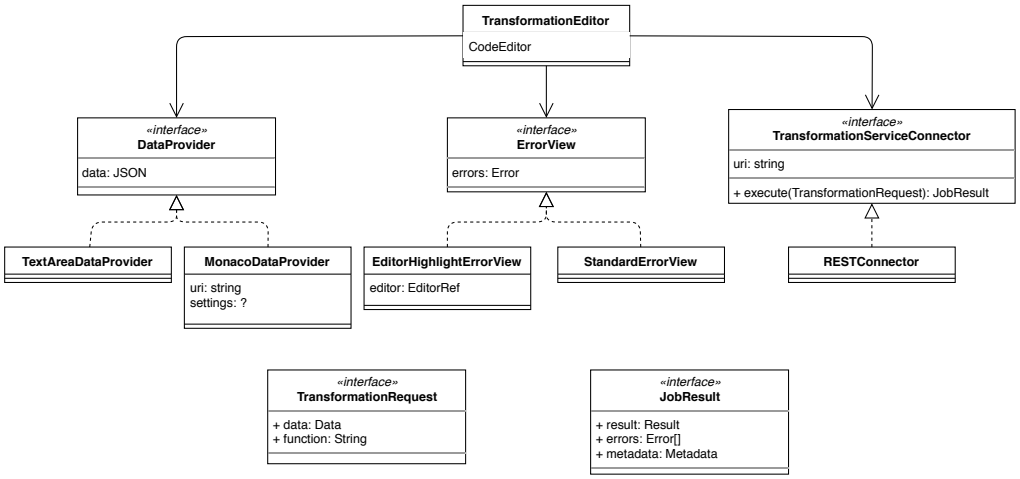

Abbildung 3.7: Geplante Codestruktur der Weboberfläche

Features bereitstellt. Mit dem Vergleich verschiedener Angebote und der Begründung der Auswahl beschäftigt sich Abschnitt 4.3.

Im unteren Teil der Oberfläche wird das Ergebnis der Transformation angezeigt. Über dem Ergebnis werden die in der Antwort erhaltenen Metadaten wie etwa Ausführungsdauer und -zeitpunkt aufgelistet. Falls ein Fehler aufgetreten ist, wird statt der Ergebnisvorschau die Fehlermedlung in ihrer rohen Form angezeigt. Eine weitere Anzeige des Fehlers geschieht innerhalb des Codeeditors, in dem die betroffene Zeile markiert wird. Der Codeeditor erfüllt hier also eine Doppelrolle und implementiert zwei Schnittstellen. Zum einen stellt er den auszuführenden Code zur Verfügung, zum anderen zeigt er entstandene Fehler an (`ErrorView`).

Softwarearchitektur

Die Softwarearchitektur der Weboberfläche (Abbildung 3.7) bildet die gerade aufgezählten, verschiedenen Möglichkeiten der Dateneingabe und der Anzeige von Fehlern nach.

Die Eingabe der Daten erfolgt über Implementierungen des Interface `DataProvider`, welche etwa eine TextArea oder eine externen Editorkomponente nutzen. Neben der grundlegenden Anzeige von Fehlern direkt in der Oberfläche wird durch `EditorHighlightErrorView` die Anzeige der Fehler direkt im Codeeditor repräsentiert.

Auch in der Weboberfläche wird über das verwendete Transportsystem abstrahiert, beispielhaft wird die Verwendung von HTTP-REST-Aufrufen implementiert.

4 Umsetzung

Die Umsetzung des in Kapitel 3 erstellten Konzepts erfolgt nicht in einem großen Block, sondern iterativ. Iterative Entwicklung beschreibt ein Vorgehen, bei dem eine Software in kleinen, voneinander getrennten Schritten verbessert wird. Agile Softwareentwicklungsprozesse stützen sich ebenfalls auf iterative Verfahren [Bec03]. Es ist zu beachten, dass die Software möglichst nach jedem Entwicklungsschritt lauffähig und benutzbar ist, sowie einen Mehrwert gegenüber der vorherigen Version besitzt.

Aufgrund der Abhängigkeiten zwischen Transformationsdienst und Weboberfläche bei bestimmten Anforderungen, wie z.B. der Erstellung und Anzeige der Fehlermeldungen, erfolgt die Implementierung an beiden Komponenten gleichzeitig. Dieses Kapitel ist somit nicht wie Kapitel 2 zwischen Dienst und Webinterface aufgeteilt, sondern nach den einzelnen Implementierungsschritten und Features chronologisch geordnet.

Abbildung 4.1 zeigt die Arbeitspakete, in welche die Arbeit an der Live-Konfiguration aufgeteilt wird und welche Anforderungen jeweils beachtet und implementiert werden. Die Abschnitte 4.8 und 4.9 beschreiben orthogonal zu diesen Arbeitspaketen, wie die Qualität der Arbeit mittels manuellen und automatisierten Tests sichergestellt wurde.

4.1 P1: Minimum Viable Product

Bei iterativen Entwicklungsmethoden ist der erste Arbeitsschritt das Erstellen eines Minimum Viable Product (MVP) [Rie14]. Ein MVP ist die einfachste Implementierung der Software, die von einem (gegebenenfalls imaginären) Kunden benutzt werden könnte und diesem einen Mehrwert bietet. Es unterstützt nur ausgewählte Basisfeatures und muss nicht den Reifegrad eines letztendlichen Produkts besitzen. Die Entwicklung eines MVP zielt vor allem auf die frühe Rückmeldung des Kunden ab. Je früher Rücksprachen zu einer Änderung von Anforderungen führen, desto einfacher sind diese umzusetzen.

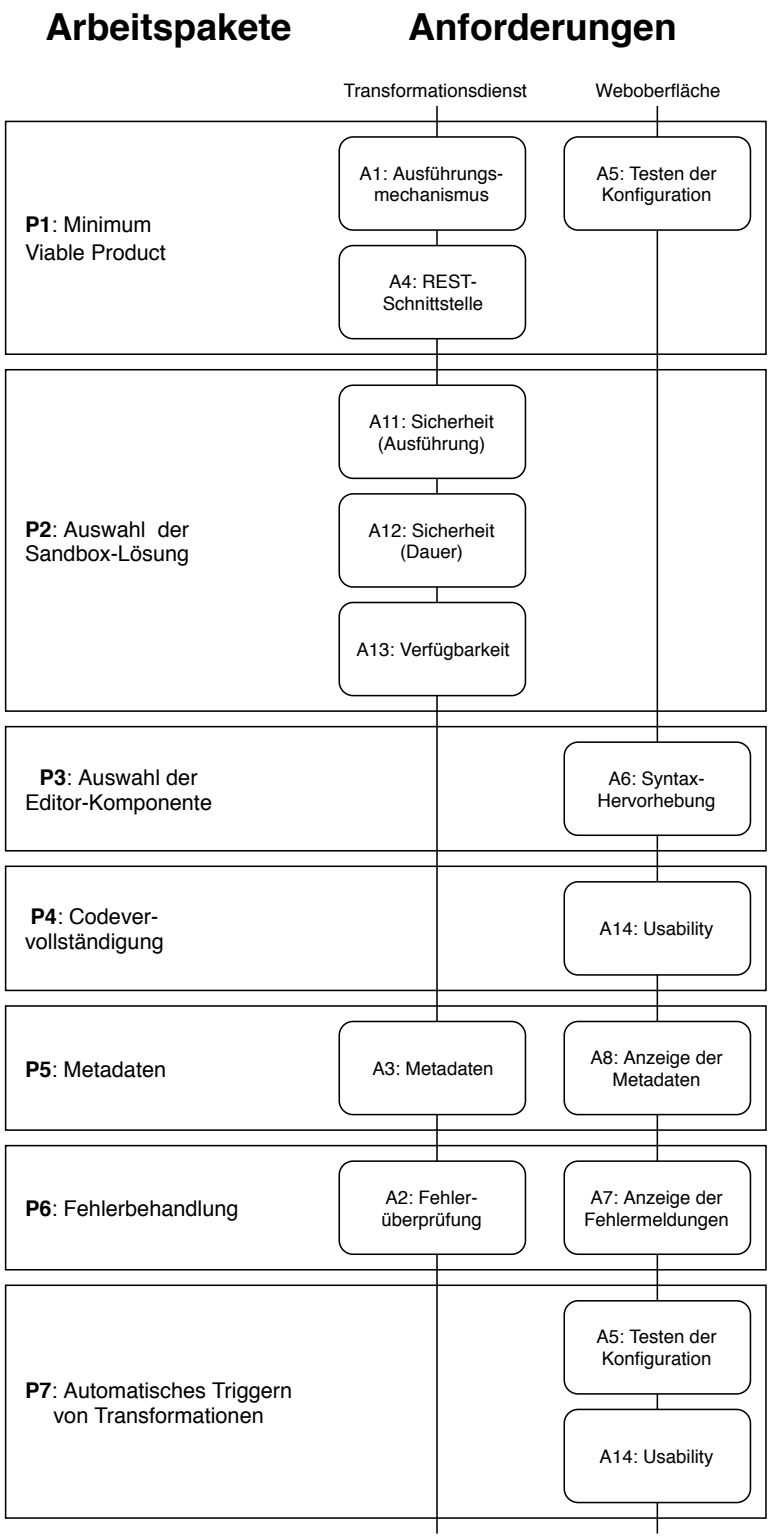

Abbildung 4.1: Arbeitspakete

Da sich der Open Data Service (ODS) derzeit in Entwicklung befindet, steht zu Beginn dieser Arbeit bereits eine grundlegende Implementierung des Transformationsdienstes zur Verfügung. Diese wurde in JavaScript erstellt und läuft auf der Node.js-Plattform. Node.js ermöglicht es, auch serverseitige Anwendungen mit JavaScript zu erstellen [Nod]. Zur Ausführung des JavaScript-Codes nutzt Node.js die Laufzeitumgebung „V8“. V8 nutzt Just-in-Time-Kompilierung und lieferte hierzu zum Zeitpunkt der Veröffentlichung von Node.js eine erhöhte Performance im Vergleich zu anderen JavaScript-Implementierungen [PW15]. Node.js benutzt eine ereignisgesteuerte Architektur. Die Ausführung von Code findet hierbei in einem einzelnen Thread statt.

Zuerst wird die bestehende Implementierung in die Programmiersprache TypeScript überführt. TypeScript ist eine von Microsoft entwickelte Programmiersprache (Apache-Licenz 2.0), welche JavaScript um ein strenges Typsystem erweitert [Micb]. TypeScript wird vor der Ausführung von einem Compiler in JavaScript übersetzt. Vor allem das Typsystem ermöglicht es, einfacher wartbare Anwendungen zu entwickeln.

Für die Konvertierung des Dienstes müssen der TypeScript-Compiler sowie die Typdefinitionen der verwendeten Pakete (bspw. `@types/express` für das benutzte Web-Framework Express) als Abhängigkeiten hinzugefügt werden. Der Compiler wird mithilfe der Datei `tsconfig.json` konfiguriert. Darüber hinaus wird mit Jest[1] ein Testing-Framework, welches TypeScript unterstützt, hinzugefügt. Jest übernimmt das Ausführen der Unit- und Integrationstests, welche im Rahmen der Implementierung des Transformationsdienstes angelegt werden. Mit ESLint[2] wird das Projekt um einen Linter ergänzt. Linter unterstützen Entwickler mit einer statischen Code-Analyse und werden im JavaScript-Umfeld zumeist für die Einhaltung von Codingstandards benutzt.

Ohne zusätzliche Werkzeuge ist es bei der Entwicklung mit TypeScript notwendig, nach jeder Änderung am Programmcode diesen erneut zu kompilieren und daraufhin das Programm neu zu starten. Dies kann jedoch vereinfacht werden, indem der „watch“-Modus des TypeScript-Compilers benutzt wird, welcher nach Änderungen automatisch einen neuen Kompilierdurchgang anstößt. Zusammen mit dem Werkzeug Nodemon[3], welches JavaScript-Anwendungen bei Änderungen am Code neu startet, kann ein flüssiges Arbeit gewährleistet werden. Die gleichzeitige Ausführung des Compilers und Nodemon wird der Bequemlichkeit wegen als Skript in der `package.json` des Projekts hinterlegt.

Das Umschreiben des Programmcodes zu TypeScript ist simpel, da TypeScript ein Superset von JavaScript ist, das heißt valider JavaScript-Code ist auch vali-

[1] `https://jestjs.io`
[2] `https://eslint.org`
[3] `https://nodemon.io/`

der TypeScript-Code. Um die Vorteile von TypeScript nutzen zu können, müssen Typdefinitionen an Funktionen und Variablen angefügt werden. Darüber hinaus werden die in Kapitel 3 konzipierten Schnittstellen wie `TransformationService` erstellt. Die bisherige Programmlogik wird als rudimentäre Implementierung dieser Interfaces wiederbenutzt.

Die Webkomponente zur Konfiguration des ODS ist in Vue.js[4] realisiert. Vue.js ist ein populäres Webframework zur Entwicklung von Single-Page-Webanwendungen. Es stellt hierbei eine Alternative zu den populären Frameworks Angular und React dar, welche von Google respektive Facebook entwickelt werden. Wie die Mitbewerber basiert Vue.js auf dem Entwurfsmuster Model View ViewModel (MVVM) [Gar11]. Mithilfe von MVVM werden die Verwaltung des Applikationszustands und die Anzeigeschicht strikt getrennt.

Durch die gezeigte Architektur ist eine Einbettung der Komponente in den existierenden Workflow möglich. Für die Implementierung wurde die Komponente jedoch gesondert entwickelt, um Wechselwirkungen mit anderen Arbeiten an der Konfigurationsoberfläche zu vermeiden.

Die Komponente für die Live-Konfiguration stellt nur die Basisfunktionalität bereit. Mittels zweier einfacher Texteingabefeldern können die Daten und der Codeabschnitt gesetzt werden. Hierbei stehen keine Codevervollständigung oder Syntaxhervorhebung zu Verfügung. Die Transformation kann durch einen Button manuell angestoßen werden. Der Code ist entsprechend der Codearchitektur aus Kapitel 3 strukturiert. Die beispielhafte Implementierung des `TextAreaDataProvider` und das Eingabefeld für den Transformationscode werden in den nächsten Arbeitsschritten im Hinblick auf die weiteren Anforderungen ergänzt.

4.2 P2: Auswahl der Sandbox-Lösung

Wie in Abschnitt 3.3.1 beschrieben, wird innerhalb des Transformationsdienstes eine Sandbox benutzt, um den vom Nutzer übergebenen Code vom Rest des Dienstes zu trennen. Hiermit wird sichergestellt, dass Transformationscode nicht den gesamten Dienst durch den Aufruf von Systemfunktionen lahmlegt (A13), wie z.B. durch einen Aufruf von `process.exit`.

Der Sandboxmechanismus muss folgende Anforderungen erfüllen:

- Aufruf von Systemfunktionen blockieren
- Einbinden von externen Bibliotheken verhindern
- Maximale Skriptlaufzeit begrenzen

[4] `https://vuejs.org/`

Name	`eval`	**vm**	**vm2**	**Container**
Zugehörigkeit	JavaScript-Sprachfeature	Node.js-Modul	Open-Source-Bibliothek	verschieden
separater Kontext	×	✓	✓	✓
Sicherheits-features	×	×	✓	✓
seperater Prozess	×	×	×	✓

Tabelle 4.1: Vergleich von Mechanismen zur Ausführung von JavaScript-Code

Es stehen verschiedene Möglichkeiten zur Ausführung von Nutzercode mit und innerhalb von Node.js zur Verfügung. Nachfolgend werden in diesem Abschnitt vier Varianten vorgestellt und entsprechend der Anforderungen evaluiert.

Der Vollständigkeit halber sei `eval` erwähnt. Diese Funktion, die ähnlich auch in anderen Skriptsprachen wie beispielsweise Python verfügbar ist, führt als String übergebenen Code aus. Hier ist zu beachten, dass `eval` keinerlei Sicherheitsmechanismen eingebaut hat. Der ausgeführte Code hat vollen Zugriff auf alle Variablen oder globale Objekte wie z.B. `process`. Das Laden von Bibliotheken mit `require` unterliegt keinen Beschränkungen und die Skriptlaufzeit ist unbegrenzt. Die Verwendung von `eval` ist daher für den hier betrachteten Anwendungsfall unangebracht.

Mit dem Modul **vm**[5] bietet Node.js eine Programmierschnittstelle zu der Laufzeitumgebung V8. Mit vm lässt sich JavaScript-Code innerhalb eines separaten Kontexts ausführen. Hierbei hat der Code nur Zugriff auf ein anderes globales Objekt als der restliche Programmcode. Die Dokumentation des vm-Moduls bezeichnet dieses jedoch explizit als kein Sicherheitsfeature: „The vm module is not a security mechanism. Do not use it to run untrusted code.“ [Nod]. Diese Aussage ist richtig, da innerhalb von vm ausgeführter Code auf das externe globale Objekt zugreifen kann, indem die Konstruktoren der übergebenen JavaScript-Objekte benutzt werden (siehe Abschnitt 5.2.3).

Die **vm2**-Bibliothek[6] basiert auf dem vm-Modul und nutzt das JavaScript-Sprachkonstrukt der Proxies, um ein Ausbrechen aus der Sandbox zu verhindern. Die unter der MIT-Lizenz bereitgestellte Bibliothek wird von einem Mitarbeiter der Firma Integromat aktiv entwickelt und betreut. vm2 ist ein aktives Projekt auf GitHub und mit über 150 bearbeiteten Issues und 50.000 Downloads pro Monat die Sandbox-Lösung für Node.js, welche am aktivsten gepflegt wird.

[5] `https://nodejs.org/api/vm.html`
[6] `https://github.com/patriksimek/vm2`

Mehrere Blogartikel, die sich mit dem Thema des Sandboxing in JavaScript beschäftigen, schlagen die Verwendung von vm2 in Kombination mit **containerbasiertem Sandboxing** vor. vm2 bietet keine hundertprozentige Sicherheit, da der Code im selben Prozess wie der restliche Programmcode ausgeführt wird. Das Auslagern der Ausführung in eigene Container kann als Backup-Lösung dienen, falls ein Ausbruch gelingt und etwa der Dienst zum Absturz gebracht wird.

Im Rahmen dieser Arbeit wird trotzdem nur vm2 ohne weitere Mechanismen genutzt. Die umfassende Abhärtung des Transformationsdienstes ist nicht das Hauptziel der Arbeit, sondern die Entwicklung der Live-Konfiguration. Außerdem ist der ODS derzeit nicht als offener Dienst im Web zugänglich, sondern nur als System für Entwickler gedacht, um ihren eigenen Code auszuführen.

4.3 P3: Auswahl einer geeigneten UI-Komponente zur Bearbeitung von Skripten

Für die benutzerfreundliche Eingabe des Transformationscodes soll die Editorkomponente Syntaxhervorhebung, das Markieren von Codestellen und im Idealfall auch Codevervollständigung anbieten (siehe u.a. A6).

Anstatt dies manuell zu implementieren, kann auf eine Open-Source-Komponente zurückgegriffen werden, welche diese Features bereits unterstützt. Es stehen mehrere Editorkomponenten zur Auswahl, welche in JavaScript geschrieben und für Weboberflächen geeignet sind. Dieser Abschnitt vergleicht vier populäre Exemplare und begründet die finale Auswahl.

CodeFlask[7] ist ein simpler Codeeditor, der vor allem für kleine Codeabschnitte und interaktive Beispiele gedacht ist. Das Projekt unter der MIT-Lizenz wird von Claudio Holanda auf GitHub entwickelt[8]. Der Hauptfokus bei der Entwicklung liegt auf einfacher Benutzung und geringer Größe, um Ladezeiten zu verringern. CodeFlask basiert auf der Bibliothek prism.js, welche Syntaxhervorhebung für Sprachen wie HTML, JavaScript und CSS anbietet. Codevervollständigung wird jedoch nicht unterstützt, ebenso nicht das Markieren von Codestellen oder -zeilen.

Im Gegensatz zu CodeFlask wird die Entwicklung des Codeeditors **CodeMirror**[9] von mehreren Entwicklern übernommen. Das Projekt steht ebenfalls unter der MIT-Lizenz. CodeMirror wird in den Entwicklerwerkzeugen der Browser Firefox, Safari und Chrome benutzt. CodeMirror unterstützt Syntaxhervorhebung für über 100 verschiedene Programmiersprachen und ermöglicht es auch, Code-

[7] `https://kazzkiq.github.io/CodeFlask/`
[8] `https://github.com/kazzkiq/CodeFlask`
[9] `https://codemirror.net/`

Name	**CodeFlask**	**CodeMirror**	**Ace**	**Monaco**
Zugehörigkeit	-	Browser Dev-Tools	Cloud9	Microsoft Visual Studio Code
Lizenz	MIT	MIT	BSD	MIT
Syntax-hervorhebung	✓	✓	✓	✓
Code-hervorhebung	×	✓	✓	✓
Codevervoll-ständigung	×	✓	✓	✓

Tabelle 4.2: Vergleich von JavaScript-Editorbibliotheken

vervollständigung zu implementieren. Dies muss jedoch manuell erfolgen (eine Version für JavaScript ist bereits mitgeliefert). Auch das Markieren von Zeilen ist mit CodeMirror möglich.

Die Online-IDE Cloud9[10], welche im Juli 2016 von Amazon Web Services übernommen wurde[The], basiert auf dem Codeditor **Ace**[11]. Der Ace-Editor steht unter der BSD-Lizenz. Das Hauptfeature des Editors ist die Kompatibilität mit dem Syntaxhervorhebungsystem von TextMate und Sublime Text. Mit diesem unterstützt Ace Syntaxhervorhebung für über 110 Programmiersprachen sowie Syntaxkontrolle für JavaScript, CoffeeScript, CSS und Xquery. Für diese Sprachen ist auch Codevervollständigung implementiert und auch das Markieren von Codezeilen ist möglich.

Schließlich stellt Microsoft mit **Monaco**[12] den Editor bereit, welcher in dem Desktop-Editor Visual Studio Code[13] (VS Code) benutzt wird. Monaco ist MIT-lizensiert. Wie auch VS Code unterstützt Monaco Syntaxhervorhebung, Markieren von Codezeilen sowie Codevervollständigung. Neben den Sprachen JavaScript, TypeScript, HTML und JSON unterstützt Monaco auch das Language Server Protocol (LSP). Das von Microsoft entwickelte LSP wird zur Verbindung von Codeditoren oder IDEs mit einen Language-Server verwendet [Mica]. Ein Language-Server übernimmt hierbei die Aufgabe, Funktionen wie Codevervollständigung oder das Springen zu Definitionen bereitzustellen. Dieses Konzept soll den Arbeitsaufwand für die Integration von verschiedenen Editoren mit Programmiersprachen verringern, indem durch das LSP ein gemeinsamer Standard genutzt werden kann. Die Dokumentation von Monaco bietet mehrere Codebei-

[10] `https://aws.amazon.com/cloud9/`
[11] `https://ace.c9.io/`
[12] `https://microsoft.github.io/monaco-editor/`
[13] `https://code.visualstudio.com/`

spiele, welche beispielsweise das Hervorheben von Codezeilen erläutern. Monaco bietet von sich aus Typdefinitionen für TypeScript, außerdem existieren mehrere Pakete, welche das Einbinden des Editors in Vue.js-Anwendungen erleichtern.

Tabelle 4.2 gibt nocheinmal einen Überblick über die gerade erwähnten Editoren und ihren Features. Für die Verwendung in der Weboberfläche zur Live-Konfiguration wird der Monaco-Editor ausgewählt. Als entscheidende Kriterien sind hier die gute Dokumentation mit Beispielcode, die große Community und die sehr aktive Entwicklung zu nennen. Außerdem ist die Möglichkeit, den Editor mit Language-Servern zu verknüpfen, sehr mächtig. Sofern auf der Serverseite eine Implementierung eines Language-Servers für die entsprechende Programmiersprache bereit steht, lässt sich die Unterstützung von anderen Programmiersprachen leicht einbauen.

4.4 P4: Codevervollständigung

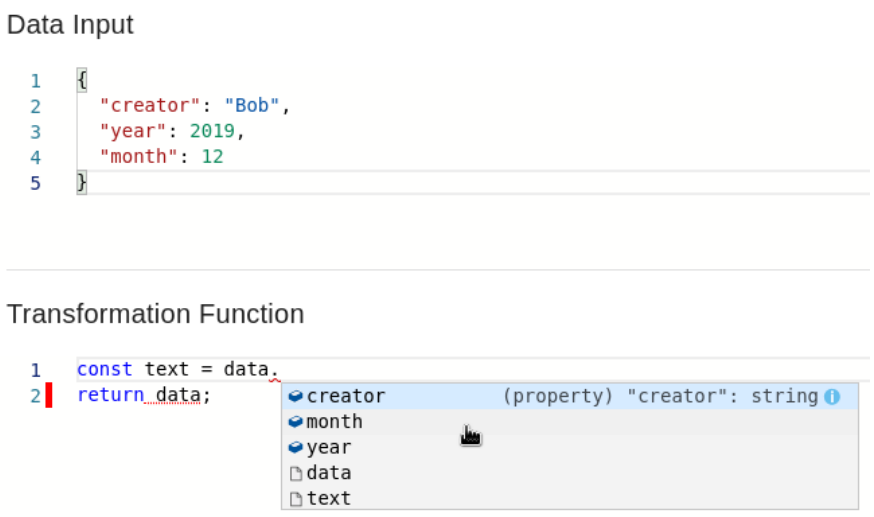

Abbildung 4.2: Automatische Codevervollständigung

Der Codeabschnitt, welcher bei der Konfiguration des Transformationsdienst eingegeben wird, beschreibt die Anweisungen, mit denen die Daten manipuliert werden. Innerhalb des Dienstes wird hierfür der übergebene Code in eine Funktion eingefügt und diese Funktion mit den Beispieldaten ausgeführt. Der Parameter, welcher die Beispieldaten enthält, heißt `data`.

Für das Erstellen des Transformationscode ist es hilfreich, dass für diesen Parameter Codevervollständigung zur Verfügung steht. Diese kann hier Tippfehler vermeiden und die Entwicklung beschleunigen.

Da die Beispieldaten innerhalb der Weboberfläche bekannt sind, ist es möglich, diese dem Codeeditor zur Verfügung zu stellen. Der JavaScript-Language-Server des Monaco Editors unterstützt das Hinzufügen von Bibliotheken. Diese Bibliotheken können jedwede Art von JavaScript-Code enthalten. Es ist somit möglich, eine Bibliothek hinzuzufügen, welche `data` als globales Objekt bereitstellt. Die Inhalte von `data` werden dann automatisch erfasst und für die Codevervollständigung bereitgestellt.

In der Implementierung ist zu beachten, dass die benutzte Funktion `addExtraLib` immer ein neue Bibliothek hinzufügt, aber ein Objekt zurückliefert, mit welchem diese wieder entfernt werden kann. Es ist notwendig, dieses Objekt zu speichern. Bei jeder Aktualisierung der Beispieldaten kann damit die alte Bibliothek entfernt werden. Erst danach werden die neuen Daten hinzugefügt.

4.5 P5: Metadaten

Bei der Ausführung einer Transformation sollen neben dem Ergebnis auch Metadaten vom Transformationsdienst zurückgeliefert werden (vgl. Anforderung A3 und A8). In diesem Arbeitsschritt werden verschiedene Optionen von Metadaten auf ihren Nutzen und Umsetzbarkeit hin evaluiert. Außerdem wird die Implementierung der ausgewählten Metadaten erläutert.

Die Metadaten sollen dem Nutzer bei der Konfiguration der Pipeline helfen. Es sind zum einen allgemeine Informationen nützlich, wie etwa der letzte Ausführungszeitpunkt, um einen Überblick über die Ausführung zu behalten. Darüber hinaus können Auskünfte über Ausführungsdauer und Ressourcennutzung helfen, Performanceprobleme zu finden und zu vermeiden.

4.5.1 Zeitmessung

Mithilfe von `Date.now` lässt sich in TypeScript die aktuelle Systemzeit als Timestamp bestimmen. Dies geschieht vor und nach der Ausführung der Transformation durch den `SandboxExecutor` und beide Timestamps werden zu den Metadaten hinzugefügt.

Für die Bestimmung der Ausführungsdauer wird jedoch eine andere Funktion benutzt, denn die Genauigkeit von `Date.now` ist auf Millisekunden beschränkt. Dies ist ausreichend für einen Zeitpunkt, für die Dauer ist jedoch eine höhere Genauigkeit wünschenswert. Node.js stellt mit der Methode `process.hrtime` eine Möglichkeit bereit, Zeit mit einer Auflösung von Nanosekunden abzufragen. Es wird ein Array im Format `[seconds, nanoseconds]` zurückgegeben, da normale JavaScript-Variablen nicht die nötige Auflösung bereitstellen. Die Methode

ist speziell für das Messen von Zeitintervallen konzipiert. Wird einem wiederholten Aufruf von `process.hrtime` das Ergebnis eines früheren Aufrufs übergeben, wird automatisch die Differenz berechnet. Der Transformationsdienst benutzt diese Methode, um die Ausführungsdauer der Transformation zu bestimmen, und liefert diese als Teil der Metadaten zurück.

4.5.2 Ressourcennutzung

Mit `process.memoryUsage` bietet Node.js eine Methode an, welche die derzeitige Arbeitsspeichernutzung des Node.js-Prozesses zurückliefert. Es werden vier verschiedene Werte zurückgeliefert:

- `heapUsed`: Misst die Größe aller Objekte, die sich derzeit auf dem Heap befinden.
- `heapTotal`: Gibt die Gesamtgröße des reservierten Speichers für den Heap an.
- `external`: Beschreibt die Größe des Speichers für externe C++-Objekte, die von der Laufzeitumgebung V8 verwaltet werden.
- `rss`: Beschreibt die Größe des „Resident Set“, welches den gesamten Heap, den Stack und den geladenen JavaScript-Code enthält

Wird die Abfrage des Speicherverbrauchs in die Transformatiosdurchführung einfügt, liefert diese korrekterweise einen gewissen Basisspeicherverbrauch des Dienstes zurück. Bei aufwendigen Tranformationen, die viele Daten enthalten (als Test wird das Invertieren eines Arrays mit 10 Millionen Einträgen genutzt), wird ebenfalls ein erhöhter Speicherverbrauch gemessen. Es ist jedoch zu beachten, dass das Speichermanagement von Node.js mithilfe von einem Verfahren zur Garbage-Collection funktioniert. Hierbei wird benutzter Speicherplatz nicht manuell vom Entwicker wieder freigegeben, sondern wird regelmäßig von der Garbage-Collection übernommen. So kann es vorkommen, dass nachfolgende Aufrufe von Transformationen einen erhöhten Speicherbedarf melden, auch wenn dieser von einer unabhängigen Tranformation stammt. Auch ein Messen des Speicherbedarfs vor und nach Ausführung des Codes ergibt kein besseres Ergebnis, da hier speicherintensive Aufrufe geringen Speicherverbrauch anzeigen können. Aufgrund dieser Einschränkungen wird die Speicherbenutzung nicht in die Metadaten des Transformationsaufrufs eingebaut.

Node.js bietet mit der Funktion `process.cpuUsage` die Möglichkeit, die verstrichene CPU-Zeit für Nutzer- und Systemcode zu bestimmen. Hierzu kann der Methode das Ergebnis eines früheren Aufrufs übergeben werden, analog zu `process.hrtime`. Die Unterscheidung zwischen Nutzer- und Systemcodezeit kann hilfreich sein, Flaschenhälse bei der Performance von Code zu entdecken. Ein großer Wert

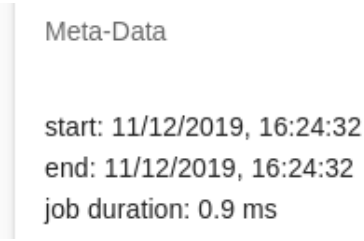

Abbildung 4.3: Anzeige der Metadaten mithilfe von Vue.js-Filtern

bei der Systemzeit zeigt, dass das System viel auf Netzwerk- oder Dateioperationen warten muss, während ein hoher Wert bei der Nutzerzeit zeigt, dass die angestellten Operationen und Berechnungen sehr aufwendig sind.

Da in den Codeabschnitten für die Transformation der Zugriff auf Systemfunktionen, die Operationen wie Dateizugriff unterstützen, jedoch nicht erlaubt wird, liefert das Abfragen der CPU-Zeiten wenig Erkenntnisse für den Transformationdienst. Auch ein Vergleich der verstrichen CPU-Zeit mit der Ausführungsdauer ist nicht aufschlussreich, da die Node.js-Umgebung mit nur einem einzigen Thread ausgeführt wird. Daher ist das Verhältnis zwischen diesen beiden Werten immer **1**. Die Methode `process.cpuUsage` bietet somit keinen Mehrwert für die Metadaten des Transformationsaufrufs.

Trotz dieser Vielzahl an Möglichkeiten bleiben die Metadaten auf Ausführungszeitpunkt und -dauer beschränkt.

4.5.3 Anzeige in der Weboberfläche

Wie in A8 beschrieben, müssen die Metadaten auch in der Weboberfläche angezeigt werden. Dies geschieht im `ResultView`, welcher auch die Anzeige des Ergebnisses bzw. der Fehlermeldung übernimmt (siehe Abbildung 4.3).

Das verwendete JavaScript-Framework Vue.js bietet mit *Filtern* eine bequeme Möglichkeit, Daten zu formatieren. Für die Anzeige der Metadaten werden dafür die Filter `timestamp` und `duration` in einer extra Datei implementiert und mithilfe von Imports für den `ResultView` zur Verfügung gestellt. Im Filter `timestamp` wird die JavaScript-interne Methode `toLocaleString` benutzt, wobei dieser per Parameter das gewünschte Datums- und Zeitformat (`numeric`) übergeben werden muss.

4.6 P6: Fehlerbehandlung

Für die Erfüllung der Anforderung A2 muss der Transformationsdienst aussagekräftige Fehlermeldungen zurückgeben.

4.6.1 Fehlerquellen

Bei der Ausführung der Transformation können an mehreren Stellen Fehler auftreten. Hierbei wird vorausgesetzt, dass die Anfrage vom Transportsystem akzeptiert wird und somit syntaktisch richtig ist, im konkreten Fall des HTTP-REST Endpunkts also valides JSON ist.

Bei der Benutzung des Sandbox-Mechanismus mit vm2 wird der übergebene Code zuerst in ein V8-internes Format kompiliert und dann ausgeführt. Sollte der Code Syntaxfehler enthalten, werden diese beim Kompilieren abgefangen. Fehler wie die Verwendung von undeklarierten Variablen treten bei JavaScript erst bei der Ausführung auf. Da jede Transformation Daten zurückliefern muss, ist ein undefinierter Rückgabewert, wie er durch das Fehlen eines `return`-Statements auftreten kann, ebenfalls als Fehlerfall zu behandeln.

4.6.2 Fehlerübertragung

Bei der Implementierung der Übertragung von Fehlermeldungen wird zuerst versucht, das von der JavaScript-Laufzeitumgebung zurückgelieferte `Error`-Objekt direkt zu verwenden. Hierbei treten jedoch zwei Probleme auf. Da die Fehler innerhalb der Sandbox auftreten, die einen anderen Kontext als der Rest des Codes benutzt, sind die zurückgegebenen Objekte Instanzen eines anderen Prototyps als erwartet. Vergleiche mit dem Operator `typeof` schlagen deshalb unerwartet fehl. Darüber hinaus lassen sich Instanzen des internen JavaScript-Objekts `Error` nicht mithilfe von `JSON.stringify` in JSON umwandeln.

Die Fehler müssen also vor der Übertragung in ein internes Format überführt werden. Dieses Format enthält neben dem Namen des Fehlers auch eine fehlerspezifische Nachricht, die Zeilennummer und Position sowie gegebenenfalls einen Stacktrace.

Bei der Umwandlung ist zu beachten, dass Kompilierungs- und Laufzeitfehler verschiedene Formate haben. Sie müssen getrennt analysiert werden, um die korrekte Zeilenummer und -position aus der Fehlermeldung zu extrahieren. Da nach dem Prinzip der Kapselung die internen Vorgehensweisen des Transformationsdienstes vor dem Nutzer verborgen werden sollen, wird bei der Behandlung der

Fehler der Stacktrace bearbeitet. Alle Einträge, welche sich auf interne Module des Dienstes, der Sandbox oder Node.JS beziehen, werden gefiltert, sodass nur die Einträge für den übergebenen Nutzercode übrig bleiben. Außerdem werden die Zeilennummern dieser Einträge korrigiert, da der Code vor der Ausführung mit einer Hilfsfunktion umgeben wird.

Bei der Behandlung des Fehlers, falls die Transformationsfunktion keine Daten zurückliefert, muss eine manuelle Fehlermeldung erzeugt werden. Als Zeilennummer wird hierbei die letzte Zeile festgelegt, da hier der `return`-Befehl erwartet wird.

4.6.3 Anzeige in der Weboberfläche

Die im Transformationsdienst erzeugten Fehlermeldungen müssen auch in der Weboberfläche angezeigt werden. Zum einen geschieht dies durch die simple Anzeige des Fehlers im JSON-Formats an der Stelle, an der sonst das Ergebnis der Transformation stehen würde. Hierdurch wird sofort sichtbar, dass ein Fehler aufgetreten ist.

Darüber hinaus wird entsprechend der Anforderung 2.1.2 der Fehler auch an der entsprechenden Codestelle markiert (siehe Abbildung 4.4). Der Monaco-Codeeditor bietet hierfür das Feature der Dekorationen an. Dekorationen können benutzt werden, um eigene CSS-Klassen auf bestimmte Teile des Codes anzuwenden. Dies kann sowohl innerhalb einer Zeile als auch für die gesamte Zeile auf einmal geschehen. Das Einfügen und Entfernen der Dekoration wird über die Methode `deltaDecorations` gesteuert. Jeder Aufruf dieser Methode liefert eine Liste von Identifiers (IDs) zurück, welche die aktuellen Dekorationen im Codeeditor beschreiben. Diese Liste kann einem erneuten Aufruf von `deltaDecorations` zusammen mit den gewünschten neuen Dekorationen übergeben werden. Monaco führt daraufhin die minimale Anzahl an Operationen aus, um diese Änderung durchzuführen. Hierfür ist es, wie bei der Implementierung der Codevervollständigung (siehe Abschnitt 4.4) nötig, das Ergebnis des vorherigen Aufrufs im State der Komponente zu speichern. Dies führt automatisch dazu, dass bei einer Änderung der Fehlermeldung die alte Dekoration gelöscht wird.

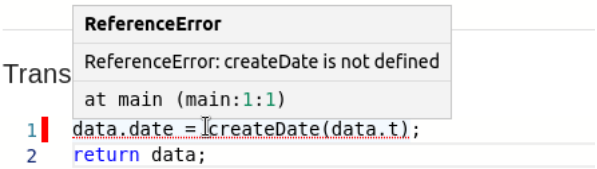

Abbildung 4.4: Anzeige eines Fehlers innerhalb des Codeeditors

4.7 P7: Automatisches Triggern von Transformationen

Als letztes Arbeitspaket wird das automatische Anstoßen der Transformationen bei Änderungen an den Daten oder dem Transformationscode implementiert. Wie in Anforderung A5 beschrieben, sichert dies einen guten Arbeitsfluss ohne Unterbrechungen.

In Vue.js lassen sich *Watcher* definieren. Dies sind Methoden, die ausgeführt werden, sobald sich der Wert der beobachteten Variable ändert. Sie sind somit ideal für das Anstoßen der Transformation.

Um die Zahl der Aufrufe zu verringern, wird erst nach 1,5 Sekunden ohne neue Eingabe ein neuer Test der Transformation ausgeführt. Hierfür wird mithilfe der JavaScript-Methode `setTimeout` eine verspätete Ausführung der `submit`-Methode geplant. Das von `setTimeout` zurückgelieferte Handle für diese geplante Methodenausführung wird gespeichert. Falls vor dem Ablauf der 1,5 Sekunden Wartezeit eine neue Änderung stattfindet, wird die ursprüngliche Planung mithilfe von `clearTimeout` gestoppt. Dies stellt sicher, dass keine unnötigen Aufrufe geschehen, deren Ergebnisse durch einen nachfolgenden Aufruf sofort überschrieben würden.

4.8 Manuelles Testen und Dokumentation

Für das manuelle Testen aller Dienste des ODS wird bei der Entwicklung die Erweiterung **REST-Client**[14] für Visual Studio Code verwendet. Mit dieser Erweiterung können HTTP-Abfragen an Webdienste ausgeführt werden, ähnlich wie mit Kommandozeilenwerkzeug `curl`[15] oder dem beliebten PostMan[16]. Im Gegensatz zu diesen werden bei der Verwendung von REST-Client Beispielabfragen in Dateien definiert. Diese Abfragen können direkt innerhalb der IDE ausgeführt werden. Die Antwort des Dienstes wird in einem extra Fenster angezeigt.

Diese definierten Testabfragen stellen im Falle des ODS auch eine Dokumentation aller REST-Endpunkte der Dienste dar.

[14] `https://github.com/Huachao/vscode-restclient`
[15] `https://curl.haxx.se/`
[16] `https://www.getpostman.com/`

4.9 Automatische Regressionstests

Damit bei der iterativen Entwicklung von neuen Features sichergestellt wird, dass die bereits existierende Funktionaltät nicht fehlerhaft wird, werden sogennante Regressionstests verwendet [Lig09]. Da diese regelmäßig ausgeführt werden müssen, ist eine Automatisierung dieser Tests sinnvoll.

Bei der Umsetzung der Arbeitsschritte wurden sowohl Komponenten- als auch Integrationstests erstellt, um die Funktion des Transformationsdienstes gewährleisten zu können. Über 20 Komponentests überprüfen die Funktionsweise der Klassen `JSTransformationService` und `VM2SandboxExecutor`. Diese werden, wie bei Komponententests üblich, jeweils separat getestet, wobei bei dem Test des Dienstes der Sandbox-Mechanismus mithilfe eines Mock-Objekt simuliert wird [MFC00]. Die Tests werden von dem bereits in Abschnitt 4.1 erwähnten Testframework Jest ausgeführt.

Darüber hinaus existieren über 10 Integrationstests, welche als Black-Box-Tests implementiert sind. Hierbei wird der gesamte Transformationsdienst als Container gestartet. Das Ergebnis von vordefinierten Aufrufen der verschiedenen Endpunkte werden mit den erwarteten Resultaten verglichen. Die Integrationstests sind in JavaScript geschrieben und benutzen ebenfalls den Test-Runner Jest.

Beide Arten von Tests sind Teil der Continuous-Integration-Pipeline, welche die Entwicklung des ODS überwacht und unterstützt.

5 Auswertung

In diesem Kapitel werden die in Kapitel 2 definierten Anforderungen auf ihre Erfüllung hin überprüft. Dies geschieht mithilfe des in Sektion 2.3 beschriebenen Bewertungsmodells.

5.1 Funktionale Anforderungen

Wie im Bewertungsmodell beschrieben, werden alle funktionalen Anforderungen mithilfe einer Checkliste daraufhin überprüft, ob sie erfüllt, teilweise erfüllt oder nicht erfüllt werden.

- A1: Ausführungsmechanismus: erfüllt

  Der Transformationsdienst unterstützt die Ausführung von in JavaScript erstellten Codeabschnitten und liefert das Ergebnis der Transformation zurück.

- A2: Fehlerüberprüfung der Transformationsfunktionen: erfüllt

  Bei der Ausführung von Transformationen werden Syntaxfehler im Transformationscode frühzeitig erkannt. Aus den Syntaxfehlern wird die entsprechende Codezeile und Position extrahiert. Auch Fehler, welche zur Laufzeit des Skriptes auftreten, werden abgefangen und mit entsprechenden Informationen zurückgegeben. Falls der Codeabschnitt keine Daten zurückliefert, wird ebenfalls eine aussagekräftige Fehlermeldung erzeugt.

- A3: Laufzeitverhalten und Metadaten: teilweise erfüllt

  Für einen besseren Überblick bei der Entwicklung werden der Ausführungszeitpunkt und das Ausführungsende der Transformation als Metadaten ausgegeben. Die Dauer der Ausführung wird ebenfalls genau berechnet und den Metadaten hinzugefügt. Es werden allerdings keine Metadaten für die in der Anforderung erwähnten Ressourcenkosten zurückgeliefert. Es wurden mehrere Varianten, den Speicherverbrauch oder die Systemauslastung zu mes-

sen, getestet. Jedoch konnten diese aufgrund der Microservice-Struktur des Projekts und den internen Abläufen der verwendeten Laufzeitumgebung Node.js nicht als aussagekräftige Metriken benutzt werden. Bei der Nutzung anderer Implementierungen ist allerdings eine Erweiterung mit diesen Daten möglich.

- A4: Schnittstelle zu anderen Microservices: erfüllt

  Entsprechend der Anforderung wurde die Schnittstelle des Transformationsdienstes als REST-Schnittstelle implementiert, welche das JSON-Format verwendet. Aufgrund der flexiblen Architektur der Dienstes ließe sich dies jedoch leicht durch ein anderes Transportsystem austauschen.

- A5: Testen der Transformationskonfiguration: erfüllt

  Die Weboberfläche bietet Eingabefelder für die Beispieldaten und den Transformationscode. Bei jeder Änderung wird automatisch ein Test der Konfiguration ausgeführt und das Resultat angezeigt.

- A6: Syntax-Hervorhebung im Codeeditor: erfüllt

  Als Eingabefeld für den Code wird eine Open-Source-Komponente verwendet, welche die Syntaxhervorhebung bereits unterstützt. Wenngleich nicht in der Anforderung erwähnt, wird durch die Benutzung dieser Komponente als Eingabefeld für die Daten auch dort Syntaxhervorhebung bereitgestellt.

- A7: Anzeige der Fehlermeldungen im Code-Editor: erfüllt

  Falls bei der Ausführung der Transformation ein Fehler zurückgeliefert wird, wird dieser direkt an der entsprechenden Stelle im Codeeditor angezeigt.

- A8: Anzeige der Metadaten in der Weboberfläche: erfüllt

  Die Weboberfläche zeigt alle Metadaten an, welche dem Ergebnis einer Transformation hinzugefügt werden.

- A9: Template-Mechanismus: nicht erfüllt

  Zum Zeitpunkt der Abgabe dieser Arbeit ist das System für wiederverwendbare Skripte in Form eines Template-Systems nicht implementiert. Die Überlegungen, wie genau dieses Feature umzusetzen ist, konnten im Zeitrahmen dieser Arbeit nicht fertiggestellt werden. So ist nicht klar, ob die Speicherung der Template-Skripte durch den Transformationsdienst oder die Weboberfläche geschehen soll.

5.2 Nicht-funktionale Anforderungen

5.2.1 Portabilität

Zur Überprüfung der Anforderung nach Portabilität und Anpassbarkeit (A10) wird eine neue Implementierung des Transformationsdienstes erstellt, welche Codeabschnitte in der Programmiersprache Python unterstützt. Der Fokus liegt hierbei darauf, wie stark die Weboberfläche angepasst werden muss, um mit diesem neuen Dienst kompatibel zu sein. Deshalb wurden im Rahmen der Entwicklung dieses Proof of Concepts nur eine rudimentäre Fehlerbehandlung und keine Sicherheitsmechanismen implementiert.

Selbst ohne Änderung an der Weboberfläche ist es möglich, Codeabschnitte in Python zu schreiben und auszuführen. Da der Codeeditor JavaScript-Code erwartet, werden zwar falsche Syntaxfehler angezeigt, diese verhindern jedoch nicht das Abschicken der Testaufforderung und die Ausführung der Transformation im Python-Dienst.

Durch eine manuelle Änderung der erwarteten Sprache des Codeeditor lässt sich die Syntaxüberprüfung und Syntaxhervorhebung auf Python umstellen (siehe Abbildung 5.1). Dies geschieht jedoch nicht automatisch. Da der Monaco-Editor Codevervollständigung für Python nicht nativ beherrscht, erfolgt hier nur rudimentäres Vorschlagen von bereits eingegeben Worten. Für eine vollständige Unterstützung von Codevervollständigung für beliebige Sprachen müsste der Codeeditor mit einem Language-Server verbunden werden (siehe Kapitel 6).

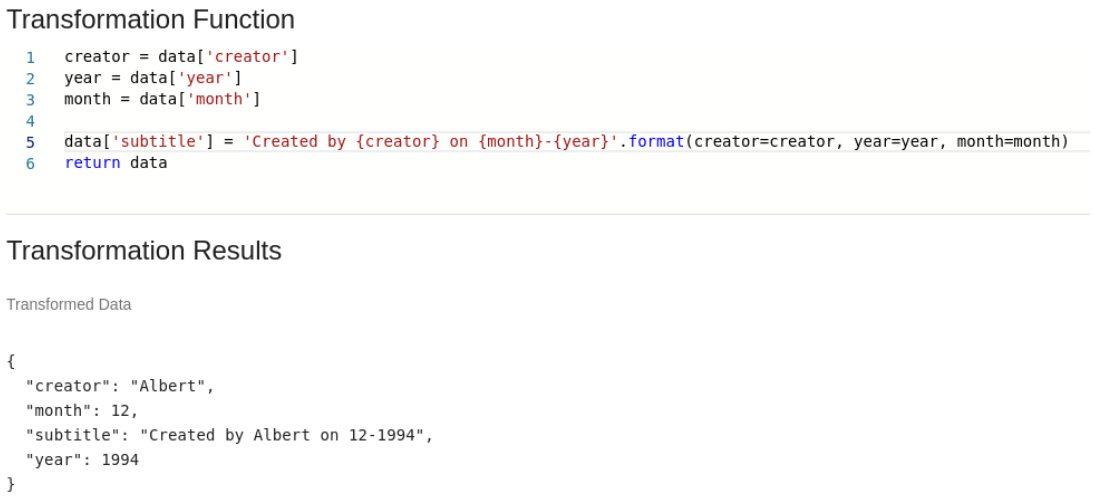

Abbildung 5.1: Syntaxhervorhebung für Transformationscode in Python

5.2.2 Usability

Anforderung A14 beschreibt, dass die Benutzeroberfläche der Webkomponente übersichtlich, ansprechend und benutzerfreundlich sein soll.

Mit der ISO-Norm 9241 existiert ein Standard, welcher Richtlinien betreffend er „Ergonomie der Mensch-System-Interaktion“ auflistet. Der Teil 110: „Grundsätze der Dialoggestaltung“ beschäftigt sich mit dem Erstellen von Benutzeroberflächen [ISO06]. Wie bei der Beschreibung des Bewertungsmodells in 2.3 erwähnt, wird die Benutzeroberfläche der Live-Konfiguration bezüglich der sieben Grundsätze dieser ISO-Norm geprüft. Hierzu werden von den zahlreichen Empfehlungen, die für diese Grundsätze in der Norm enthalten sind (oftmals sechs pro Grundsatz), jedoch nur eine oder zwei exemplarisch ausgewählt.

Aufgabenangemessenheit

- „Der Dialog sollte dem Benutzer solche Informationen anzeigen, die im Zusammenhang mit der erfolgreichen Erledigung der Arbeitsaufgabe stehen.“ erfüllt

  Die Arbeitsaufgabe des Nutzers ist die Konfiguration des Transformationsdienstes mithilfe eines Codeabschnitts und Beispieldaten. Die Weboberfläche zeigt diesen Code zentral an, sowie ein Eingabefeld für die Beispieldaten und das Ergebnisfeld.

- „Die Form der Eingabe und Ausgabe sollte der Arbeitsaufgabe angepasst sein.“ erfüllt

  Die Eingabe der Beispieldaten und des Codes erfolgt in Codeeditoren, welche automatische Formatierung und Syntaxhervorhebung unterstützten. Das JSON-Ergebnis wird formatiert und eingerückt angezeigt.

Selbstbeschreibungsfähigkeit

- „Wenn eine Eingabe verlangt wird, sollte das interaktive System dem Benutzer Informationen über die erwartete Eingabe bereitstellen.“ erfüllt

  Beide Eingabefelder der Webkomponente sind bei dem Start der Konfiguration mit Beispielinhalt gefüllt, der einen Rückschluss auf die erwarteten Eingaben zulässt.

Erwartungskonformität

- „Auf Handlungen des Benutzers sollte eine unmittelbare und passende Rückmeldung folgen, soweit dies den Erwartungen des Benutzers entspricht.“ erfüllt

  Bei jeder Änderung der Konfiguration wird nach 1,5 Sekunden automatisch ein Test angestoßen, dessen Ergebnis baldmöglichst angezeigt wird.

Lernförderlichkeit

- „Regeln und zugrunde liegende Konzepte, die für das Erlernen nützlich sind, sollten dem Benutzer zugänglich gemacht werden.“ erfüllt

  Die Syntaxhervorhebung und Codevervollständigung im Codeeditor unterstützen den Nutzer bei der Erstellung der Codeabschnitte in JavaScript.

Steuerbarkeit

- „Wenn es für die Arbeitsaufgabe zweckmäßig ist, sollte der Benutzer voreingestellte Werte ändern können.“ nicht erfüllt

  Außer den Beispieldaten und dem Codeabschnitt bietet die Webkomponente keine Einstellungsmöglichkeiten, wie etwa für die Zeit bis zum automatischen Test oder der maximalen Skriptausführungsdauer.

Fehlertoleranz

- „Das interaktive System sollte den Benutzer dabei unterstützen, Eingabefehler zu entdecken und zu vermeiden.“ erfüllt

  Eine vom Transformationsdienst zurückgelieferte Fehlermeldung wird deutlich in roter Schrift im Ergebnisfenster angezeigt.

- „Aktive Unterstützung zur Fehlerbeseitigung sollte dort, wo typischerweise Fehler auftreten, zur Verfügung stehen.“ erfüllt

  Der Fehler wird über die mitgelieferten Informationen zur Zeilennummer und Position in der entsprechenden Stelle im Codeeditor markiert.

Individualisierbarkeit

- „Das interaktive System sollte es dem Benutzer erlauben, zwischen verschiedenen Formen der Darstellung zu wählen, wenn es für die individuellen Bedürfnisse unterschiedlicher Benutzer zweckmäßig ist. “ nicht erfüllt

  Die Webkomponente bietet derzeit keine Individualisierungsmöglichkeiten. Ansätze zur Individualisierung müssten den gesamten ODS umfassen und nicht nur die einzelne Komponente zur Transformationskonfiguration.

5.2.3 Sicherheit und Verfügbarkeit

In den Anforderungen A11 bis A13 wird beschrieben, dass schadhafter Code die Funktionsweise des Transformationsdienstes nicht beeinträchtigen darf. So soll der Dienst weder durch Endlosschleifen blockiert, noch Zugriff auf das Hostsystem erlauben.

Die Sicherstellung dieser Anforderungen soll durch die sorgfältige Auswahl eines geeigneten Sandbox-Mechanismus gewährleistet sein (vergleiche Abschnitt 4.2).

Die Überprüfung der Anforderungen wird durch das Übergeben von Schadcode getestet. Je nachdem ob der Transformationsdienst die Ausführung dieses Schadcodes mit einer Fehlermeldung abbricht, oder aber eine Verletzung der Anforderungen geschieht, wird der Test mit bestanden oder nicht bestanden bezeichnet

- Endlosschleife: bestanden

  Die Ausführung einer Endlosschleife wie `while(true){}` wird nach der eingestellten Zeitsperre (standardmäßig fünf Sekunden) mit einem `TimeoutError` abgebrochen.

- Nachladen von Bibliotheken: bestanden

  In JavaScript erfolgt das Laden von Bibliotheken mithilfe des `require`-Befehls. Wenn dieser Befehl innerhalb des Transformationscodes verwendet wird, wird die Ausführung mit einem `ReferenceError: require is not defined` unterbrochen.

- Stoppen des Dienstes durch Zugriff auf Systemfunktionen: bestanden

  Durch die Verwendung des globalen JavaScript-Objekts `process` lässt sich der aktuelle Node.js-Prozess stoppen. Die Sandbox verhindert jedoch den Zugriff auf dieses Objekt. Ein Zugriff wie beispielsweise `process.exit(0)` wird ebenfalls mit einem `ReferenceError` unterbunden.

- Ausbruch aus der Sandbox: bestanden

Sobald der Sandbox Daten übergeben werden, besteht hier die Möglichkeit, aus der Sandbox auszubrechen und Zugriff auf das Hostsystem zu erlangen. In JavaScript ist dies, sofern keine Gegenmaßnahmen ergriffen werden, durch die Verwendung des Konstruktor-Systems möglich. Jeder Konstruktor ist eine Funktion und der Konstruktor des Funktions-Objekts kann dazu verwendet werden, beliebigen Code auszuführen, ähnlich zu `eval`. Dies kann beispielsweise wie folgt missbraucht werden:

```
console.constructor.constructor('return␣process;')
   ();
```

Die verwendete Bibliothek vm2 fängt dies jedoch durch die Verwendung der Proxies ab und es gelingt kein Ausbruch aus der Sandbox. Das Ergebnis der Ausführung ist wieder ein `ReferenceError: process is not defined`.

5.3 Zusammenfassung

Das Konzept zur Live-Konfiguration wurde erfolgreich erstellt und beispielhaft implementiert. Dabei wurden alle Arbeitsschritte in dem Entwicklungsprozess als Teil dieser Arbeit dokumentiert.

In Kapitel 2 wurden alle funktionalen und nicht-funktionalen Anforderungen erfasst, die die Implementierung des Live-Konfiguration-Konzepts am Ende der Arbeit erfüllen soll.

Die Entwicklung des Konzepts wurde in Kapitel 3 beschrieben. Hierfür wurde die Idee zuerst generalisiert und ein allgemeines Konzept erstellt, welches dann auf den konkreten Fall des Transformationsdienstes angewendet wurde.

Alle Entwicklungsschritte der Implementierung des Konzepts wurden in Kapitel 4 aufgezählt. Neben Erläuterungen, wie gewisse Entscheidungen getroffen wurden, wurden auch Probleme bei der Entwicklung deutlich gemacht.

Zusammenfassend wurden fast alle Anforderungen erfüllt, welche an die Live-Konfiguration gestellt wurden. Zu Beginn der Arbeit standen zur Konfiguration nur simple Texteingabefelder zur Verfügung und es gab wenig Rückmeldung für fehlerhafte Eingaben. Da die Weboberfläche nicht nutzerfreundlich gestaltet war, war die Konfiguration des Transformationsdienstes sehr fehleranfällig. Durch das automatische Testen des Transformationscodes ist die Erstellung des Transformationscodes jetzt erheblich angenehmer. Die Live-Vorschau der Ergebnisdaten erleichtert das Anpassen des Transformationscodes, der die entsprechenden Befehle zur Datenverarbeitung enthält. Falls inkorrekter Code eingegeben wird oder bei der Ausführung Fehler auftreten, werden die aussagekräftigen Fehlermeldungen auch direkt im Codeeditor anzeigt.

6 Ausblick

Die derzeitige Implementierung der Live-Konfiguration des Transformationsdienstes bildet ein solides Fundament. Nichtsdestotrotz würden einige Weiterentwicklungen das Gesamtprodukt komplementieren.

Das Feature der Templates, welches im Rahmen der Arbeit nicht mehr fertiggestellt werden konnte, würde die Erstellung von Transformationscode durch die Möglichkeit der Wiederverwendung von Abschnitten weiter vereinfachen. Das Sicherheitslevel der verwendeten Sandbox-Lösung vm2 ist zwar hinreichend, dennoch kann eine Benutzung von containerbasierten Mechanismen eine Überlegung wert sein. Die Verwendung des Google-Projekts gVisor[1] könnte hier eine gute Lösung sein.

Neben Verbesserungen der Benutzerfreundlichkeit und der Sicherheit kann der Transformationsdienst auch durch die Unterstützung weiterer Sprachen erweitert werden. Wie in 5.2.1 beschrieben, bietet das implementierte Konzept durch die flexible Softwarearchitektur und die Unterstützung des Language Server Protocols beste Voraussetzungen dafür, auch andere Sprachen wie Python zu unterstützen. Eine Alternative ist auch die Verwendung von deklarativen Sprachen anstelle der derzeit benutzten Skriptsprache. Während theoretisch auch bekannte Transformationssprachen wie XSL Transformation benutzt werden könnten, kann es vor allem bei sehr großen Datenmengen hilfreich sein, auf speziell für diesen Anwendungsfall entwickelte Sprachen zurückzugreifen. Vielversprechend ist hier ein noch recht unbekanntes Projekt der englischen Firma DataSift[2]: die deklarative Ingestion Data Mapping Language (IDML)[3] wurde speziell für die Verarbeitung von großen Mengen an unstrukturierten Daten entworfen.

Die Unterstützung zusätzlicher Sprachen könnte auch dazu genutzt werden, das Konzept der Live-Konfiguration in weiteren Anwendungsfeldern wiederverwenden zu können.

[1] `https://github.com/google/gvisor`
[2] https://datasift.com/
[3] http://idml.io/

Literaturverzeichnis

[BCD16] R. Buyya, R.N. Calheiros und A.V. Dastjerdi. *Big Data: Principles and Paradigms*. Elsevier Science, 2016.

[Bec03] Kent Beck. *Extreme Programming: die revolutionäre Methode für Softwareentwicklung in kleinen Teams]*. Pearson Deutschland GmbH, 2003.

[BG11] Johannes Breimeier und Heinz-Peter Gumm. *Einführung in die Informatik.* De Gruyter Oldenbourg, 2011.

[CM98] Guy Cousineau und Michel Mauny. *The functional approach to programming*. Cambridge University Press, 1998.

[Gar11] Raffaele Garofalo. *Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern*. Microsoft Press, 2011.

[Gol+96] Ian Goldberg et al. "A secure environment for untrusted helper applications: Confining the wily hacker". In: *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*. Bd. 6. 1996.

[ISO06] ISO. *Ergonomie der Mensch-System-Interaktion - Teil 110: Grundsätze der Dialoggestaltung*. Beuth Verlag, Berlin, 2006.

[LF14] James Lewis und Martin Fowler. *Microservices*. `https://martinfowler.com`. 2014.

[Lig09] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media, 2009.

[Mar09] Robert C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[Mar17] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[Mey88] Bertrand Meyer. *Object-oriented software construction*. Prentice hall New York, 1988.

[MFC00] Tim Mackinnon, Steve Freeman und Philip Craig. "Endo-testing: unit testing with mock objects". In: *Extreme programming examined* (2000), S. 287–301.

[Mica] Microsoft. *Official Page for Language Server Protocol.* `https://microsoft.github.io/language-server-protocol/`. Accessed: 2019-12-15.

[Micb] Microsoft. *TypeScript - JavaScript that scales.* `http://typescriptlang.org`. Accessed: 2019-12-08.

[MP13] Michael Mikowski und Josh Powell. *Single Page Web Applications: JavaScript End-to-end.* Manning Publications Co., 2013.

[Nah03] Fiona Nah. *A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait?* Behaviour und Information Technology, 2003.

[New15] Sam Newman. *Building microservices: designing fine-grained systems.* O'Reilly Media, Inc., 2015.

[Nod] Node.js Foundation. *Node.js v13.3.0 Documentation.* `https://nodejs.org/api/documentation.html`. Accessed: 2019-12-09.

[PW15] Robert Prediger und Ralph Winzinger. *Node. js: Professionell hochperformante Software entwickeln.* Carl Hanser Verlag GmbH Co KG, 2015.

[Rie14] Eric Ries. *Lean Startup: Schnell, risikolos und erfolgreich Unternehmen gründen.* Redline Wirtschaft, 2014.

[The] The Cloud9 Team. *Cloud9 now runs on and integrates with AWS.* `https://c9.io/announcement`. Accessed: 2019-12-14.

[Wes+17] Bridgette Wessels et al. *Open Data and the Knowledge Society.* Amsterdam University Press, 2017.

[YC79] Edward Yourdon und Larry L Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design.* Prentice-Hall, Inc., 1979.