



HOCHSCHULE FULDA
FACHBEREICH ANGEWANDTE INFORMATIK
STUDIENGANG WI (B.Sc.)

Low code Applikation Entwicklung mit SAP AppGyver im Vergleich zu nativen Fiori Entwicklung

Bachelorarbeit von Fangfang Tan
Matrikelnummer: 1222047

Erstgutachterin: Prof. Dr. Norbert Ketterer
Zweitgutachter: M. A. Mike Zashka
Abgabetermin: 6. Februar 2023



Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben. Ich erkläre weiterhin, dass die vorliegende Arbeit in gleicher oder ähnlicher Form noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Bad Hersfeld, den 24. Januar 2023

DEINE Unterschrift

Danksagung

An erster Stelle möchte ich meiner Betreuer Prof. Dr. Norbert Ketterer danken, der mich richtungsweisend und mit viel Engagement während meiner Arbeit begleitet hat.

Besonderen Dank gebührt meinem Betreuer Mike Zashka für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit. Ein herzliches „Dankeschön! “ geht auch an allen anderen Kollegen und Kolleginnen der Firma p36, die mich herzlich aufgenommen und mir während der Schreibphase meiner Bachelorarbeit wertvollen Unterstützungen gegeben haben.

Zusammenfassung

In dieser Arbeit werden xxxx.

Inhaltsverzeichnis

Abbildungsverzeichnis	VII
Tabellenverzeichnis	VIII
Quelltextverzeichnis	IX
Abkürzungsverzeichnis	X
1 Einleitung	1
1.1 Problemstellung und Motivation	1
1.2 Ziele der Arbeit	2
1.3 Beschreibung des Anwendungsfalls	3
1.4 Aufbau der Arbeit	4
2 Grundlagen	6
2.1 Entwicklung mit Low-Code/No-Code	6
2.2 Architektur von SAP-Anwendungen in der SAP Business Techno- logy Plattform	8
2.3 SAP Fiori	10
2.4 SAP AppGyver	11
2.4.1 Grundlage von AppGyver	11
2.4.2 Entwicklungsumgebung: Composer Pro	12
2.5 SAPUI5	16
2.5.1 Grundlage von SAPUI5	16
2.5.2 Entwicklungsumgebung: Visual Studio Code	17
2.6 Fiori Elements	18
2.6.1 Grundlage von Fiori Elements	18
2.6.2 Entwicklungsumgebung: Business Application Studio	20
3 Konzept der Transformationsengine	22

3.1	Allgemeines	23
3.2	Spezifischer Ansatz	26
3.3	Generischer Ansatz mit Excel-Datei	26
3.3.1	Zusätzliche Einstellungen	28
3.4	Generischer Ansatz mit Mapping-Datei	30
3.4.1	JSON-Schema	33
4	Implementierung	35
4.1	Sprachen und Bibliotheken	35
4.1.1	JSON	35
4.1.2	Excel	37
4.1.3	Testen	38
4.2	Ein- und Ausgabeparameter	38
4.3	Ansatz mit Excel-Datei	40
4.4	Ansatz mit Mapping-Datei	46
4.4.1	Datenbindung	47
4.4.2	Produktdaten schreiben	50
5	Evaluation	53
5.1	Gegenüberstellung der drei Ansätze	53
5.2	JSON-Abfragesprache	56
5.3	Massen-Upload	58
6	Zusammenfassung und Ausblick	61
	Literaturverzeichnis	65
A	JSON-Schema	73

Abbildungsverzeichnis

2.1	Architektur einer SAP-Anwendung in der SAP BTP	10
2.2	Pages einer AppGyver-Anwendung	13
2.3	Toolbar in Composer Pro	13
2.4	View-Components in Composer Pro	14
3.1	Excel-Vorlage für Taiwan – Ausschnitt 1	22
3.2	Excel-Vorlage für Taiwan – Ausschnitt 2	23
3.3	Excel-Vorlage für Taiwan – Ausschnitt 3	24
3.4	Beispiele für Datenelemente	24
3.5	JMESPath-Ausdrücke in der Excel-Vorlage einer Behörde	27
3.6	Das Settings-Arbeitsblatt aus Ansatz 3.3	29
4.1	Sequenzdiagramm für Ansatz 3.3	41
4.2	Sequenzdiagramm für Ansatz 3.4	46
4.3	Aktivitätsdiagramm für <code>writeOneDevice</code> aus Ansatz 3.4	51
5.1	Testergebnisse zum Massen-Upload	59

Tabellenverzeichnis

1.1	Definition der Anwendungsfall	3
2.1	AppGyver-Baustein in die Schichten des MVC-Models	14
3.1	Funktionale und nichtfunktionale Anforderungen	26
4.1	Überblick über die verwendeten Bibliotheken	39
5.1	Evaluationsmatrix für die Transformationsengine	54
5.2	Evaluationsmatrix für die JSON-Abfragesprache	57

Quelltextverzeichnis

3.1	Beispiel einer Mapping-Datei für <code>SheetMappings</code>	30
3.2	Beispiel einer Mapping-Datei für <code>ElementMappings</code>	32
4.1	Auszüge aus der Klasse <code>TransformationEngine</code>	40
4.2	Einlesen der Spalteninformationen in <code>TemplateReader</code>	40
4.3	Hilfsfunktion beim Einlesen der Spalteninformationen	43
4.4	Beispiel eines komplexen Datenelements	44
4.5	Datenbindung für Arbeitsblätter der Mapping-Datei	47
4.6	Setter / Getter für verbindliche Spalten in <code>SheetMapping</code>	49
A.1	JSON-Schema für die Mapping-Datei	73

Abkürzungsverzeichnis

API	Application Programming Interface – Programmierschnittstelle
BTP	Business Technology Plattform
BAS	Business Application Studio
CAP	SAP Cloud Application Programming Model
CDS	Core Data Servicel
CORS	Cross Origin Resource Sharing
CRM	Customer Relationship Management
CRUD	Create Read Update Delete – Vier fundamentale Optionen des Datenmanagement
CSS	Cascading Style Sheets
CSV	Comma-Separated-Values – Datenformat
GPS	Global Positioning System
HTML	HyperText Markup Language – Auszeichnungssprache
IDE	Integrierte Entwicklungsumgebung
JSON	JavaScript Object Notation – Datenserialisierungsformat
LCNC	Low-Code/No-Code
MVC	Model-View-Controller – Software Designmuster
OData	Open Data Protocol
NDC	Native Device Capabilities

NPM	Node Package Manager
PDF	Portable Document Format
PT	Personentage
SAP	Systemanalyse Programmentwicklung – Softwarekonzern
SAPUI5	SAP UI Development Toolkit für HTML5
SDK	Software Development Kit
UI	User Interface
URL	Uniform Resource Locator – Internetadresse
UX	User Experience
VS-Code	Visual Studio Code – Software Entwicklungstool
XML	Extensible Markup Language – erweiterbare Auszeichnungssprache
YAML	YAML Ain't Markup Language – Datenserialisierungsformat

Kapitel 1

Einleitung

1.1 Problemstellung und Motivation

Seit die Branchenanalysten von Forrester Research im Jahr 2014 erstmals das Konzept von Low-Code erwähnten [RJR22], hat sich der zugehörige Markt rasant entwickelt. Immer mehr Unternehmen nutzen Low-Code-Plattformen, um Anwendungen schneller zu entwickeln und damit den digitalen Wandel zu beschleunigen. Die Analysten von Gartner erwarten, dass im Jahr 2025 rund 70% der von Unternehmen entwickelten Anwendungen auf Low-Code-Technologien basieren werden [HV22].

Im Bereich der Enterprise Software werden die meisten Low-Code-Plattformen verwendet, um eine bestimmte Anwendungsform in einem spezifischen Kontext zu entwickeln. In der Studie „No-Code/Low-Code 2022“ im Magazin COMPUTERWOCHE, gibt die Mehrheit der befragten Unternehmen an, dass sie Low-Code hauptsächlich in den Bereichen CRM (34%) und ERP (31%) einsetzen. Speziell ausgerichtete Low-Code Plattformen werden ebenfalls im HR-Umfeld (19%) verwendet, sowie für die Erstellung digitaler Workflows und Verwaltungsprozesse (jeweils 16%). Nur 10% der Befragten nutzen eine universell einsetzbare Plattform, die sich für übergreifende und flexible Geschäftsprozesse eignet. Laut Jürgen Erbdinger, einem Low-Code-Experten der Low-Code-Plattform ESCRIBA, fehlt den Plattformen hierfür die entsprechende Tiefe [AS22].

AppGyver betrachtet sich selbst als die weltweit erste professionelle Low-Code Plattform, die es ermöglicht, Anwendungen für unterschiedliche Geschäftsprozesse, Anwendungsszenarien und auch Endgeräte zu erstellen [SAP22b]. Im Februar

2021 wurde AppGyver von dem Marktführer im Bereich Enterprise Software SAP übernommen und wird seitdem in deren Entwicklungsportfolio rund um die SAP Business Technology Plattform eingegliedert [Cen21], [Com22b]. Seit dem 15. November 2022 wurde SAP AppGyver in SAP Build App umbenannt und ist nun Teil von SAP Build. AppGyver steht damit in Konkurrenz zu etablierten Tools und Frameworks zur Anwendungsentwicklung: SAPUI5 ist ein JavaScript-Framework und bildet die Grundlage nahezu aller heute entwickelten SAP-Oberflächen. Die Erstellung von SAPUI5-Anwendungen setzt jedoch ein tieferes technisches Verständnis voraus. Basierend auf SAPUI5 steht mit SAP Fiori Elements ein Framework zur Verfügung, welches durch Annotationen die einfache Erstellung von datengetriebenen Oberflächen erlaubt. Dank der guten Integration in die SAP-eigenen Entwicklungsumgebungen, das SAP Business Application Studio, kann SAP Fiori Elements in Kombination mit dem SAP Application Programming Model auch im Bereich der Low-Code Entwicklung platziert werden [Ele22]. Für Unternehmen ergibt sich in Zukunft nun die Fragestellung, welche Plattform und Tools im SAP-Umfeld eingesetzt werden sollten, um Anwendungen zu entwickeln. Die Aufgabe dieser Bachelorarbeit besteht darin, den Entwicklungsprozess von SAP AppGyver, SAPUI5, sowie Fiori Elements in Kombination mit dem SAP Application Programming Model zu bewerten, Vor- und Nachteile der jeweiligen Lösung herauszuarbeiten und dadurch eine Entscheidungsmatrix zu entwerfen, welche die Wahl zwischen diesen drei Technologien vereinfacht.

Diese wissenschaftliche Arbeit wird dabei unterstützt durch die Firma p36 GmbH. P36, mit dem Sitz im Bad Hersfeld, wurde 2015 von Patrick Pfau und Robin Wennemuth gegründet. Das mit 27 Mitarbeitern noch recht kleine, aber stark wachsende Softwareunternehmen besitzt einen starken Fokus auf die Entwicklung von Cloud-basierten Anwendungen im SAP-Umfeld [Gmb22]. Bei der Umsetzung der Anwendungen setzt p36 überwiegend auf die sehr technische SAPUI5-Entwicklung und evaluiert derzeit den Einsatz von Low-Code-Plattformen. p36 stellt deswegen einen, sich an realen Kundenanforderungen orientierenden, Anwendungsfall zur Verfügung, der im Rahmen der Arbeit als Grundlage der Evaluierung dienen soll.

1.2 Ziele der Arbeit

Die folgenden Fragen werden in der Bachelorarbeit behandelt werden:

- Was genau verbirgt sich hinter dem Begriff Low-Code und wie grenzt sich Low-Code von bisherigen Arten der Entwicklung ab?

- Wie wird eine benutzerspezifische Anwendung mit dem Low-Code/No-Code basierten Tool SAP AppGyver implementiert?
- Wie wird eine benutzerspezifische Anwendung mit SAPUI5 implementiert?
- Wie wird eine benutzerspezifische Anwendung mit Fiori Elements implementiert?
- Welche Vor- und Nachteile dieser drei Technologien lassen sich durch die exemplarische Umsetzung herausstellen?
- Welche Technologie eignet sich in Zukunft für welche Umsetzungsszenarien?

1.3 Beschreibung des Anwendungsfalls

In dieser Arbeit werden die drei genannten Technologien verwendet, um eine konkrete Anwendung zu entwickeln. Der Anwendungsfall definiert sich, wie folgt:

Name:	Applikation zur Verwaltung von Produktinformationen
Umsetzung in:	<ul style="list-style-type: none"> • SAP Fiori Elements mit SAP Business Application Studio (Backend + UI) • SAP AppGyver (UI) • SAPUI5 (UI)
Anforderungen Backend:	<ul style="list-style-type: none"> • Bereitstellung einer Datenbank-Entität Products mit folgenden Eigenschaften: <ul style="list-style-type: none"> – ID; title; materialNumber; description; price; stock • Bereitstellung eines OData-Services zum Auslesen, Erstellen und Aktualisieren (CRUD) der Produkte
Anforderungen Frontend:	<ul style="list-style-type: none"> • Funktionen: <ul style="list-style-type: none"> – Listenansicht zur Anzeige aller Produkte – Einzelansicht für ein Produkt – Maske zum Pflegen eines einzelnen Produkts • Datenanbindung: <ul style="list-style-type: none"> – Anbindung an den OData-Service zum Auslesen und Schreiben von Produkten • Look and Feel: <ul style="list-style-type: none"> – Implementierung in Anlehnung an die SAP UX-Guideline SAP Fiori

Tabelle 1.1: Definition der Anwendungsfall

Zusätzlich zu den umzusetzenden Funktionalitäten wird eine Reihe weiterer

Funktionen ohne praktische Umsetzung untersucht, um SAPUI5, Fiori Elements und AppGyver tiefergehend zu evaluieren. Diese Funktionen umfassen:

- Integration von Suchfilter und Paginierung
- Integration von Bild und PDF-Datei
- Integration von Barcode-Scanner-Funktionen
- Nutzung mobiler Funktionen wie Sensoren
- Möglichkeiten zum Deployment für unterschiedliche Endgeräte
- Freie Gestaltungsmöglichkeiten

1.4 Aufbau der Arbeit

Die vorliegende Bachelorarbeit gliedert sich in insgesamt sechs Kapitel. In der Einleitung werden die Problemstellung und Motivation, die Ziele der Arbeit und der umzusetzende Anwendungsfall vorgestellt.

Im 2. Kapitel werden zunächst die grundlegenden Konzepte erläutert. Der erste Abschnitt beschäftigt sich mit dem Low-Code/No-Code (LCNC) Konzept und liefert eine Definition von LCNC und einen Überblick über existierende LCNC-Plattformen. Kapitel 2 beinhaltet ebenfalls einen Überblick über die Architektur von SAP-Anwendungen in der SAP Business Technology Plattform, sowie, im dritten Abschnitt, die Grundlagen von SAP Fiori. Der vierte, fünfte und letzte Abschnitt dieses Kapitels beschreibt die Grundlagen von AppGyver, SAPUI5 und Fiori Elements, sowie die Entwicklungsumgebung, in denen der genannte Anwendungsfall entwickelt wird, nämlich SAP Business Application Studio, Composer Pro und Visual Studio Code.

Kapitel 3 bis 5 bilden den Hauptteil der Bachelorarbeit. Im dritten Kapitel wird der Umsetzungsprozess des Anwendungsfalls mit Fiori Elements, AppGyver und SAPUI5 beschrieben. Die zu beschreibenden Funktionen umfassen:

- Bereitstellung eines OData-Services zum Auslesen, Erstellen und Aktualisieren der Produkte
- Erstellung einer Listenansicht zur Anzeige aller Produkte
- Erstellung einer Einzelansicht für ein Produkt
- Erstellung einer Maske zum Pflegen eines einzelnen Produkts

Im 4. Kapitel werden weitere Funktionen, ohne technische Implementierung, untersucht, um Fiori Elements, AppGyver und SAPUI5 eingehender zu bewerten. Basierend auf Kapitel 3 und Kapitel 4 konzentriert sich Kapitel 5 auf die Gegenüberstellung und Bewertung der drei Tools. Hierfür werden die Be-

wertungsmatrizen definiert, die Bewertung durchgeführt und anschließend die Bewertungsergebnisse diskutiert und interpretiert. Kapitel 6, das letzte Kapitel der Bachelorarbeit, enthält abschließend ein Fazit und einen Ausblick auf die künftige Forschung.

Kapitel 2

Grundlagen

2.1 Entwicklung mit Low-Code/No-Code

Low-Code/No-Code ist ein Ansatz der Softwareentwicklung, bei dem Anwendungen mit wenig oder gar keinem selbst programmierten Code erstellt werden können. Pro-Code hingegen bezieht sich auf die klassische Entwicklung, bei der die Codezeilen von Hand geschrieben werden. Anstatt auf komplexe Programmiersprachen zurückzugreifen, kann LCNC-Entwicklung die Anwendungen durch visuelle Programmierung, also durch Anklicken, Ziehen und miteinander verbinden von Anwendungskomponenten, erstellen. Die Low-Code/No-Code-Plattformen bieten hierfür spezielle visuelle Programmierungsumgebungen an, die aus einer Reihe an vorgefertigten Code-Bausteinen und den Möglichkeiten diese in Form einer Anwendung zusammenzusetzen, bestehen. No-Code-Plattformen ersetzen die traditionelle codebasierte Entwicklungsumgebung dabei vollständig, während bei Low-Code-Plattformen möglicherweise Basis-Programmierkenntnisse erforderlich sind.

Auch wenn es bereits in der Vergangenheit Ansätze zur visuellen Programmierung gab, so ist die derzeitige LCNC-Entwicklung aufgrund des Reifegrades des Toolings eine ernsthafte Alternative zur Pro-Code-Entwicklung. Einige Experten glauben, dass LCNC die Zukunft der Softwareentwicklung ist, weil es einen schnelleren Entwicklungsprozess ermöglicht. Mit den einfach zu bedienenden visuellen Benutzeroberflächen, sowie den ausgereiften Entwicklungstoolkits ist man in der Entwicklung deutlich schneller, als wenn man Tausende von Codezeilen schreiben muss. Neben dem Zeitfaktor spielt auch die damit einzusparenden Kosten eine große Rolle [Con22].

Ein weiterer Vorteil der LCNC-Entwicklung ist, dass sie den Mangel an qualifizierten Entwicklern kompensiert. LCNC-Entwicklung eignet sich für Entwickler aller Niveaus. Die No-Code-Plattformen sind insbesondere für Citizen Developer sinnvoll, die möglicherweise überhaupt keine Programmierausbildung haben. Ein Citizen Developer ist ein Mitarbeiter, der mit zugelassenen Tools Anwendungen für eigene Nutzung oder die Nutzung durch andere erstellt [GG22]. Darüber hinaus bieten LCNC-Plattformen professionellen Entwicklern Unterstützung, um die aufwändigen zugrundeliegenden architektonischen und infrastrukturellen Aufgaben zu reduzieren.

Der Markt für LCNC-Plattformen ist in den letzten Jahren deutlich gewachsen. Nach Angabe von G2, eine der Website für Softwarelisten und Bewertungen, gibt es (Stand November 2022) 226 Low-Code Plattformen [Ove22a] und 288 No-Code Plattformen [Ove22b] auf dem Markt. Neben spezialisierten Unternehmen/Start-Ups, stellen auch größere Unternehmen LCNC-Plattformen für ihr jeweiliges Ökosystem zur Verfügung. Dazu einige Beispiele:

App Engine von ServiceNow wurde im März 2021 veröffentlicht [doc22c]. Es ermöglicht großen Unternehmen Low-Code-Anwendungen zu erstellen und bereitzustellen. ServiceNow stellt eine Reihe an Entwicklungsvorlagen für gängige Anwendungsfälle bereit, um die Erstellung der Anwendungen zu erleichtern [CVE22]. App Engine erlaubt den Nutzern allerdings auch, Code mit traditionellen Programmiersprachen wie HTML, Javascript sowie CSS zu schreiben und zu bearbeiten.

Salesforce, vom gleichnamigen Unternehmen, ist eine Plattform für Customer-Relationship-Management (CRM) und besitzt umfangreiche Funktionen einer App-Entwicklungsplattform, um die Standard-CRM-Funktionalitäten der Plattform zu erweitern. Mit Hilfe der visuellen Programmierung können Workflow-basierte Anwendungen schnell erstellt werden, um Geschäftsprozesse abzubilden oder Kunden Zugang zu wichtigen Informationen zu gewähren. Die Salesforce-Plattform bietet neben dem Low-Code-Ansatz auch eine vollständig angepasste Anwendungsentwicklung für unterschiedliche Programmiersprachen und ist daher auch geeignet für den Code-basierten Ansatz [G222].

OutSystems vom gleichnamigen deutschen Hersteller, ist ein Beispiel für eine spezialisierte LCNC-Plattform ohne direkte Einbindung in ein größeres Ökosystem. Auch hier steht die visuelle Full-Stack-Entwicklung im Vordergrund, mit der Benutzeroberflächen, Geschäftsprozesse, Logik und Datenmodelle aufgebaut und implementiert werden können. Auch bei OutSystems ist es jedoch möglich, eigenen Code für die Anwendungserstellung hinzuzufügen

Der Wettbewerb im Trend-Thema LCNC ist heute sehr stark. Auf der einen Seite gibt es die großen Unternehmen wie ServiceNow und Salesforce, die über viele Ressourcen und Fachkräfte für die Entwicklung ihrer Plattformen verfügen und ihre Kunden mit der Bereitstellung von LCNC-Funktionalitäten weiter an die Plattform binden wollen. Die resultierenden Anwendungen sind zumeist plattformabhängig, haben jedoch den großen Vorteil, dass die Daten aus dem jeweiligen Ökosystem auch anwendungsübergreifend wiederverwendet werden können. Auf der anderen Seite gibt es die spezialisierten LCNC-Anbieter, wie OutSystems und Mendix, mit denen die Benutzer unabhängige Anwendungen entwickeln können und weniger an eine Plattform oder einen Anbieter gebunden sind [CVE22].

Im weiteren Verlauf dieser Arbeit soll der Fokus auf der LCNC-Plattform SAP AppGyver liegen. AppGyver ist einer der Top-Anbieter für die LCNC-Entwicklung und kann als hybride Plattform angesehen werden. Ursprüngliche unabhängig und spezialisiert, entwickelt sich AppGyver durch den Kauf durch SAP und der Integration in das SAP Ökosystem zu einer leistungsfähigen Plattform, die beide Welten miteinander verbindet. Auf AppGyver wird in Abschnitt 2.4 näher eingegangen.

2.2 Architektur von SAP-Anwendungen in der SAP Business Technology Plattform

Der praktische Teil dieser Arbeit befasst sich mit der Erstellung von Anwendungen in den drei gewählten Technologien: AppGyver, SAP Fiori Elements und SAPUI5. Die grundlegende Architektur der Anwendungen orientiert sich an der heutigen Referenzarchitektur von Anwendungen auf der SAP Business Technology Plattform. Frühere SAP-Standards zur Erstellung von web-basierten Anwendungen waren eng gekoppelt mit den Backendsystemen und wurden in den jeweiligen Programmiersprachen erstellt – wie z.B. WebDynpro für Java oder WebDynpro für ABAP. Der vollständige HTML-Code, inklusive der darzustellenden Daten, wurde serverseitig generiert und das Resultat an das Endgerät übermittelt und dort durch den Browser interpretiert [Eng20, S.46]. Aufgrund der Notwendigkeit des so genannten Server-Roundtrips hatte diese Technologie einige Nachteile:

- Enge Kopplung von Daten und Darstellung.
- Aufgrund der Komplexität musste der generierte HTML-Code server-seitig gerendert werden. Deshalb war es möglich, dass die Anwendung auf dem

Endgerät nicht optimal zur Darstellung kam.

- Es gab nur sehr eingeschränkte Möglichkeiten, die Benutzeroberfläche zu gestalten.
- WebDynpro unterstützt keine Gestensteuerung oder sonstige Technologien, die für mobile Endgeräte notwendig sind.
- Wenn die Bandbreite zwischen dem Endgerät und dem Server unzureichend ist, kann die Wartezeit sehr lang sein.

Seit 2012 verfolgt SAP einen neuen Ansatz für webbasierte Anwendungen. Die Daten und ihre Bereitstellung als OData-Service werden von der eigentlichen Darstellung im Browser getrennt. Der OData-Service dient als Kommunikator zwischen Backend und Frontend und wird von der UI konsumiert.

Der erste Schritt dieses Ansatzes wurde in der On-Premise-Welt mit SAP Netweaver Gateway realisiert. Danach wurde das Konzept auch in die Cloud überführt und bietet dort die Möglichkeit, eigene OData-Services bereitzustellen oder Services aus der On-Premise-Welt zu integrieren. Dieser Ansatz hat viele Vorteile:

- Durch die Trennung von Daten und Benutzeroberfläche kann die UI sehr flexibel gestaltet werden.
- Die einmal bereitgestellten Daten können von unterschiedlichen Frontend-Applikationen genutzt werden.
- Die UI kann in unterschiedlichen Technologien und auf unterschiedlichen Endgeräten erstellt werden und dort auch native (mobile) Funktionen unterstützen.
- Die reine Bereitstellung der Daten auf dem Server ist deutlich schneller und demnach sind die Wartezeiten kürzer.

Abbildung 2.1 zeigt die Architektur einer moderner SAP-Anwendung in der SAP Business Technology Plattform. Auf der mittleren Ebene befindet sich die SAP Business Technology Plattform, welche die zentrale Plattform zur Bereitstellung von Daten für webbasierte Anwendungen ist. In der BTP lassen sich eigene Datenbanken halten und eigene Services zur Verfügung stellen. Es ist jedoch ebenfalls möglich, Daten aus der SAP On-Premise-Welt (via Cloud Connector) oder auch von Drittanbieter-Systemen zentral zu integrieren. Die Daten werden jeweils als REST oder OData-Services zur Verfügung gestellt und können von Anwendungen auf der Benutzerseite konsumiert werden.

Für diese Bachelorarbeit wird ein OData-Service auf der SAP Business Technology Plattform erstellt. Auf die Anbindung eines On-Premise-Systems oder

das eines Drittanbieters wird an dieser Stelle verzichtet. Für die Erstellung des Backend-Parts (Datenbank + OData-Service) wird auf Fiori Elements und das SAP Cloud Application Programming Model (kurz: SAP CAP) zurückgegriffen. Zwar stehen auf der BTP technologisch auch andere Backend-Frameworks zur Verfügung, der Entwicklungsansatz mit SAP CAP und Fiori Elements ist durch die native Integration in das Business Application Studio als Low-Code-Entwicklungsumgebung jedoch der Quasi-Standard.

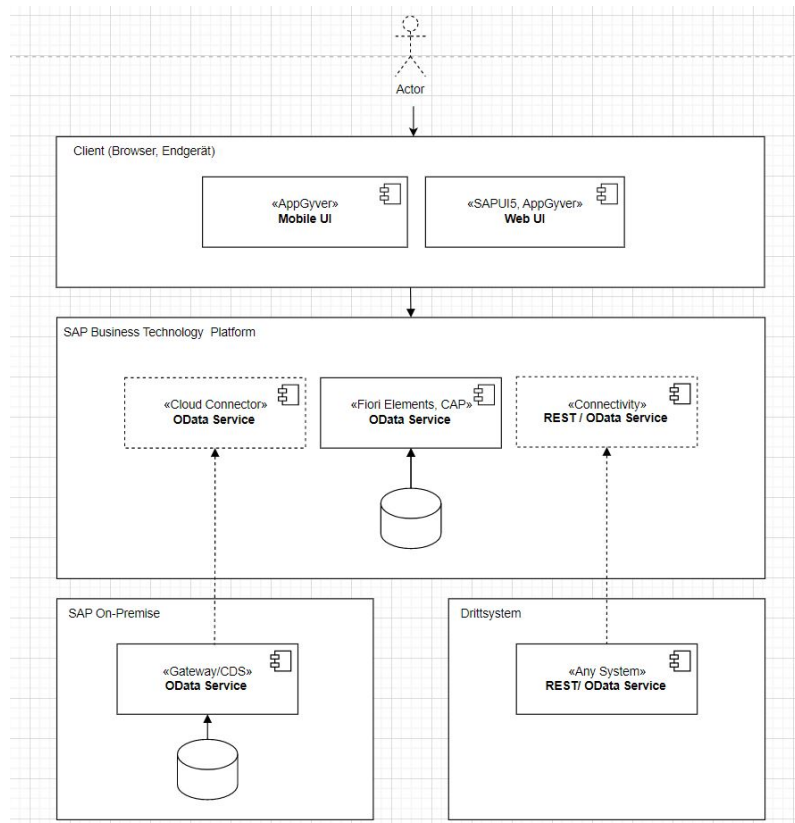


Abbildung 2.1: Architektur einer SAP-Anwendung in der SAP BTP

2.3 SAP Fiori

SAP Fiori wurde von der SAP als visuelle Leitlinie eingeführt, um User Interfaces über unterschiedliche Anwendungen hinweg zu standardisieren. Hinter dem Begriff verbergen sich jedoch ebenfalls technologische Aspekte, wie beispielsweise SAP Fiori Elements.

Das Grundkonzept von SAP Fiori ist es, Benutzeroberflächen so zu gestalten, dass Nutzer von Geschäftsanwendungen in ihrer täglichen Arbeit bestmöglich unterstützt werden, unabhängig davon, welche Endgeräte sie benutzen. Im Mit-

telpunkt stehen dabei Themen wie Usability, die Haptik und die User Experience der Anwendungen und folgende Grundsätze [Eng20, S.31]:

- Eine SAP-Fiori-App stellt dem Anwender nur die Funktionen zur Verfügung, die seiner Rolle entsprechen, sodass er nicht durch irrelevante Optionen abgelenkt wird.
- Mit SAP Fiori können Anwender sowohl auf mobilen Geräten als auch auf PCs arbeiten, wobei die Fiori-Anwendungen an das jeweilige Gerät angepasst werden müssen.
- SAP-Fiori-App statuen exakt die Funktionen des Anwendungsfalles aus. Die Funktionen, die nicht für Abarbeiten des Anwendungsfalles erforderlich sind, werden nicht in die Fiori-App ausgerichtet.
- SAP Fiori folgt einer einheitlichen Interaktion Designsprache und verfügt über ein standardisiertes Oberflächendesign.
- Die wesentlichen Funktionen der Fiori-Apps sollten für den Anwender intuitive bedienbar sein. Die Fiori-Apps sollen auch ansprechend sein [Eng20, S.34-35].

Der Fokus auf spezialisierte Applikationen macht es notwendig, diese zentral bereitzustellen. Das SAP Fiori Launchpad ist deswegen der zentrale Bereich, in welchem die SAP Fiori-Anwendungen zusammengeführt werden. Es stellt den Fiori-Apps Services wie Navigation und Anwendungskonfiguration zur Verfügung. Das Launchpad ist rollenbasiert und muss anwenderspezifisch angepasst werden. Die Rolle des Anwenders definiert somit, welche Fiori-App auf den Launchpad angezeigt werden [Gui22].

SAP Fiori als Design-Richtlinie wird im weiteren Verlauf nicht weiter betrachtet. Die Grundsätze finden sich jedoch in SAP Fiori Elements und auch in SAPUI5 wieder.

2.4 SAP AppGyver

2.4.1 Grundlage von AppGyver

AppGyver ist ein Pionier in der LCNC-Entwicklung. Das Unternehmen mit Hauptsitz in Helsinki wurde im Jahr 2010 gegründet und hat seit Gründung den Fokus auf der LCNC-Entwicklung [Lea22]. Mit Composer Pro stellt AppGyver eine zentrale Entwicklungsumgebung bereit, um Anwendungen für unterschiedliche Geschäftsprozesse und Anwendungsszenarien zu entwickeln, ohne eigenen Code zu schreiben. Diese Anwendungen können nicht nur als Webanwendungen, sondern auch als mobile Anwendungen eingesetzt werden. AppGyver unterstützt

sowohl iOS mit Bereitstellung der Anwendungen im App Store als auch Android Phone mit Bereitstellung in Google Play.

Im Februar 2021 wurde AppGyver von SAP übernommen und seitdem gibt es 2 Editionen: die Community Edition und die SAP Enterprise Edition. Die Community Edition basiert auf der initialen Version von AppGyver und bleibt zunächst unabhängig von SAP. Die Benutzer können es weiterhin kostenlos nutzen. Die SAP Enterprise Edition dagegen wird in das SAP-Ökosystem integriert und ist Bestandteil der SAP BTP. Zu den zusätzlichen Funktionen der Enterprise-Version zählen:

- Integration mit der SAP BTP Authentifizierung für Webanwendungen direkt in AppGyver.
- Erweiterte Integration von Daten aus anderen SAP-Systemen.
- Neue Enterprise-Funktionen, wie beispielsweise die Einführung einer Übersetzungsvariablen.
- Nutzer können das Projekt in Echtzeit mit anderen teilen

Am 15. November 2022, während der Anfertigung dieser Arbeit, erfolgte ein Rebranding von SAP AppGyver in SAP Build Apps. Zudem wird das Tool in Zukunft Teil einer Suite an Applikationen unter dem Label SAP Build sein, welche den Fokus auf die gesamtheitliche LCNC-Entwicklung von Anwendungen, die Automatisierung von Prozessen sowie das Design von Unternehmenswebsites legt. Neben SAP Build Apps sind auch Build Process Automation und Build Work Zone in SAP Build enthalten [Lea22]. Neben der reinen Integration, wird SAP Build App zudem in Zukunft um neue Funktionen erweitert, wie beispielsweise "Visual Cloud Functions", die die Speicherung von Daten in der Cloud und die Ausführung von Geschäftslogik ermöglichen [App22b]. Diese Erweiterungen werden jedoch im Rahmen dieser Thesis nicht weiter betrachtet.

2.4.2 Entwicklungsumgebung: Composer Pro

Eine Entwicklungsumgebung ist eine Zusammenstellung von Funktionen und Werkzeugen, die zur Entwicklung einer Anwendung notwendig sind. Werden diese gesamtheitlich und zentralisiert (via Internet) bereitgestellt, dann spricht man auch von einer Entwicklungsplattform. Composer Pro ist die zentrale Entwicklungsplattform von AppGyver. Dabei handelt es sich um eine spezialisierte LCNC-Plattform, mit der Anwendungen visuell und ohne Programmierung erstellt werden können. Der Aufbau der Plattform und die Funktionen sind dabei an die Zielgruppe, Citizen Developer ohne Programmiererfahrung, angepasst. Dennoch ist ein Verständnis des grundlegenden Aufbaus der Umgebung, sowie

der Prinzipien zur Entwicklung einer Anwendung notwendig.

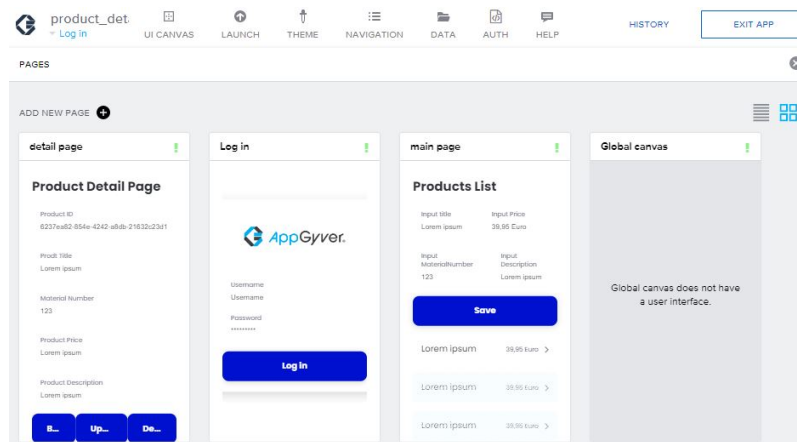


Abbildung 2.2: Pages einer AppGyver-Anwendung

Eine Anwendung in AppGyver ist grundsätzlich in mehrere Pages unterteilt. Jede Page besitzt einen eigenen Canvas, auf dem weitere Inhalte platziert werden können. Am oberen Rand der Benutzeroberfläche befindet sich die zentrale Toolbar, über die der Entwickler auf alle Ressourcen und Werkzeuge von AppGyver zugreifen kann.



Abbildung 2.3: Toolbar in Composer Pro

Dem LCNC-Ansatz folgend, finden sich in AppGyver keine programmatischen Bausteine (Code-Files, Klassen, etc.), sondern die diversen Funktionalitäten sind in eigenen, proprietären, Strukturen abgelegt. Dabei lassen sich diese grob in die Schichten des MVC-Modells einteilen. Das MVC-Paradigma strukturiert die Implementierung einer Anwendung in folgende drei Schichten:

- **M** steht für Model und repräsentiert das Datenmodell. Das Datenmodell stellt die relevanten Daten bereit.
- **V** bezieht sich auf View, d.h. die Präsentation. Diese Schicht ist zuständig für die Darstellung auf den Endgeräten und die Realisierung der Benutzerinteraktionen.
- **C** steht für Controller, also die Steuerung. Controller steuern und verwalten die Views. Der Controller kommuniziert mit dem Modell, wenn eine Benutzeraktion mit einer Datenänderung stattfindet.

View

Applikations-Schicht	AppGyver-Baustein
View	Page; View Component; Properties; Theme
Controller	Logic Flows; Formula Functions
Model	(Data) Variables; Data Resource

Tabelle 2.1: AppGyver-Baustein in die Schichten des MVC-Modells

Eine Anwendung in SAP AppGyver besteht aus mehreren Pages, d.h. mehreren eigenen Sichten. Diese werden aus vorgefertigten und konfigurierbaren View Components zusammengesetzt. Der Komponentenbibliothek in Composer Pro bietet einen Überblick über alle verfügbaren Komponenten. Diese sind in drei Registerkarten unterteilt. Unter CORE sind die Kernkomponenten verfügbar, die in den meisten Anwendungen verwendet werden. Dies sind beispielsweise Texte, Buttons oder Input-Felder. Unter BY ME sind die Komponenten aufgelistet, die der Entwickler selbst für diese Anwendung erstellt hat. Komponenten, die aus dem Marketplace für diese Anwendung hinzugefügt wurden, sind auf der Registerkarte *INSTALLED* zu finden [App22c]. Jede View Component besitzt spezifische Eigenschaften (Properties), die sich in dem kontext-sensitiven Panels „Component Properties“ und „Style“ angepasst werden können. Der Layout-Tree, der sich unten in der rechten Seitenleiste befindet, zeigt die komplette Struktur der Komponenten in der Anwendung an und ermöglicht die direkte Auswahl. Weiterhin ist es möglich, einen grundlegenden Theme mit Styles bereitzustellen, der die grundlegenden Farben, Schriften, etc. für die Anwendung festlegt.

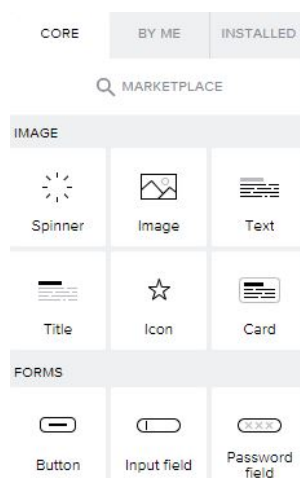


Abbildung 2.4: View-Components in Composer Pro

Model

AppGyver unterstützt in Bezug auf die Datenintegration zwei Szenarien: Es können Daten lokal im Projekt angelegt und abgespeichert werden. Diese Daten sind dann jedoch nur für die lokale Applikation gültig. Alternativ kann auf externe Datenendpunkte (REST, OData) zugegriffen werden. Die Daten werden dann über die externen Schnittstellen abgefragt und können in der Anwendung angezeigt werden. Zur Abbildung von Datenstrukturen und -flüssen gibt es in Composer Pro das Konzept der Variablen. Damit können ohne Programmierung verschiedene Arten von Strukturen festgelegt werden. „Page Variablen“ existieren nur für die aktuelle Seite und enthalten beliebige Werte, während „App Variablen“ über die gesamte Anwendung hinweg existieren. „Data Variablen“ befinden sich ebenfalls nur auf der aktuellen Seite und beinhalten die Funktion, interne und externe Datenstrukturen zu mappen. Diese Variablen können an die spezifischen Eigenschaften (Properties) einer View Component gebunden werden (Data-Binding) und stellen somit das Bindeglied zwischen Model und View dar.

Weitere Variablen zur Ablage von Werten sind „Page Parameter“ (schreibgeschützte Textvariablen), die zur Übertragung von Daten zwischen den Seiten verwendet werden können und „Translation Variablen“, die für sprachabhängige Texte verwendet werden können.

Controller

Neben den Daten ist auch die Geschäftslogik ohne oder mit geringen Programmierkenntnissen umsetzbar. SAP AppGyver stellt hier Logische Ablauffunktionen bereit, mit denen per Drag&Drop, sowie einer Konfiguration, komplexere Prozesse definiert werden können. Der Logic-Canvas befindet sich in der unteren Hälfte der Benutzeroberfläche. Für jede Komponente gibt es einen eigenen Logic-Canvas-Kontext, der eine spezifische Konfiguration erlaubt. Damit auf Nutzereingaben reagiert werden kann, stellen die View Components Events zur Verfügung. Diese werden immer dann geworfen, wenn eine spezifische Interaktion auftritt.

Weiterhin existieren in AppGyver Formeln, die dazu genutzt werden können, um komplexere Logiken abzubilden. Dazu gibt es einen eigenen Formel-Editor, in dem die Logiken hinterlegt werden können. Tatsächlich stehen dort Formel-Typen zur Verfügung, die sich nahe an der richtigen Programmierung bewegen (Logische Operatoren, IF-Statements, etc) [App22a].

todo: Abbildung 2.3 Exemplarische Formel in AppGyver

2.5 SAPUI5

2.5.1 Grundlage von SAPUI5

SAP Fiori beschreibt als Leitfaden die Entwicklungsrichtlinie zur Erstellung von Anwendungen im SAP-Umfeld. Für die Umsetzung sind jedoch auch technische Bausteine notwendig. Mit SAP WebDynpro, der vormals führenden UI-Technologie, ist SAP Fiori aber schwer umzusetzen, da der HTML-Code auf dem Server generiert und dann an das Endgerät übermittelt wird. In diesen Kontext ist die Entwicklung eines clientseitigen Ansatzes erforderlich. Bei einem clientseitigen Ansatz steht der Frontend-Server im Mittelpunkt der Kommunikation und lädt das UI-Framework, das die weiteren Verarbeitungsschritte übernimmt [Eng20, S.45-47].

SAPUI5 (kurz für: SAP UI Development Toolkit für HTML5) ist ein JavaScript-basiertes clientseitiges Framework und eine UI-Bibliothek, mit der SAP Fiori-Anwendungen sehr flexibel entwickelt und auf verschiedene Plattformen portiert werden können. Die SAPUI5-Bibliothek basiert auf modernen Web-Standards, wie JavaScript, HTML5 und CSS3 [Ant16, S.139]. SAPUI5 verfügt über mehr als 500 UI-Controls, die an den neuesten SAP Fiori Design Guidelines ausgerichtet sind, und bietet integrierte Unterstützung für Enterprise-Funktionen, wie Datenbindung, Routing, Message Handling, Mehrsprachigkeit, etc. Heute ist SAPUI5 der Standard für die Implementierung von Frontend-Anwendungen im SAP-Umfeld [Com22a].

Da es sich bei SAPUI5 nicht um eine LCNC-Technologie handelt, ist ein grundlegendes Programmierverständnis in der Sprache JavaScript notwendig. Zudem setzt SAPUI5 das Verständnis einiger Entwurfsmuster voraus. Das MVC-Paradigma beispielsweise strukturiert die Implementierung von SAPUI5-Anwendungen in drei Schichten, die sich so auch im Quellcode wiederfinden.

- **Model:** Hier stehen spezielle Klassen zur Verfügung, die den Zugriff auf unterschiedliche Arten von Daten abstrahieren (JSONModel, ODataModel, XMLModel).
- **View:** Die View beinhaltet den Aufbau des User Interfaces und SAPUI5 stellt hier UI Controls zur Verfügung, dafür genutzt werden.

Diese lassen sich via JavaScript, HTML oder XML definieren und konfigurieren.

- **Controller:** Zu jeder View gibt es in SAPUI5 einen Controller, der eine Benutzeraktion über Events und zugehörige Callback-Implementierungen steuert, sowie komplexere Geschäftslogik enthalten kann [Ant16, S.149].

Da es sich bei SAPUI5 um ein komplexeres Framework handelt, können an dieser Stelle nicht alle Facetten im Detail erklärt und beschrieben werden. In Kapitel 3.3 werden ihm Rahmen der Implementierung jedoch einige genutzte Dinge vorgestellt.

2.5.2 Entwicklungsumgebung: Visual Studio Code

Durch den freien Programmieransatz ist SAPUI5 nicht an eine Entwicklungsumgebung/-plattform gebunden. Im Rahmen dieser Thesis wird Visual Studio Code (kurz: VS-Code) für die Umsetzung verwendet. VS-Code ist ein Quellcode-Editor von Microsoft, der im Jahr 2015 für verschiedene Betriebssysteme wie Windows, MacOS und Linux veröffentlicht wurde. Er unterstützt diverse Programmiersprachen und Frameworks (z.B. JavaScript, TypeScript und Node.js), kann jedoch über das Erweiterungskonzept um weitere Programmiersprachen, Laufzeiten und Funktionen ergänzt werden [Cod22a]. VS-Code ist heute aufgrund der ausgereiften Funktionen und der guten Bedienung ein Standard in der Entwicklung von Web-Anwendungen [Wik22c] und wird deswegen an dieser Stelle potenziellen anderen Entwicklungsumgebungen vorgezogen.

Der Arbeitsbereich des VS Code besteht aus ein oder mehreren Verzeichnissen. Es vereinfacht die sprachübergreifende Anwendungsentwicklung in einer integrierten Entwicklungsumgebung. VS Code integriert ein voll funktionsfähiges Terminal mit dem Editor, um Shell Skript und Kommanden durchzuführen. Das integrierte Terminal kann verschiedene Shells verwenden, die auf dem Rechner installiert sind [Cod22b].

Die lokale Implementierung der SAPUI5-Anwendung in VS-Code erfordert die zusätzliche Installation von weiteren Bibliotheken: das SAP Cloud Application Programming Model (kurz: SAP CAP) und des Easy UI5 Generators. Das SAP CAP ist ein Framework zur Erstellung von Backend-Anwendungen und kommt auch bei Fiori Elements zum Einsatz [SAP22a]. Mit Hilfe von Core Data Services als die universelle Modellierungsspra-

che für Domänenmodelle und Servicedefinitionen, können in sehr kurzer Zeit Datenmodelle generiert und als OData-Service bereitgestellt werden [SAP22a]. Der Easy UI5 Generator enthält Vorlagen zur Erstellung einer SAPUI5-Anwendung mit aktuellen Best Practices. So lassen sich bereits die grundlegenden Strukturen einer Anwendung erstellen, auf deren Basis die Entwicklung dann stattfinden kann [SAP22c]

Um die UI5-Anwendung zu implementieren, muss die Entwicklungsumgebung wie folgt eingerichtet werden:

- Herunterladen und Installieren des aktuellen Node.js Installationspakets von <https://nodejs.org/en/download/> inklusive Runtime und Package Manager (npm). Node.js bietet eine asynchrone, ereignisgesteuerte Laufzeitumgebung für Javascript und wird für das lokale Betreiben eines Webservers verwendet [Wik22a].
- Installieren von yeoman und dem Easy-ui5 Generator. Yeoman ist ein Kommandozeilen-Scaffolding-Tool für Node.js, um das Gerüst für die weitere Entwicklung der SAPUI5-Anwendung zu generieren
- Installieren der SAP Cloud Application Programming Model-Module (@sap/cds-dk und @sap/cds).
- Herunterladen und Installieren der SQLite-Datenbank-Treiber. Als ein leicht eingebettetes Datenbanksystem, lässt sich SQLite direkt in entsprechende Anwendungen integrieren, ohne eine weitere Server-Software [Wik22b].
- Installation der VS Code Erweiterungen für SAP Fiori und SAP CAP CDS, zur Unterstützung bei der Entwicklung.

Das cds-dk und SQLite werden benötigt, um bei der lokalen Entwicklung einen OData Mock Service zu erzeugen. Die VSCode-Erweiterungen bieten Sprachunterstützung für die Core Data Services (CDS), welche die Grundlage des SAP Cloud Application Programming Model (CAP) bilden, sowie für SAPUI5 [Mar22].

2.6 Fiori Elements

2.6.1 Grundlage von Fiori Elements

SAP Fiori als gestalterische Richtlinie lässt sich mit SAPUI5 technisch umsetzen. Allerdings ist dort immer Programmieraufwand nötig. Als LCNC-Ansatz stellt die SAP jedoch mit Fiori Elements eine Technologie zur

Verfügung, mit der UI Controls und sogar komplette Applikationen automatisch auf Basis von Metadaten zur Laufzeit generiert werden können. Die Metadaten werden dabei via OData-Annotationen beschrieben, die vom Backend-Service bereitgestellt werden müssen [Eng20, S.48]. SAP Fiori Elements basiert technologisch auf SAPUI5 und erweitert dieses um intelligente Komponenten und Views. SAP Fiori Elements umfasst sogenannte Floorplans, d.h. Grundrisse und UI-Patterns für gängige Anwendungsfälle. Fiori Elements verfügt über die folgenden vier grundlegenden Floorplans:

- List Report und Object Page

Ein List Report wird verwendet, wenn Elemente in einer Tabelle oder Liste dargestellt werden sollen. Mit dem List Report können die Objekte angezeigt, gefiltert und bearbeitet werden. Der List Report wird in der Regel in Verbindung mit der Objekt Page verwendet. Auf die Objekt Page kann der Nutzer durch Klicken auf ein einzelnes Element in der Liste gelangen und dort detaillierte Informationen über einzelne Objekte angezeigt bekommen und es bearbeiten.

todo: Abbildung 2.4 List Report und Object Page

- Worklist

Eine Worklist zeigt ebenfalls eine Liste von Elementen an, die von Benutzer bearbeitet werden sollen. In einer Worklist ist jedoch keine komplexe Filterung möglich und die Anzeige, bzw. das Editieren erfolgt ausschließlich in der Worklist-Ansicht [Doc22b].

- Overview Page

Ein Overview Page ermöglicht es, den Nutzern eine große Menge unterschiedlicher Informationen für einen Überblick bereitzustellen. Unterschiedliche Informationen werden in verschiedenen Cards visualisiert. Eine Card zeigt die Details zu einem bestimmten Geschäftsobjekt. Mit der Overview Page können Anzeigen, Filtern und Verarbeiten von Daten einfach und effizient gemacht werden.

todo: Abbildung 2.5 Overview Page

- Analytical List Page

Die Analytical List Page ist die Weiterentwicklung von List Report und

verfügt über mehr Funktionen: Daten können in verschiedene Perspektiven analysiert werden, z.B. durch Drill-Downs zur Ursachenforschung [Doc22b].

Mit Hilfe von Extension Points in den Floor-Plans können in einer Fiori Element-Anwendung zusätzliche Funktionalitäten hinzugefügt werden. Dafür ist jedoch immer eine Programmierung erforderlich. SAP Fiori Elements ohne Extensions benötigt keinen Programmieraufwand, setzt jedoch einen OData-Service und eine Konfiguration der Annotationen voraus. Hierfür stehen potentiell unterschiedliche Technologien zur Verfügung. Dank der SAP-eigenen Entwicklungsumgebung, SAP Business Application Studio, können jedoch komplette SAP Fiori Elements-Anwendungen inklusive der Definition von Datenstrukturen und OData-Services und Annotationen visuell aufgebaut werden [Doc22a].

2.6.2 Entwicklungsumgebung: Business Application Studio

Grundsätzlich kann SAP Fiori Elements mit verschiedenen Entwicklungsumgebungen erstellt werden. Durch die sehr gute Integration und Bereitstellung einer LCNC-Umgebung, eignet sich das SAP Business Application Studio (BAS) jedoch sehr gut für das Erstellen einer Anwendung im Kontext dieser Thesis. BAS ist ein Service der SAP Business Technology Platform (SAP BTP), der seit Februar 2020 als Nachfolger der SAP Web IDE zur Verfügung steht. SAP BAS ist ein Cloud-basiertes Werkzeug, das nicht lokal installiert werden kann und im Browser des Nutzers ausgeführt wird und entspricht damit den Kriterien einer Entwicklungsplattform [Por22].

Das Business Application Studio lässt sich via Dev Spaces für unterschiedliche Anwendungsarten konfigurieren. Die Entwicklungsszenarien umfassen zum Beispiel SAP Fiori, SAP S/4HANA Erweiterungen, Workflows und SAP HANA-Anwendungen. In jedem Dev Space werden je nach Auswahl eine Reihe von vordefinierten Erweiterungen installiert, die spezialisierte Funktionen bereitstellen [Por22].

Neben der klassischen Code-basierten Entwicklung, stellt BAS auch die Möglichkeit bereit, Low-Code basierte „Full-Stack Anwendungen“ zu entwickeln. Dafür sind dann diverse Wizards und UI-Masken vorhanden, die im Hintergrund automatisch den notwendigen Quellcode interpretieren und erstellen. Eine Full-Stack-Anwendung besteht dabei aus dem Datenmodell und OData-Service des SAP Cloud Application Programming Models und SAP Fiori Elements als Frontend-Technologie. Alle notwendigen Parameter

lassen sich via UI konfigurieren.

Im Rahmen dieser Arbeit wird ein Dev Space erstellt, der als Entwicklungsumgebung für die Low-Code-basierte Full-Stack-Anwendung ausgewählt wurde, um den in Kapitel 1, Abschnitt 1.3 beschriebenen Anwendungsfall mit Fiori-Elements zu entwickeln.

Kapitel 3

Konzept der Transformationsengine

Das Ziel dieser Bachelorarbeit ist es, eine Transformationsengine zu bauen, die bestimmte Daten von verschiedenen Produkten in eine vorgegebene Excel-Vorlage schreibt. Nach einer allgemeinen Vorstellung der Problemstellung und der Rahmenbedingungen, werden die Konzepte zu drei möglichen Lösungsansätzen näher erläutert. Die letzten beiden werden später in Kapitel 4 implementiert.

	A	B	C	D	E	F	G	H	I	J	K
1	查登資訊		廠商聯絡資訊		器材識別(DI)資訊						
2	許可證字號(類型)	許可證字號	廠商聯絡電話	廠商電子郵件	UDI發碼機構	基本DI	內含數量	使用單位層級DI識別碼	型號	產品描述	DI 資料發
3	License Type	License No	Phone	Email	Issuing Agency	Primary DI Number	Device Count	Unit of Use DI Number	Catalog Number	Device description	DI Reco Publish D
4	衛部醫器製	008888	(02)11520520	service@MIT-Shield.com	GS1	0081255071998	1		MIT-Shield T1	神盾AAA醫用防護衣符合感染控制防護服規範，具有良好的液體阻隔特性，可降低疾病傳播、交叉感染	
5											

Abbildung 3.1: Ausschnitt 1 der Excel-Vorlage von Taiwan

3.1 Allgemeines

Die Transformationsengine soll ausgewählte Produktdaten aus der UDI Platform von p36 in Excel-Arbeitsmappen schreiben. Form, Struktur und Inhalt der Excel-Dateien ist dabei behördlich vorgegeben. Die erfolgreich ausgefüllten Dateien werden später bei den Behörden hochgeladen, um die Produkte samt zugehöriger Informationen in deren Datenbanken zu registrieren – dies findet allerdings außerhalb der Transformationsengine statt.

Die verschiedenen Behörden publizieren jeweils ihre individuelle Excel-Vorlage, die zum Import verwendet werden muss. Wie bereits in Kapitel ?? erläutert, führen in den nächsten Jahren verschiedene Gesundheitsministerien ein UDI-System ein, bei dem man die Produkte per Excel-Datei registrieren muss. Als Erstes stand die SFDA aus Saudi-Arabien auf der Agenda von p36, bis sie die Veröffentlichung des Excel-Templates kurzfristig um ein Jahr verschoben haben. Dies ist leider gängige Praxis seitens der Behörden, sodass es umso wichtiger ist, dass die Plattform flexibel ist. Für Ende des Jahres ist nun die Einbindung von Taiwan geplant – hier existiert bereits eine Vorlage inklusive Übersetzung. Drei Auszüge mit einem eingetragenen Beispielprodukt sind in den Abbildungen 3.1, 3.2 und 3.3 zu sehen.

	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE
1	產品特性		標識上之PI資訊				注意事項資訊			滅菌方式		
2	是否為單次 使用器材	是否可重複 使用器材	產品標示是 否有批號	產品標示是 否有序號	產品標示是否 有製造日期	產品標示是否 有有效期間 (保存期限)	產品標示是否 含有天然橡膠 (乳膠)成份	產品標示是否含有 DEHP(塑化劑)成分	特殊儲存 條件	本器材是否 為滅菌包裝	本器材是否 為使用前滅 菌器材	本器材之滅 菌方式
3	For Single- Use	For Multiple- Use	Lot or Batch Number	Serial Number	Manufacturing Date	Expiration Date	Device required to be labeled as containing natural rubber latex or dry natural rubber	Device required to be labeled as containing DEHP	Special Storage Conditions	Device packaged as Sterile	Requires Sterilization Prior to use	Sterilization Methods
4	Y	N	Y	N	Y	N	N	N			Y	乾熱 Dry Heat, 環氧乙烷 Ethylene Oxide, 濕熱或蒸氣 Moist Heat or Steam
5												

Abbildung 3.2: Ausschnitt 2 der Excel-Vorlage von Taiwan

Die geforderten Datenelemente sind dabei vielschichtig – von der UDI und deren Vergabestelle, über Informationen zu Hersteller, Modell, Material und Inhaltsstoffe bis hin zur Produktion, Lagerung oder Sterilisationsme-

	A	B	C	D	E	F	G	H
1	儲存和運作資訊							
2	許可證字號(類型)	許可證字號	基本DI	器材儲存/使用環境要求類別	器材儲存/使用環境要求之下限	器材儲存/使用環境要求之上限	器材儲存/使用環境要求之單位	
3	License Type	License No	Primary DI Number	Storage and Handling	Low value	High value	Unit of Measure	
4	衛部醫器製	008888	00812550719998	溫度 (Temperature)	-20	40	攝氏度 (Degrees Celsius)	
5	衛部醫器製	008888	00812550719998	濕度 (Humidity)	20	55	相對濕度百分比 (Relative Humidity Percentage)	
6								
7								
8								
	產品資訊	包裝層級DI資訊	臨床尺寸規格	儲存和運作資訊	資料參照表			

Abbildung 3.3: Ausschnitt 3 der Excel-Vorlage von Taiwan

thoden sowie viele weitere Details. Einige Beispiele sind in Abbildung 3.4 aufgelistet. Momentan unterschneidet p36 insgesamt fast 200 verschiedene Datenelemente für fünf Behörden (EU, FDA, NMPA, MFDS und SFDA). Diese sind in den Agency-Device-Models (ADM) implementiert, zusätzlich gibt es das übergeordnete Common-Device-Model (CDM), welches die gängigsten Elemente einer hypothetischen „typischen“ Behörde in sich vereint. Unter den Datenelementen befinden sich auch 15 sogenannte komplexe Elemente, die wiederum aus einfachen Elementen bestehen. Mit jeder zusätzlichen Behörde ergeben sich in der Regel neue Elemente, außerdem können Kunden Anpassungen und Erweiterungen initiieren (die allerdings für die Weiterleitung an die Behörde nicht maßgeblich sind bzw. sein dürfen). Die Daten sind intern im JSON-Format gespeichert.

1	Element label	Property of	Type
2	Issuing Agency		select box
3	UDI-DI		text
4	Device Count		number
11	Catalog Number		text
12	Device Description		free text
14	Material		text
16	DI Record Publish Date		date
21	Device Subject to Direct Marking		yes / no
37	Product Codes		multiple values from select box and free text
129	Package Information		table
130	Package DI	PackageInformation	text
131	Quantity per package	PackageInformation	number
133	Package Type	PackageInformation	text
149	Storage and Handling		table
150	Type	Storage	select box
151	Storage Condition	Storage	text in different languages
152	Low Value	Storage	decimal number
153	High Value	Storage	decimal number
154	Unit Of Measure	Storage	select box

Abbildung 3.4: verschiedene Datenelemente

Es gibt viele fundamentale Produktinformationen, die von allen Behörden gefordert werden, aber auch einige Unterschiede besonders bei spezielleren Daten. Die am häufigsten verwendeten Elemente hat p36 im CDM zusammengefasst.

Zusätzlich können die Datentypen oder -formate variieren, wenn eine Behörde bestimmte Werte für ein Datenelement erwartet oder bei lokalen Datums-/Zeitangaben. Dies umfasst ebenfalls die Kodierung von Wahrheitswerten: Während manche Behörden direkt mit **true** und **false** als Boolean arbeiten, akzeptieren andere zum Beispiel nur **Y** bzw. **N** als valide Eingabe, so wie es in Taiwan der Fall ist (siehe Abbildung 3.2). Des Weiteren kann sich die Wertemenge individueller Datenelemente unterscheiden, denn verschiedene Behörden kodieren bestimmte Eigenschaften mit anderen Wertelisten. Dazu müssen zunächst die p36-internen Werte in die behördenspezifische Sprache „übersetzt“ werden.

Neben den einfachen Datenelementen mit 1:1-Beziehung, die in einer einzigen Zelle abgebildet werden können, gibt es auch die bereits erwähnten komplexen Datenelemente mit 1:n-Beziehung. Diese werden in zusätzlichen Arbeitsblättern über ggf. mehrere Zeilen angegeben, wie z. B. anhand der beiden Lagerungswerte in Abbildung 3.3 zu erkennen ist. Hier hat ein Produkt zwei Einträge beim komplexen Element „Storage and Handling“ – es sind Informationen zur Temperatur sowie zur Luftfeuchtigkeit hinterlegt. Der Schlüssel für die einzelnen Arbeitsblätter ergibt sich in der Regel aus der eindeutigen UDI-DI (hier mit „Primary DI Number“ bezeichnet, vgl. Kapitel ??); als zusätzliche Informationen werden in diesem Fall noch Lizenztyp und -nummer wiederholt. Darüber hinaus sind auch exotischere Einträge möglich, wie z. B. eine Liste von Sterilisationsmethoden in Spalte AE von Abbildung 3.2. Hier wird das komplexe Element „Sterilisation“ zu einem Eintrag in einer Zeile verschmolzen.

Rot eingefärbte Spaltennamen sind obligatorisch, schwarz beschriftete Spalten optional. Welche Elemente verpflichtend sind, variiert von Behörde zu Behörde.

Die meisten großen Medizinprodukte-Hersteller haben nicht nur einige wenige Produkte am Markt platziert, sondern bieten vornehmlich eine breite Produktpalette an, die sich zusätzlich durch eine Vielzahl unterschiedlicher Ausprägungen und Variationen weiter auffächert – und das in den verschiedensten Verpackungseinheiten. Multipliziert man die Farben, Formen und Größen, entsteht eine enorme Menge an Devices, die alle ihre eigene UDI besitzen und im System registriert werden müssen. Hierzu empfiehlt sich der Massen-Upload. Die Transformationsengine muss in der Lage sein, viele Produkte (in der Größenordnung von 1.000–10.000) simultan in die Excel-Vorlage zu schreiben.

Zusammengefasst ergeben sich unter anderem die folgenden Anforderungen:

Nr.	funktionale Anforderungen
A-1	Transformation von Produktdaten in Excel-Vorlage
A-2	Mapping von Wahrheitswerten
A-3	Mapping von Datums- und Zeitformaten
A-4	Mapping von Wertelisten einzelner Datenelemente

Nr.	nichtfunktionale Anforderungen
A-5	generisch bzgl. verschiedener Behörden
A-6	benutzerfreundliche Wartung bei Änderungen
A-7	Massen-Upload möglich, in annehmbarer Zeit

Tabelle 3.1: Funktionale und nichtfunktionale Anforderungen

3.2 Spezifischer Ansatz

Die naheliegendste und simpelste Lösung ist es sicherlich, für jede Behörde spezifisch und individuell die Transformation hart zu kodieren.

Hierbei könnte für jede Behörde eine eigene Klasse entstehen, in der exklusiv das Format der Excel-Arbeitsmappe sowie deren individueller Inhalt spezifiziert wird. Diese Abbildung kann einfach starr aufgeschrieben werden.

Aus Zeitgründen wurde dieser Ansatz allerdings weder tiefergehend verfolgt noch implementiert, sondern er sei hier nur in der Theorie erwähnt, als Vergleichsgrundlage für die spätere Evaluierung.

Stattdessen liegt der Fokus dieser Bachelorarbeit auf den folgenden beiden generischen Ansätzen, deren Implementierung unabhängig von den Behörden funktioniert.

3.3 Generischer Ansatz mit Excel-Datei

Da p36 die Transformationsengine nicht nur für eine Behörde nutzen möchte, sondern grundsätzlich mit einer langfristigen und entsprechend generischen Denkweise agiert, müssen die behördenspezifischen Informationen extern abgespeichert werden anstatt im Quellcode enthalten sein. Die

Behördenanforderungen beschreiben quasi eine Abbildung von den Spalten der Excel-Mappe auf bestimmte Datenelemente. Inhalt der Abbildung ist dann das Produktportfolio, das man sich als ein n -dimensionales Tupel einzelner Produkte vorstellen kann, welches auf n Zeilen abgebildet wird (bei 1:1-Beziehungen).

Die erste Idee bestand darin, die Definition dieser Abbildung in die Excel-Arbeitsblätter auszulagern.

Konkret wird das mit JMESPath-Ausdrücken (siehe Kapitel ??) in der Excel-Vorlage umgesetzt. Das heißt die Solution Manager müssen zunächst in der Vorbereitung das Excel-Template der Behörde entsprechend ausfüllen und auf der Plattform hinterlegen. Dies geschieht einmalig, wenn die Behörde neu ins System aufgenommen wird bzw. immer dann, wenn Änderungen auf Seiten der Behörde oder p36 auftreten. Hierfür muss in jede Spalte, die mit Daten zu füllen ist, in der Zeile, in die das erste Produkt geschrieben werden soll, stattdessen der JMESPath-Ausdruck eingegeben werden, und zwar innerhalb spitzer Klammern. Falls die Spalte verpflichtend ausgefüllt werden muss, wird dies mit einem einleitenden Ausrufezeichen ! gekennzeichnet – der Syntax ist also **<!JMESPath-Ausdruck>**. Die Notation wurde willkürlich festgelegt und kann angepasst werden. Abbildung 3.5 gibt einen kleinen Einblick, wie eine solche Excel-Datei aussehen könnte. Die Wahl von JMESPath als Abfragesprache ist dabei weitestgehend arbiträr und sie könnte auch durch eine vergleichbare Sprache ersetzt werden.

	A	B	C	D	E	F	G	H
1		BehördenTestTemplate						
2								
3		Device	IssuingAgency	Date	CatalogNumber	Device Count	Sterilization Method	RequiresSterilizationPi
4		<Device>	<IssuingAgency>	<Date>	<CatalogNumber>	<DeviceCount>	<[Sterilizations[]].Steriliza	<RequiresSterilizationPi
5								
6								

Abbildung 3.5: JMESPath-Ausdrücke in der Excel-Vorlage einer Behörde

Diese Ausdrücke können beliebig kompliziert werden. Die ersten Spalten in Abbildung 3.5 beinhalten lediglich die Bezeichnungen der Datenelemente und sind in vielen Fällen ausreichend. Durch einen Punkt gelangt man eine Ebene tiefer, außerdem ist Filtern per Fragezeichen möglich, vgl. Abschnitt ???. Mit

<TradeNames[?TradeNameLanguage == 'EN'].TradeName>

wird beispielsweise nur der englische Handelsname in die Zelle geschrieben. Darüber hinaus gibt es noch einige Spezialfälle. JMESPath stellt verschiedene eingebaute Funktionen zur Verfügung, die man per Pipe-Symbol

| hintereinander ausführen kann, um so weitergehende Selektionen bzw. Manipulationen durchzuführen. Auf zwei Beispiele wird im Folgenden eingegangen.

- Das Datenelement **ProductionIdentifizier** fasst bei p36 verschiedene Kennwörter in einem Array zusammen. In der Excel-Datei hat allerdings jedes Kennwort seine eigene Spalte – ist der Identifikator im Array enthalten, wird dies bei der Behörde mit **true** angezeigt und entsprechend bei Absenz mit **false**. Das gewünschte Ergebnis erzielt der Ausdruck

```
<ProductionIdentifizier | contains(@,'IDENTIFIER')> ,
```

wobei **IDENTIFIER** durch das jeweilige Schlüsselwort ersetzt werden muss.

- Die bereits erwähnten Sterilisationsmethoden sollen alle zusammen in einer Zelle aufgelistet werden, jeweils mit Kommas getrennt. Dies lässt sich durch

```
<[Sterilizations[].SterilizationMethod,  
Sterilizations[].OtherSterilizationMethods] |  
[].to_string(@) | join(', ',@)>
```

realisieren. Hier werden zunächst alle Datenfelder, die Sterilisationsmethoden beinhalten, in einem Array gesammelt, danach in Strings umgewandelt und zuletzt vereinigt. Das @ dient als Platzhalter für das jeweils aktuelle Element im Array. Die Umwandlung in Strings ist nötig, weil die Funktion **join** nur für String-Arrays definiert ist.

3.3.1 Zusätzliche Einstellungen

Um ergänzende Informationen zu übertragen, wie beispielsweise die behördenspezifische Darstellung von Wahrheitswerten, wird ein zusätzliches Arbeitsblatt namens „Settings“ in die Vorlage eingefügt. Ein Beispiel zeigt Abbildung 3.6. Die Projektion der beiden Wahrheitswerte muss unter dem Schlüsselwort „Boolean“ angegeben werden. Weitere Mappings für einzelne Datenelemente sind unter der Elementbezeichnung in spitzen Klammern möglich. Die Bezeichnung muss mit dem Ausdruck übereinstimmen, der in der Spalte angegeben wurde, in der das Datenelement verwendet wird. Hier bietet sich eine Excel-interne Verlinkung an, anstatt einen längeren Ausdruck erneut abzutippen oder zu kopieren. Es muss jeweils der Wert in der p36-internen Schreibweise eingegeben werden und in der Zelle rechts

daneben der Ziel-Wert in Behördensprache. Dies ist in Zeile 11 bzw. 20 in Abbildung 3.6 zu sehen, bei der z. B. mit Zahlen kodierte Sterilisationsmethoden in ihre chinesische Bezeichnung überführt werden.

	A	B	C	D	E	F	G	H
5								
6	Boolean							
7	TRUE	yes						
8	FALSE	no						
9								
10								
11	<[Sterilizations[].SterilizationMethod, Sterilizations[].OtherSterilizationMethods]]>							
12		1 乾熱 Dry Heat						
13		2 環氧乙烷 Ethylene Oxide						
14		3 過氧化氫 Hydrogen Peroxide						
15		4 濕熱或蒸氣 Moist Heat or Steam						
16		5 輻射 Radiation						
17		6 過濾滅菌法 Filtration Sterilization						
18		7 其他						
19								
20	<MriSafetyInformation>							
21	SAFE	safe						
22	SEMI_SAFE	semi safe						
23	UNSAFE	unsafe						
24								
25								

Abbildung 3.6: Das Settings-Arbeitsblatt

Beim Einlesen der Vorlage werden die Informationen im Settings-Sheet verarbeitet, in Hashtabellen abgespeichert und das Arbeitsblatt anschließend gelöscht, sodass es letztendlich nicht bei der Behörde mit hochgeladen wird. Entscheidend beim Einlesen ist die vertikale Anordnung der Elemente bzw. ihrer Werte und dass eine Liste von Werte-Paaren jeweils mit einer Leerzeile abgeschlossen wird. Es besteht die Möglichkeit über dieses Arbeitsblatt noch weitere Informationen oder andere besondere Einstellungen einfließen zu lassen, deren Logik allerdings erst in der Transformationsengine implementiert werden muss.

Zusammengefasst besteht das Konzept also darin, dass die Metainformationen zur Abbildung zwischen Excel-Datei und Produktdaten einmalig zu Beginn in die Excel-Vorlage geschrieben werden, wenn die Behörde in die UDI Platform aufgenommen wird. Dies kann noch durch ein zusätzliches Arbeitsblatt mit besonderen „Settings“ individualisiert werden. Bei regulatorischen Änderungen muss entsprechend die Excel-Vorlage bearbeitet werden, in der Regel bleibt aber der Code der Transformationsengine unberührt und dient für alle Behörden gleichermaßen.

3.4 Generischer Ansatz mit Mapping-Datei

Eine Variation vom bisherigen Ansatz 3.3 ist es, die Abbildung von den Spalten der Excel-Datei zu den Produktdaten in einer zusätzlichen Datei zu speichern. Sie wird im Folgenden Mapping-Datei genannt. Dadurch bleibt die Excel-Vorlage der Behörde unberührt und die Informationen übersichtlicher.

Zunächst wurde als Mapping eine normale Textdatei mit selbst gestalteter Notation anhand von Keywörtern und rudimentärem Parsing gewählt, bei dem der eingelesene Text nach den Schlüsselwörtern abgesucht wird. Dies ist aber unnötig umständlich, sodass in einem zweiten Schritt auf eine YAML-Datei umgestellt wurde. Die Auszeichnungssprache YAML wurde 2001 von Clark Evens entworfen und ist in [NMA21] spezifiziert. Das YAML-Format ist sehr gut für Menschen lesbar sowie leicht zum Editieren. Im Gegensatz zu JSON oder XML ist es noch reduzierter durch die Absenz von Klammern und Anführungszeichen und damit noch übersichtlicher – der Vergleich wird in [EH11] vertieft.

Im Unterschied zu Ansatz 3.3, welcher mit **JMESPath** arbeitet, wird hier die JSON-Abfragesprache **JSONata** (siehe Abschnitt ??) verwendet. Beide Sprachen eignen sich in beiden Ansätzen und sind theoretisch austauschbar, aber um beide auszutesten und weil **JSONata** noch umfangreicher ist, wurde gewechselt. Die Vor- und Nachteile werden später in Kapitel 5 detaillierter erläutert.

Die Mapping-Datei ist strukturell in zwei Teile untergliedert.

Ein Teil beschreibt die Arbeitsblätter: Er wird mit dem Schlüssel **SheetMappings** eingeleitet und enthält eine assoziative Liste von einzelnen Arbeitsblättern, die wiederum aus Schlüssel-Werte-Paaren bestehen. Als Key dient dabei der Name des Arbeitsblattes in der Excel-Datei. Hierbei werden unter anderem auch chinesische Schriftzeichen unterstützt, was insbesondere für Taiwan ein wichtiger Punkt ist. Einige Sonderzeichen werden von Excel selbst ausgeschlossen, ebenso zu lange Namen, die mehr als 31 Zeichen enthalten. Zur Visualisierung ist in Quelltext 3.1 ein Auszug einer beispielhaften Mapping-Datei dargestellt.

```
1 SheetMappings:
2   Sheet 1:
3     row: 4
4     columns:
5       B: Device
```

```

6      C: Date
7      D: >
8          $exists(DeviceCount) ? DeviceCount : "NA"
9      E: $join(Sterilizations.SterilizationMethod.$string(),", ")
10     F: >-
11         RequiresSterilizationPriorUse ? 1 : 0
12     G: CatalogNumber
13     H: TradeNames[TradeNameLanguage = "EN"].TradeName
14     I: $contains($join(ProductionIdentifier),"BATCH_NUMBER")
15     J: |
16         "RUBBER" in ProductionIdentifier
17     K: '"DEHP" in ProductionIdentifier'
18     M: MriSafetyInformation
19     N: UnitOfUseDi
20     L: PackageInformation
21     mandatoryColumns: [B, G, D]
22
23     產品資訊:
24     row: 2
25     category: TradeNames
26     columns:
27         A: Device
28         B: IssuingAgency
29         C: 'TradeNames.($exists(TradeNameLanguage)?TradeNameLanguage:null)'
30         D: "TradeNames.($exists(TradeName) ? TradeName : null)"
31     mandatoryColumns:
32     - A
33     - B

```

Quelltext 3.1: Beispiel einer Mapping-Datei für SheetMappings

Für jedes Arbeitsblatt müssen die folgenden Informationen angegeben werden:

- Unter **row** wird die Start-Zeile gespeichert, ab der die Produktdaten eingetragen werden können.
- Dem Schlüssel **columns** ist ein Mapping aller zu füllenden Spalten zugeordnet. Hierbei dienen die Excel-Spaltennamen jeweils als Key und als Value wird der JSONata-Ausdruck angegeben, über welchen später die gewünschte Produktinformation gefiltert wird.
- In **mandatoryColumns** kann eine Liste verpflichtender Spalten übergeben werden. Dies kann kompakt in eckigen Klammern wie in Zeile 21 erfolgen oder über mehrere Zeilen mit Bindestrichen wie ab Zeile 31.
- Mit **category** kann das komplexe Datenelement angegeben werden, dessen Inhalte in das jeweilige Arbeitsblatt geschrieben werden sollen. Ist

die Kategorie gesetzt, werden nur für die Produkte Zeilen erzeugt, die tatsächlich auch Einträge unter besagtem komplexen Datenelement haben.

Dabei spielt weder die Groß- und Kleinschreibung der Spaltennamen eine Rolle, noch in welcher Reihenfolge die Wertepaare eingegeben werden. Zur Übersicht bietet sich eine alphabetische Anordnung allerdings an.

Die JSONata-Ausdrücke können beliebig komplex werden und mehrere Zeilen umfassen oder Zeichen enthalten, die man in YAML escapen muss, wie beispielsweise Doppelpunkte (:), Doppelkreuze (#) oder Anführungsstriche zu Beginn (" bzw. '). Dafür bietet YAML mehrere Möglichkeiten: Der gesamte Ausdruck kann in Anführungszeichen gesetzt werden (siehe Zeile 29 und 30), oder man leitet einen ggf. mehrzeiligen Ausdruck mit den Symbolen > bzw. | ein. Dies ist in den Zeilen 7 und 15 dargestellt. Mit dem Größer-Als-Zeichen > werden alle internen Zeichenumbrüche gelöscht, wohingegen der Pipe-Befehl | die einzelnen Zeilen erhält. Mit einem optionalen Bindestrich - dahinter wie in Zeile 10 wird auch kein Zeilenumbruch am Ende des Blocks hinzugefügt. So lassen sich beliebig lange Funktionen in JSONata schreiben.

Der andere Teil der Mapping-Datei beschreibt die Abbildungen für einzelne Datenelemente oder besondere Formate. Er wird mit dem Schlüsselwort **ElementMappings** eingeleitet. Hier ist als Beispiel ein Auszug abgebildet:

```
1 ElementMappings:
2   Boolean:
3     true: Y
4     false: N
5
6   Date:
7     yyyy-MM-dd: yyyy\m\d
8     d.MM.yyyy: m/d/yy
9
10  SterilizationMethod:
11    1: 乾熱 Dry Heat
12    2: 環氧乙烷 Ethylene Oxide
13    3: 過氧化氫 Hydrogen Peroxide
14    4: 濕熱或蒸 Moist Heat or Steam
15    5: 過濾滅菌法 Filtration Sterilization
16
17  MriSafetyInformation:
18    SAFE: 0
19    NOT_SAFE: 1
20    SEMI_SAFE: 2
```

Quelltext 3.2: Beispiel einer Mapping-Datei für ElementMappings

Unter dem Stichwort **Boolean** können behördenspezifische Wahrheitswerte gesetzt werden, wie zum Beispiel **Y** und **N** wie in Abbildung 3.2. Außerdem ist es via **Date** möglich, Datums- und Zeitangaben passend für die Excel-Vorlage zu formatieren. Dabei muss als Schlüssel das interne Format in p36 angegeben werden und als Wert das gewünschte Excel-Format. Mit **m/d/yy** zeigt Excel das Datum automatisch in der lokalen Standard-Schreibweise an. Für speziellere Formate ist gegebenenfalls das Escapen der Schrägstriche notwendig, wie in Zeile 7. Genau wie im vorherigen Ansatz können zusätzlich Mappings für einzelne Datenelemente spezifiziert werden. Diese müssen hier lediglich mit dem Namen des Elements eingeleitet werden, nicht dem gesamten **JSONata**-Ausdruck analog zu Abschnitt 3.3.1. Durch die YAML-Notation können einfache Zuweisungen von p36-internen Werten zu den behördenspezifischen Übersetzungen intuitiv angegeben werden. Dabei werden alle primitiven Datentypen sowie Strings unterstützt, auch in gemischter Form.

3.4.1 JSON-Schema

Da die Mapping-Datei eine ganz bestimmte Struktur einhalten muss, wird diese anhand eines Schemas überprüft. YAML ist zwar eine Obermenge von JSON und hat Funktionalitäten, die über JSON hinausgehen, aber für die meisten YAML-Dateien eignet sich dennoch ein JSON-Schema zur Validierung ihrer Form. So auch bei der Transformationsengine. In [Dro22] und [Jac16, S. 21–29] wird aus der praktischen Sicht heraus vertieft, wie ein JSON-Schema aufgebaut ist und welche Möglichkeiten es gibt, die Struktur einer JSON-Datei zu beschreiben, während Pezoa et. al. in [PRS16] das JSON-Schema theoretisch beleuchten und formal definieren.

Das Schema selbst wird ebenfalls im JSON-Format geschrieben, in diesem Fall entsprechend Version v6 der JSON-Schema-Spezifikation. Sie wird zu Beginn mit dem Schlüsselwort **\$schema** angegeben. Das gesamte Schema ist im Quelltext A.1 im Anhang A zu finden. Es besteht aus den beiden JSON-Objekten **SheetMappings** und **ElementMappings**, wobei letzteres auch weggelassen werden darf, falls keinerlei extra Mappings nötig sind. In **SheetMappings** können unter **additionalProperties** die Namen der Arbeitsblätter als Objekt hinzugefügt werden, welche die Eigenschaften **row** und **columns** verpflichtend haben müssen, während **category** und die Liste der **mandatoryColumns** optional sind. Bei der Startzeile muss es sich entsprechend der Excel-Konvention um einen Integer beginnend ab Eins

handeln. Außerdem muss mindestens eine Spalte angegeben werden. Die Spaltennamen werden ebenso durch Excel vorgegeben und müssen aus ein bis zwei Buchstaben bestehen. Dieses Muster muss auch im Array **mandatoryColumns** eingehalten werden. Auf die Beachtung von Groß- und Kleinschreibung kann verzichtet werden, da intern ohnehin alles in Großbuchstaben umgewandelt wird.

Das Objekt **ElementMappings** kann die Eigenschaften **Boolean** (welches wiederum nur das Mapping für **true** und **false** beinhalten darf) und **Date** besitzen, sowie zusätzliche Objekte für Abbildungen einzelner Datenelemente. Hierbei sind Kombinationen aus allen möglichen einfachen Typen von Zahlen über Wahrheitswerte bis hin zu Strings erlaubt.

Durch das Schema können falsche Eingaben, die später zu Fehlern führen, bereits zu Beginn identifiziert werden. Ursprünglich war die Transformationsengine so konzipiert, dass alle durch die Schemavalidierung erkannten, fehlerhaften Pfade des **JsonNodes mapping** eine Warnung im Log erzeugt haben und danach entfernt wurden. Dadurch konnte das Programm weiterlaufen und es entstand eine Excel-Datei, bei der lediglich einzelne Spalten oder Arbeitsblätter nicht gefüllt waren – nämlich diejenigen, die bei der Validierung entfernt wurden. Aufgrund von Feedback im Team wurde die Funktionalität allerdings umgestellt, sodass stattdessen eine Exception geworfen wird. Diese macht direkt auf die Fehler aufmerksam und sie können nicht im Log oder der Excel-Datei übersehen und damit ignoriert werden.

Die Mapping-Datei kann noch beliebig erweitert werden, wobei diese Erweiterungen natürlich auch im Schema und in der Implementierung der Transformationsengine eingebaut werden müssen. Denkbar ist zum Beispiel die Angabe der maximalen Anzahl an Produkten pro Arbeitsmappe bzw. Zeilen pro Arbeitsblatt. Falls diese Zahl überschritten wird, könnte jeweils eine weitere Excel-Datei erstellt werden, um so Probleme bei der Generierung sehr großer Excel-Dateien zu umgehen.

Kapitel 4

Implementierung

Im Folgenden werden zunächst ganz allgemein die bei der Implementierung verwendeten Sprachen und Bibliotheken vorgestellt und der In- und Output der Engine näher beleuchtet, bevor auf die konkrete Umsetzung der beiden generischen Ansätze aus Kapitel 3.3 und 3.4 eingegangen wird.

4.1 Sprachen und Bibliotheken

Bei p36 wird im Backend weitestgehend mit Java gearbeitet, während im Frontend TypeScript zum Einsatz kommt. Auch wenn langfristig immer mehr auf TypeScript umgestellt werden soll, fiel daher die Wahl der Programmiersprache auf Java. Dadurch kann die Transformationsengine nahtlos in die bestehende Struktur eingebunden werden und die Anzahl neuer Abhängigkeiten von externen Bibliotheken wird niedrig gehalten. Die Transformationsengine wurde als Maven-Projekt aufgesetzt und verwendet Java 8 als Version [SDW20, S. 1–16].

4.1.1 JSON

Die Produktdaten liegen im JSON-Format vor. JSON ist mittlerweile eines der weit verbreitetsten und gängigsten Datenaustauschformate. Da es jedoch keine native JSON-Schnittstelle in Java gibt, wurden im Lauf der Zeit eine Vielzahl an open-source Bibliotheken zur Verwendung von JSON entwickelt. Sie dienen im Wesentlichen dazu, JSON zu verarbeiten, zu speichern oder auszutauschen. Ersteres wird durch das Parsen, Abfragen, Transformieren und Generieren von JSON-Dokumenten ermöglicht,

letzteres durch Serialisierung bzw. Deserialisierung, d. h. die Umwandlung eines Java-Objekts in einen String im JSON-Format bzw. die Rückrichtung. Jede API weist ihre Vor- und Nachteile auf und variiert in der Syntax sowie im Umfang. Durch die große Anzahl decken die Schnittstellen mit ihren individuellen Einsatzgebieten in Summe fast alle Anwendungsmöglichkeiten ab und bieten viel Flexibilität. Eine ausführliche Gegenüberstellung von vier exemplarischen Bibliotheken ist in [Vit22] zu finden, während [HDB21] 20 APIs bzgl. des Ein- und Ausgabeverhaltens vergleicht. In den Benchmarks [VK18] und [Ren21] wird hingegen die Performanz gemessen. Eine Auswahl an Bibliotheken gestaffelt nach Popularität¹ ist hier aufgelistet:

- **Jackson** von Tatu Saloranta (2008)
- **Gson** von Google (2008)
- **org.json** von Douglas Crockford / Sean Leary (2010)
- **json-simple** von Fang Yidong / Davin Loegering (2008)
- **JSON-P** und **JSON-B** aus der Jakarta EE² (2019)

Jackson [Fri19, S. 323–403] schneidet in den oben genannten Benchmarks leistungstechnisch besonders gut ab, während **Gson** [Fri19, S. 243–298] im fehlerfreien Parsen beeindruckt. Beide Bibliotheken werden mit Abstand am häufigsten verwendet und bestechen durch ihre vielseitigen Funktionalitäten, insbesondere im Bereich der Serialisierung. Im Gegensatz dazu sind **org.json** und **json-simple**, wie der Name schon andeutet, leichtgewichtig und unkompliziert in der Verwendung, beschränken sich allerdings nur auf das Lesen und Schreiben. **JSON-P** (Processing) und **JSON-B** (Binding) sind als Standard in Jakarta eingebunden und können u. a. mit **JsonPointer** arbeiten, vgl. [Abt22, S. 21–34].

In diesem Projekt wird JSON mittels **Jackson** eingebunden. Die Bibliothek ist wie bereits erwähnt besonders populär, schnell und umfangreich und sie zeichnet sich durch regelmäßige Veröffentlichung neuer Releases aus, siehe [Fas22]. Außerdem wird sie bereits innerhalb der UDI Platform verwendet, wodurch Einheitlichkeit erhalten werden kann. Ein für die

¹ gemessen an der Häufigkeit der Nutzung in anderen Artefakten aller verzeichneten Maven-Repositories (auf <https://mvnrepository.com>)

² Jakarta EE, ehemals Java Platform, Enterprise Edition: Spezifikationen für Unternehmens- und Webanwendungen, die auf der Java Standard Edition aufsetzen

Transformationsengine relevanter Nachteil besteht darin, dass die Navigation entlang bestimmter Pfade nur umständlich umgesetzt werden kann. Dafür unterstützt **Jackson** allerdings das Prinzip der Datenbindung, was in Abschnitt 4.4 genutzt wird.

Natürlich muss auch die Bibliothek für die gewählte JSON-Abfragesprache eingebunden werden, also in diesem Fall **JMESPath** bzw. **JSONata4Java**. Beide Schnittstellen unterstützen die Verwendung unter **Jackson** (als auch **Gson**), sodass es zu keinen Kompatibilitätsproblemen kommt.

In Ansatz 3.4 wird zusätzlich noch ein JSON-Schema-Validator für die YAML-Datei eingebunden. Hier gibt es ebenfalls viele verschiedene Bibliotheken zur Auswahl, die sich alle nicht wesentlich unterscheiden, außer in den Randbedingungen, die für Transformationsengine vernachlässigbar sind.

4.1.2 Excel

Für die Bearbeitung von Excel-Dateien in Java ist in der Regel die Bibliothek **Apache POI** die erste Wahl. Die wenigen Alternativen bieten zum Teil eine höhere Geschwindigkeit und geringeren Speicherverbrauch, dafür muss der Benutzer aber an anderen Stellen Einschränkungen hinnehmen vgl. [Raj21]. Um die nahtlose Eingliederung in das bestehende System zu gewährleisten, wurde sich direkt für **Apache POI** entschieden. Hiermit können nicht nur Excel-Dokumente, sondern allgemein Microsoft Office-Dateien dynamisch gelesen und geschrieben werden. Die Abkürzung **POI** stand anfangs ironischerweise für „Poor Obfuscation Implementation“, wurde aber später aus Marketinggründen fallengelassen. Heute ist **Apache POI** eine extrem mächtige Bibliothek, die allen Anforderungen zur Erstellung der Excel-Dateien innerhalb dieses Projektes gerecht wird. Einziges Manko ist die Bearbeitung sehr großer Arbeitsmappen mit über 50.000 Zeilen, wobei vergleichbare Bibliotheken zwangsläufig auch ab einem gewissen Punkt an ihre Grenzen stoßen. Damit der Heap-Speicherplatz nicht überläuft, gibt es eine Streaming-Erweiterung, bei der immer nur auf einen gleitenden Bereich von Zeilen Zugriff besteht. Dies bringt natürlich Nachteile mit sich, insbesondere bei der Formelauswertung, außerdem sind einige Funktionalitäten noch nicht implementiert, siehe [OB22]. Für die Transformationengine ist das Schreiben der Excel-Datei via Streaming noch nicht nötig, aber für die UDI Plattform allgemein arbeitet p36 gerade an einer Umstellung für den Massen-Upload von bis zu 10.000 Produkten gleichzeitig.

4.1.3 Testen

Gerade im medizinischen Bereich sind Tests zur Qualitätssicherung von besonderer Bedeutung. Unter dem Stichwort „GxP“ müssen nicht nur die Medizinprodukte selbst Richtlinien zur Sicherheit, Wirksamkeit und ihrer Verwendung erfüllen, sondern indirekt auch die zugrundeliegende Lieferkette, vgl. [Sai22]. Darunter fällt insbesondere p36 als IT-Zulieferer. Als anerkannter Standardleitfaden für computergestützte Systeme dient der GAMP 5 Guide [ISPE22].

Zur Validierung der Software kommen dabei verschiedene Testarten zum Einsatz: von Modultests über Integrationstests und Funktions-/Systemtests bis hin zu Akzeptanztests. Auf unterster Ebene wurden auch für dieses Projekt Modultests geschrieben, um die einzelnen Bestandteile der Transformationsengine lokal zu testen. Umgesetzt wurde dies mit **JUnit 4**, siehe dazu [Wes06] und [Kle19, S. 47–79]. Neben verschiedenen Testfällen für die jeweiligen Klassen mit ihren Methoden, wurde auch getestet wie viele Produkte in einem Zug in eine Excel-Vorlage geschrieben werden können, bevor es zum Heap-Overflow kommt. Darauf wird in Abschnitt 5.3 noch näher eingegangen.

Das Logging läuft über die Fassade **SLF4J** (Simple Logging Facade for Java, vgl. [QOS22]), in welcher wiederum das Framework **Log4j 2** angewendet wird. Zur Vereinfachung wird dabei das **Project Lombok** [ZS22] verwendet, welches mit einer Annotation automatisch das gewünschte Log-Feld für eine Klasse erzeugt. Des weiteren kommen **Lombok**-Annotationen an verschiedenen Stellen als Shortcuts für Getter, Setter und Null-Checks zum Einsatz.

Damit sind bereits alle Abhängigkeiten erwähnt und die Maven-Konfigurationsdatei **pom.xml** bleibt relativ überschaubar. Insbesondere sind viele der Bibliotheken ohnehin schon anderweitig in der UDI Platform in Verwendung – dabei wurde darauf geachtet, dass die Versionen möglichst übereinstimmen. Die Bibliotheken sind in Tabelle 4.1 zur Übersicht aufgelistet.

4.2 Ein- und Ausgabe

Man kann sich die Transformationsengine als in sich geschlossene Blackbox vorstellen, die in der UDI Platform zum Einsatz kommt.

Bibliothek	groupId	artifactId	version
Apache POI	org.apache.poi	poi & poi-ooxml	4.1.2
Jackson	com.fasterxml.jackson.core	jackson-databind	2.13.0
JMESPath	io.burt	jmespath-jackson	0.5.1
JSONata	com.ibm.jsonata4java	JSONata4Java	1.7.8
JSON Schema Validator	com.networknt	json-schema-validator	1.0.70
JUnit	junit	junit	4.12
Logging	org.apache.logging.log4j	log4j-core log4j-slf4j-impl	2.7
Project Lombok	org.projectlombok	lombok	1.18.24

Tabelle 4.1: Überblick über die verwendeten Bibliotheken

Die Eingabeparameter sind dabei die Produktdaten als String im JSON-Format sowie die Excel-Vorlage. Bei Ansatz 3.4 ist zusätzlich noch die bereits erwähnte YAML-Datei nötig, welche das Mapping zwischen den Produktdaten und der Excel-Datei definiert. Die beiden Dokumente werden als `InputStream` übergeben. Der Konstruktor mit den beschriebenen Eingabeparametern der Klasse `TransformationEngine` ist im folgenden Code ab Zeile 1 abgebildet.

Als Ausgabe wird ein `OutputStream` erzeugt, der die mit den im Mapping spezifizierten Produktdaten ausgefüllte Excel-Datei enthält. Auch diese Methode ist im folgenden Quelltext 4.1 ab Zeile 18 dargestellt.

```

1  /**
2   * constructor for the transformation engine
3   *
4   * Reads the Excel template of a specific agency via InputStream.
5   * Reads the YAML file via InputStream that contains the mapping between
6   * the Excel sheet and the device data given by JSONata expressions.
7   * Reads the device data via String that shall be filled into the
8   * template. It needs to be in a JSON format of an array containing a
9   * JSON object for each device with all of its data.
10  *
11  * @param inTemplate InputStream of the Excel template given by agency
12  * @param inMapping InputStream of the YAML file describing the mapping
13  * between Excel template and device data
14  * @param deviceData String in JSON format of an array of all device
15  * objects with their data
16  * @throws TransformationException custom exception for any errors that
17  * specifically occur during the transformation
18  */
19  public TransformationEngine(@NonNull InputStream inTemplate, @NonNull
20  InputStream inMapping, @NonNull String deviceData) throws

```

```

    TransformationException {
14     setTemplate(inTemplate);
15     setMapping(inMapping);
16     setDevices(deviceData);
17 }

18 /**
19  * Populates the Excel template with the given device data based on the
    mapping information in the YAML file.
20  * Writes the Excel workbook to the OutputStream, for example a file.
21  *
22  * @param outExcel the OutputStream for the filled Excel template
23  * @return OutputStream with the filled Excel template
24  * @throws TransformationException custom exception for any errors that
    specifically occur during the transformation
25  */
26 public OutputStream fillTemplateWithDevices(@NonNull OutputStream
    outExcel) throws TransformationException {
27     this.filledTemplate = fillTemplate(template, mapping, devices);
28     return writeExcel(filledTemplate, outExcel);
29 }

```

Quelltext 4.1: Auszüge aus der Klasse TransformationEngine

4.3 Ansatz mit Excel-Datei

Die Implementierung von Ansatz 3.3 mit der bearbeiteten Excel-Datei ist im Sequenzdiagramm in Abbildung 4.1 grob dargestellt.

Im Konstruktor werden die Produktdaten **deviceData** eingelesen und als **JsonNode** abgespeichert. Außerdem wird die Excel-Arbeitsmappe aus dem **InputStream inTemplate** mit der Bibliothek **Apache POI** geladen.

Bevor die Arbeitsblätter mit Daten gefüllt werden können, wird zunächst das Settings-Arbeitsblatt ausgelesen und nach möglichen Mappings durchsucht. Diese werden in Hashtabellen gespeichert und im Anschluss kann das Settings-Sheet gelöscht werden. Dies wurde bereits in Abschnitt 3.3.1 näher beschrieben und ist hier nur kurz skizziert.

Danach wird jedes Arbeitsblatt Zelle für Zelle nach Eingaben durchforstet, die mit < beginnen und > enden, um alle Spalten zu finden, welche mit Produktdaten gefüllt werden sollen. Der Code dafür ist im Quelltext 4.2 abgebildet.

```

1 public HashMap<String, ColumnInfo> getColumns(Sheet templateSheet,
    JmesPath<JsonNode> jmesPath) throws TransformationException {
2     HashMap<String, ColumnInfo> columns = new HashMap<String,ColumnInfo>();

```

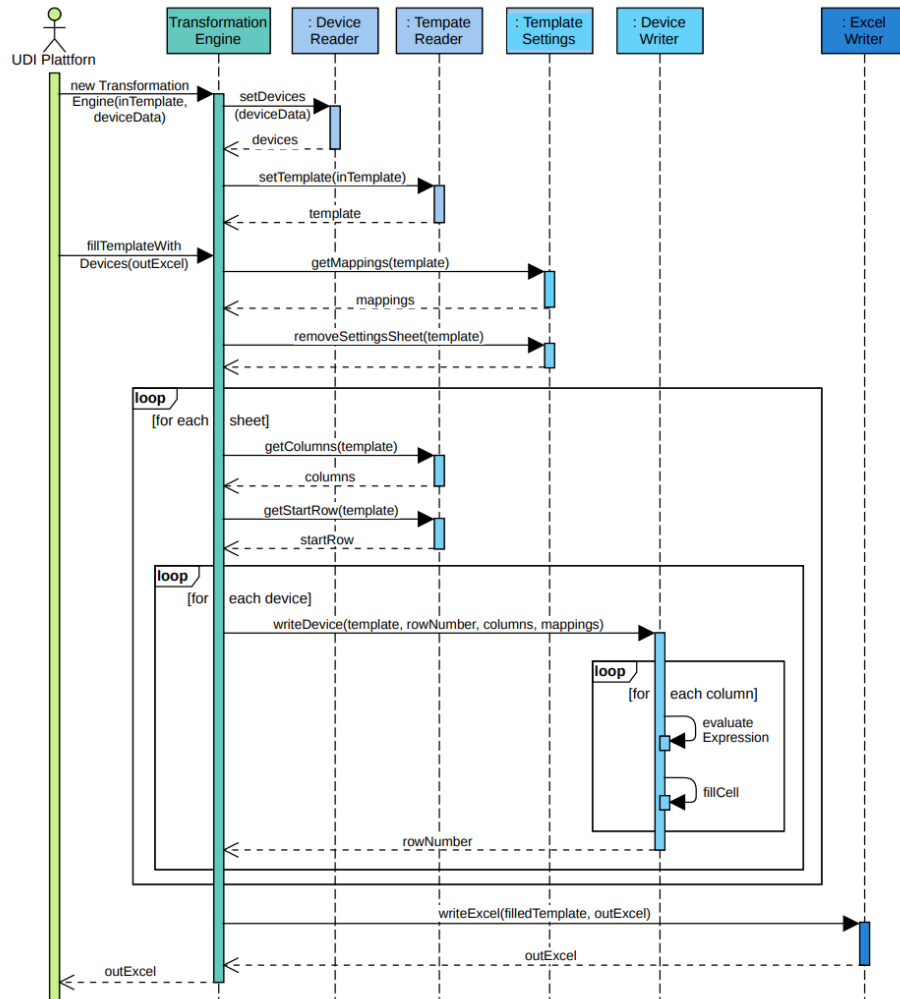


Abbildung 4.1: Sequenzdiagramm

```

3  this.startRow = -1;
4
5  for (Row row : templateSheet) {
6      for (Cell cell : row) {
7          if (cell.getCellType() == CellType.STRING) {
8              String cellValue = cell.getRichStringCellValue().getString();
9              if (cellValue.startsWith("<") && cellValue.endsWith(">")) {
10                 //entry found
11                 cellValue = cellValue.substring(1, cellValue.length() - 1);
12
13                 //check if element is mandatory or cell can be left empty
14                 boolean mandatory = false;
15                 if (cellValue.statssWith("!")) {
16                     mandatory = true;
17                     cellValue = cellValue.substring(1, cellValue.length());
18                 }
19             }
20         }
21     }
22 }
  
```

```
19
20     //parse JMESPath expression
21     try {
22         Expression<JsonNode> expression = jmesPath.compile(cellValue)
23         ;
24     } catch (ParseException e) {
25         String colLetter = CellReference.convertNumToColString(cell.
26             getColumnIndex());
27         String message = String.format("JMESPath expression of sheet
28             \"%s\", column %s cannot be parsed.\n%s",
29             templateSheet.getSheetName(), colLetter, e.getMessage()
30             );
31         log.error(message);
32         throw new TransformationException(message, e);
33     }
34
35     //save all infos for this column
36     ColumnInfo column = new ColumnInfo(cell.getColumnIndex(),
37         mandatory, expression);
38     columns.put(cellValue, column);
39
40     checkIfRowsAlign(templateSheet, row.getRowNum());
41 }
```

Quelltext 4.2: Einlesen der Spalteninformationen in `TemplateReader`

Sobald eine Zelle gefunden wurde, die ein Spalteneintrag enthält (siehe Zeile 9), werden die entsprechenden Informationen verarbeitet und abgespeichert. Wie man in Zeile 32 im folgenden Code sieht, wird nicht nur die Spaltennummer gespeichert, sondern auch die Angabe, ob die Spalte verpflichtend ist. Ebenso der kompilierte `JMESPath`-Ausdruck, mit dem später die geforderten Werte der einzelnen Produkte ermittelt werden können. Falls der Ausdruck fehlerhaft ist und nicht geparkt werden kann, wird eine `Exception` ausgelöst, gefangen und in die eigens definierte `TransformationException` umgewandelt.

Zum Schluss in Zeile 35 wird außerdem in der Methode `checkIfRowsAlign` die Excel-Zeilenummer abgeglichen (siehe Quelltext 4.3). Diese beschreibt die Start-Zeile, in die der Inhalt des ersten Produkts eingetragen wird und womit die `JMESPath`-Ausdrücke aus der Vorlage überschrieben werden. Unterscheiden sich die Start-Zeilen innerhalb eines Arbeitsblattes, wird ebenfalls eine `TransformationException` geworfen, weil die Einträge eines

Datensatzes nicht alle in gerader Fluchtlinie (d. h. in einer Zeile) angeordnet sind.

```

1 private checkIfRowsAlign(Sheet templateSheet, int rowNum) throws
   TransformationException;
2 switch (this.startRow) {
3     case -1: //first entry - initialize
4         this.startRow = rowNum;
5         break;
6     case rowNum: //all good
7         break;
8     default: //no alignment - error
9         String message = String.format("The starting rows for the different
   device data fields do not align in sheet \"%s\".",
10             templateSheet.getSheetName());
11         log.error(message);
12         throw new TransformationException(message);
13         break;
14 }
15 }

```

Quelltext 4.3: Hilfsfunktion beim Einlesen der Spalteninformationen

Obwohl über Blätter, Spalten und Zeilen iteriert wird, liegt der Aufwand im Bereich $\mathcal{O}(n)$, wobei n die Anzahl der Spalten beschreibt bzw. umformuliert die Anzahl der von der Behörde geforderten Produktinformationen. Diese Anzahl wird nicht sonderlich groß, sondern bewegt sich im zwei- bis unteren dreistelligen Bereich¹. Die Anzahl der Arbeitsblätter und nicht-leeren Zeilen in einer unausgefüllten Vorlage ist daher vernachlässigbar klein und entsprechend verbraucht die Suche nach den Spalten, d. h. die Ausführung von `getColumns`, wenig Zeit.

Nachdem alle Spalten- und Mapping-Informationen vorliegen und ausgewertet sind, werden die einzelnen Produkte in die Vorlage geschrieben. Dies geschieht in der Klasse `DeviceWriter`. Hierbei wird nochmal über die Spalten iteriert. Die JMESPath-Ausdrücke werden auf das explizite Produkt angewendet und der so erhaltene Wert wird zunächst anhand der vorliegenden Mappings in die behördenspezifische Format- oder Formulierungsvorgabe umgewandelt. Der ggf. „übersetzte“ Wert wird dann entsprechend seines Types in die Excel-Zelle geschrieben. Dabei werden Wahrheitswerte, Zahlen und Text sowie Datumsangaben unterstützt. Für Letztere ist das Format allerdings aktuell fest vorgegeben – hier könnte man

¹ Datensatz von Behörden mit UDI-System in [Len22]: näherungsweise Anzahl an geforderten Datenattributen – Minimum: 13 (Singapur); Maximum: 130 (EU)

durch zusätzliche Angaben im Settings-Arbeitsblatt behördenspezifische Wünsche inkludieren.

Wie bereits erwähnt, ist in der Regel pro Spalte und Produkt nur genau ein Wert gefordert, aber es kann auch vorkommen, dass zu einem Produkt mehrere Informationen bzgl. einer Kategorie vorliegen, die in mehrere Excel-Zeilen geschrieben werden müssen (siehe Abbildung 3.3). Der **JMESPath**-Ausdruck liefert in diesem Fall ein Array, deren einzelne Einträge jeweils eine eigene Excel-Zeile erhalten. Hierbei ist wichtig, dass **null**-Werte mitgegeben werden, falls ein Datenelement in einem Array nicht vorhanden sein sollte, sodass für ein Produkt die Länge des Arrays pro Kategorie in jeder Spalte gleich ist. Dadurch wird gewährleistet, dass jeweils alle Werte einer Zeile zueinander gehören. Standardmäßig werden Nullwerte bei einer **JMESPath**-Projektion allerdings herausgefiltert und nicht berücksichtigt. Um dies zu umgehen, kann man sich Multiselect-Listen bedienen, die mit dem Pipe-Operator dann im Nachgang wieder glättet werden. Als Beispiel kann man das komplexe Element **TradeNames** betrachten.

```
1 {  
2   "TradeNames": [  
3     {  
4       "TradeName": "englishName",  
5       "TradeNameLanguage": "EN"  
6     },  
7     {  
8       "TradeName": "noLanguageGiven"  
9     },  
10    {  
11      "TradeName": "deutscherName",  
12      "TradeNameLanguage": "DE"  
13    }  
14  ],  
15  ...  
16 }
```

Quelltext 4.4: Beispiel eines komplexen Datenelements

Um für jedes Objekt eine neue Zeile zu erzeugen, in der sowohl **TradeName** als auch **TradeNameLanguage** in einer eigenen Spalte angegeben werden, bieten sich die folgenden **JMESPath**-Ausdrücke an:

- `<TradeNames[].TradeName | []>`
- `<TradeNames[].TradeNameLanguage | []>`

Durch die zusätzlichen eckigen Klammern um die Datenelemente **TradeName**

und **TradeNameLanguage** wird anstatt einer normalen Projektion eine Multiselect-Liste erzeugt, die in diesem Fall aus ein-elementigen Arrays besteht, weil nur ein Datenelement gesucht ist. Jedes Auswertungsteilergebnis ist dabei enthalten, auch wenn es **null** beträgt – in diesem Beispiel die nicht vorhandene Angabe der Sprache im zweiten Objekt (Zeile 8). Mit `[]` wird das erzeugte Array von Arrays wieder geglättet. Alternativ kann mit der eingebauten Funktion **map** gearbeitet werden. Der Ausdruck `map(&TradeNameLanguage, TradeNames[])` führt zu demselben Ergebnis, da auch hier eine neue Liste erzeugt wird, die die gleiche Länge wie die ursprüngliche Liste hat. Jedes Objekt in **TradeNames** wird auf die **TradeNameLanguage** abgebildet, falls vorhanden – ansonsten auf **null**. Im Excel-Template wird der Nullwert dann als leere Zelle interpretiert.

Liegen fehlerhafte Produktdaten vor oder können **JMESPath**-Ausdrücke nicht interpretiert werden, bricht das Programm mit einer Fehlermeldung ab und die Informationen zum Produkt bzw. zur Spalte im Arbeitsblatt werden geloggt. Ursprünglich war die Logik so, dass Produkte mit Datenfehlern (z. B. ein fehlender, verpflichtender Wert) aus der Excel-Vorlage gelöscht und geloggt, alle restlichen Produkte aber eingetragen wurden. Dadurch konnte immer eine valide Excel-Datei erzeugt werden, die möglicherweise aber nur zu einem Bruchteil ausgefüllt war. Die Rücksprache im Team ergab, dass das erfolgreiche Durchlaufen trotz einzelner Fehler dazu führt, dass diese übersehen werden. Anstatt lediglich Einträge im Log zu hinterlassen, wird nun eine Exception geworfen.

Die Implementation des **DeviceWriters**, um die Daten eines Produktes in die Arbeitsmappe zu schreiben, folgt bei beiden Ansätzen identischen Prinzipien. Ansatz 4.4 ist etwas umfangreicher und es wurden einige Details optimiert, daher wird zur Veranschaulichung an dieser Stelle auf das später folgende Aktivitätsdiagramm 4.3 verwiesen.

Konnten alle Produktdaten erfolgreich in die Arbeitsmappe eingetragen werden, wird das ausgefüllte Excel-Template in der Klasse **ExcelWriter** in einen **OutputStream** geschrieben. Dabei kann es sich beispielsweise um eine Datei handeln, die dann erzeugt wird.

4.4 Ansatz mit Mapping-Datei

Die Implementierung für den Ansatz mit der extra YAML-Datei für die Projektion von Produktdaten in die Excel-Datei ist in Abbildung 4.2 in einem Sequenzdiagramm dargestellt. Er ist ähnlich aufgebaut wie Ansatz 4.3. Im Konstruktor werden wie bisher die Excel-Vorlage geladen und die Produktdaten deserialisiert. Zusätzlich wird nun die Mapping-Datei eingelesen und direkt validiert (siehe Abschnitt 3.4.1). Um die Produktdaten in die Arbeitsmappe zu schreiben, wird ebenfalls wieder über die Arbeitsblätter, die Produkte und die einzelnen Spalten iteriert. Dies geschieht in der Klasse `DeviceWriter`.

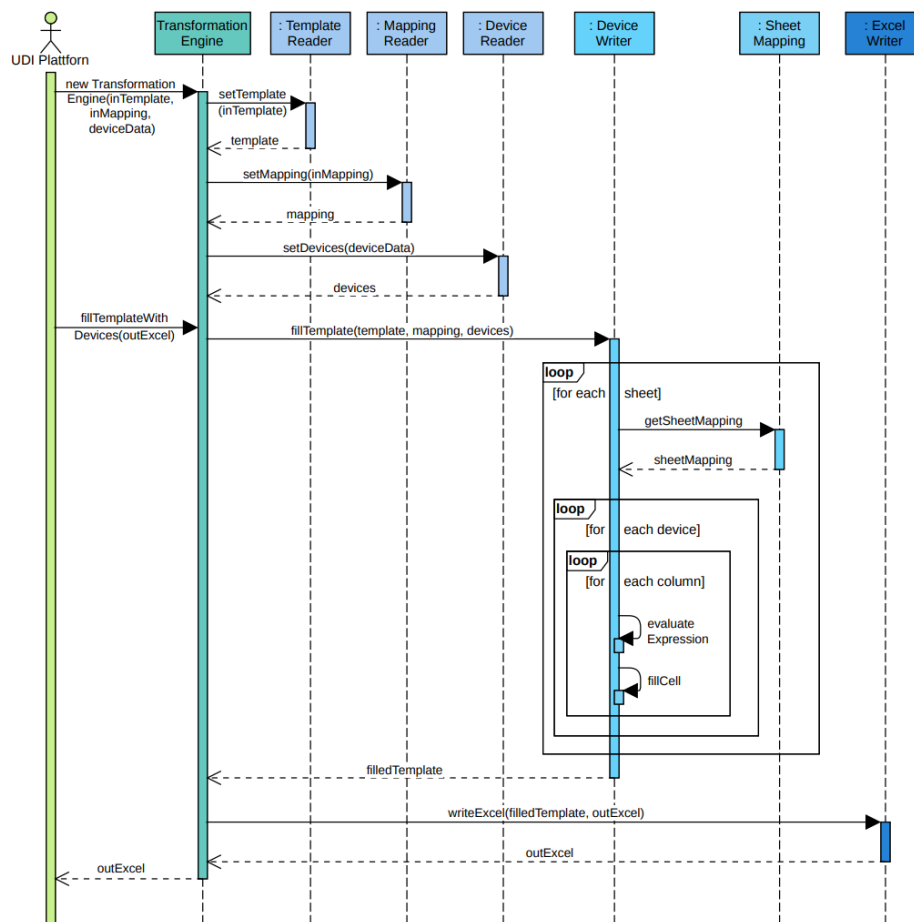


Abbildung 4.2: Sequenzdiagramm

Eine interne Verbesserung zur Implementierung des vorherigen Ansatzes besteht darin, dass die Produktdaten, die auf behördenspezifische Werte abgebildet werden sollen, direkt als erstes vor den verschiedenen Schlei-

fendurchlaufen transformiert werden. Das gilt auch für die Datums- und Zeitformate. Dabei wird für jedes gewünschte Excel-Format ein **CellStyle** angelegt, in einer Hashtabelle gespeichert und später auf diejenigen Zellen angewendet, die ein Datum im angegebenen Format enthalten (sollen). Für jeden in der Mapping-Datei angegebenen Arbeitsblatt-Titel werden die folgenden vier Schritte durchgeführt:

1. Das zugehörige Arbeitsblatt in der Excel-Vorlage suchen: Kann kein Arbeitsblatt mit dem angegebenen Namen gefunden werden, wird ein Fehler geworfen und die Transformationsengine bricht ab.
2. Die Angaben zum Arbeitsblatt aus der Mapping-Datei per Datenbindung in ein Objekt der Klasse **SheetMapping** übertragen: Hierauf wird im folgenden Abschnitt 4.4.1 noch näher eingegangen.
3. Die Variablen aus **SheetMapping** abrufen, unter anderem die Startzeile, die verschiedenen Spaltennamen mit deren **JSONata**-Ausdrücken sowie eine Liste aller verpflichtenden Spalten: Wurden dabei syntaktisch falsche Werte übergeben, wird ebenfalls ein Fehler geworden.
4. Jedes Produkt in eine oder ggf. mehrere Zeilen des Excel-Arbeitsblatts schreiben: Dabei wird wie in Kapitel 3.3 vorgegangen, indem spaltenweise der **JSONata**-Ausdruck ermittelt und dann im richtigen Format in die passende Zelle geschrieben wird. Dies wird in Abschnitt 4.4.2 erläutert.

4.4.1 Datenbindung in SheetMapping

Alle JSON-Daten zu einem Arbeitsblatt werden dank der Bibliothek **Jackson** an eine Instanz der Klasse **SheetMapping** gebunden, wie der Codeausschnitt 4.5 zeigt. Man spricht hier von Daten-Deserialisierung, da JSON-Knoten in Java-Objekte umgewandelt werden. Die Datenbindung an ein Objekt hat gegenüber der Speicherung im Baummodell den Vorteil, dass Abfragen deutlich bequemer durchführbar sind und einfacher mit den Daten gearbeitet werden kann. Anstatt wiederholt die Pfade des Baumes abzulaufen, können Parameter direkt referenziert werden.

```
1 ObjectMapper jsonReader = new ObjectMapper();  
2 jsonReader  
3     .enable(DeserializationFeature.ACCEPT_EMPTY_STRING_AS_NULL_OBJECT);  
  
4 private SheetMapping getSheetMapping(ObjectMapper jsonReader, String  
    sheetName) throws TransformationException {
```

```
5 InjectableValues injectableValues =
6     new InjectableValues.Std().addValue(String.class, sheetName);
7 try {
8     return jsonReader
9         .reader(injectableValues)
10        .treeToValue(sheetMappings.get(sheetName), SheetMapping.class);
11 } catch (JsonProcessingException | IllegalArgumentException e) {
12     String message = String.format("Sheet called \"%s\" could not be
13         filled, because the mapping cannot be parsed properly.",
14         sheetName);
15     log.error(message);
16     throw new TransformationException(message, e);
17 }
```

Quelltext 4.5: Datenbindung für Arbeitsblätter der Mapping-Datei

Umgesetzt wird die Anbindung durch Annotationen in der Klasse **SheetMapping**. Mit **@JacksonInject** wird der Name des Arbeitsblatts zur Klasse hinzugefügt, um diese Info den Fehlermeldungen mitgeben zu können. Dies wird in Zeile 5 im Code 4.5 vorbereitet. Mit **@JsonProperty** werden alle weiteren Attribute gesetzt, die im Anschluss aufgelistet sind.

- die Kategorie **category** als **String**:
Falls kein komplexes Datenelement angegeben wurde, auf welches sich das Arbeitsblatt bezieht, beträgt der Wert **null**.
- die Startzeile **row** als **Integer**:
Die Zeile muss größer als 0 sein, ansonsten bricht die Transformationsengine mit einer Fehlermeldung ab. Außerdem wird sie um eins reduziert, da die Bibliothek **Apache POI** mit 0-basierten Zeilennummern arbeitet, während die Zeilen in Excel bei 1 beginnen.
- die Spalten **columns** als **HashMap <String, String>** mit dem Spaltennamen als Schlüssel und dem **JSONata**-Ausdruck als Wert:
In der zugehörigen Getter-Methode werden die Spaltennamen zunächst auf syntaktische Korrektheit überprüft (1-2 Buchstaben von A bis Z) und dann in die entsprechende 0-basierte Spaltenzahl umgewandelt, mit welcher **Apache POI** rechnet. Die Funktion ist im Code 4.6 in Zeile 14– 28 dargestellt. Außerdem werden die **JSONata**-Ausdrücke mittels der Bibliothek **JSONata4Java** geparsed und in **Expressions** umgewandelt. So wird also der Typ **Map <Integer, Expressions>** zurückgegeben. Tritt ein Fehler im Spaltennamen oder während des **JSONata** Parsens auf, wird eine **TransformationException** geworfen.

- die verpflichtenden Spalten `mandatoryColumns` als `HashSet <String>`: Bevor die Menge der Spalten herausgegeben wird, werden auch hier die Spaltennamen in Zahlen, beginnend bei 0, umgerechnet und gegebenenfalls ein Fehler geworfen.

Im folgenden Codeausschnitt 4.6 ist exemplarisch die Implementierung für die obligatorischen Spalten dargestellt. Alle gewöhnlichen Setter- und Getter-Methoden ohne besondere Funktionalitäten werden einfachheitshalber mit Hilfe von Lombok-Annotationen deklariert, wie z. B. in Zeile 2.

```

1  @JsonProperty("mandatoryColumns")
2  @Setter
3  private Set<String> mandatoryColumns = new HashSet<String>();
4
5  //Getter
6  public Set<Integer> getParsedMandatoryColumns() throws
    TransformationException {
7      Set<Integer> mandatoryColumnsParsed = new HashSet<Integer>();
8      mandatoryColumns.forEach((columnLetter) -> {
9          mandatoryColumnsParsed.add(parseCol(columnLetter));
10     });
11     return mandatoryColumnsParsed;
12 }
13
14 private Integer parseCol(String columnLetter) throws
    TransformationException {
15     columnLetter = columnLetter.toUpperCase();
16     if (!columnLetter.matches("[A-Z]{1,2}")) {
17         String message = String.format(
18             "Yaml file describing agency template is not filled correctly: in
              sheet \"%s\", column %s is not a valid excel column name.",
19             sheetName, columnLetter);
20         log.error(message);
21         throw new TransformationException(message);
22     }
23     int columnNumber = 0;
24     for (int i = 0; i < columnLetter.length(); i++) {
25         columnNumber = columnNumber * 26 + columnLetter.charAt(i) - 'A' + 1;
26     }
27     return columnNumber - 1;
28 }

```

Quelltext 4.6: Setter / Getter für verbindliche Spalten in `SheetMapping`

Die `TransformationException` ist eine benutzerdefinierte Ausnahme (siehe dazu [SDW20, S. 32f] und [GHK14, S. 309–338]), welche möglichst alle checked Exceptions zusammenfasst, die bei der Transformation entstehen können. Ausgenommen sind ungeprüfte Laufzeit-Fehler. `NullPointerException` werden mittels Lombok erzeugt, falls ein Eingabeparameter `null`

ist. Betrachtet man die Transformationsengine im Zusammenspiel mit der restlichen Plattform, wird so direkt ersichtlich, dass ein Fehler innerhalb der Transformationsengine aufgetreten ist und im Log kann die genaue Ursache nachgelesen werden.

4.4.2 Produktdaten schreiben

Die spaltenweise Auswertung des JSONata-Ausdrucks und das Füllen der Zelle in der Excel-Vorlage funktioniert ganz analog zu Abschnitt 4.3. Es ergibt sich das Aktivitätsdiagramm aus Abbildung 4.3 für die Methode `writeOneDevice` der Klasse `DeviceWriter`, um – wie der Name schon sagt – die Daten zu einem Produkt in ein Arbeitsblatt zu schreiben.

Um den Prozess im Aktivitätsdiagramm nachvollziehen zu können, wird zunächst der Unterschied zwischen einfachen und komplexen Datenelementen näher erläutert. Einfache Datenelemente stehen in einer 1:1-Beziehung zum Produkt, das heißt es liegt nur eine einzige Ausprägung vor, die dem Datenelement zugeordnet ist. Dies kann als String, Zahl oder Wahrheitswert sein. Ein Spezialfall bildet der in Abschnitt 3.3 erwähnte `ProduktionIdentifizier`, welcher in einem Array von Strings unterschiedliche einfache Elemente zusammenfasst. Er wird jedoch bereits über den JSONata-Ausdruck abgefangen und in einen Wahrheitswert umgewandelt, sodass hierfür keine zusätzliche Logik implementiert werden muss. Die komplexen Datenelemente bestehen hingegen aus einem Array von Objekten, welche wiederum einfache Datenelemente enthalten. Ein Beispiel wurde bereits in Quelltext 4.4 gegeben. Sie stehen also in einer 1:n-Beziehung, d. h. ein Produkt kann mehrere unterschiedliche Ausprägungen / Einträge für ein komplexes Datenelement haben. In Excel können alle einfachen Elemente in einem Arbeitsblatt in einer Zeile aneinander gereiht werden, da ein Element genau einer Zelle entspricht. Die komplexen Elemente werden in der Regel in zusätzlichen, „relationalen“ Arbeitsblättern dargestellt, in denen jedes Objekt des Arrays des komplexen Elements einer Zeile entspricht. Ein Produkt füllt in diesem Fall also n Zeilen, wobei $n \geq 0$ gilt. Falls für ein Produkt keine Daten bzgl. eines komplexen Elements vorliegen (d. h. $n = 0$), wird es im entsprechenden relationalen Arbeitsblatt auch nicht aufgelistet, ansonsten wird über alle n Objekte im Array iteriert.

Die Methode `writeOneDevice` beginnt mit der Initialisierung verschiedener Flags und Counter, die im weiteren Verlauf Verwendung finden.

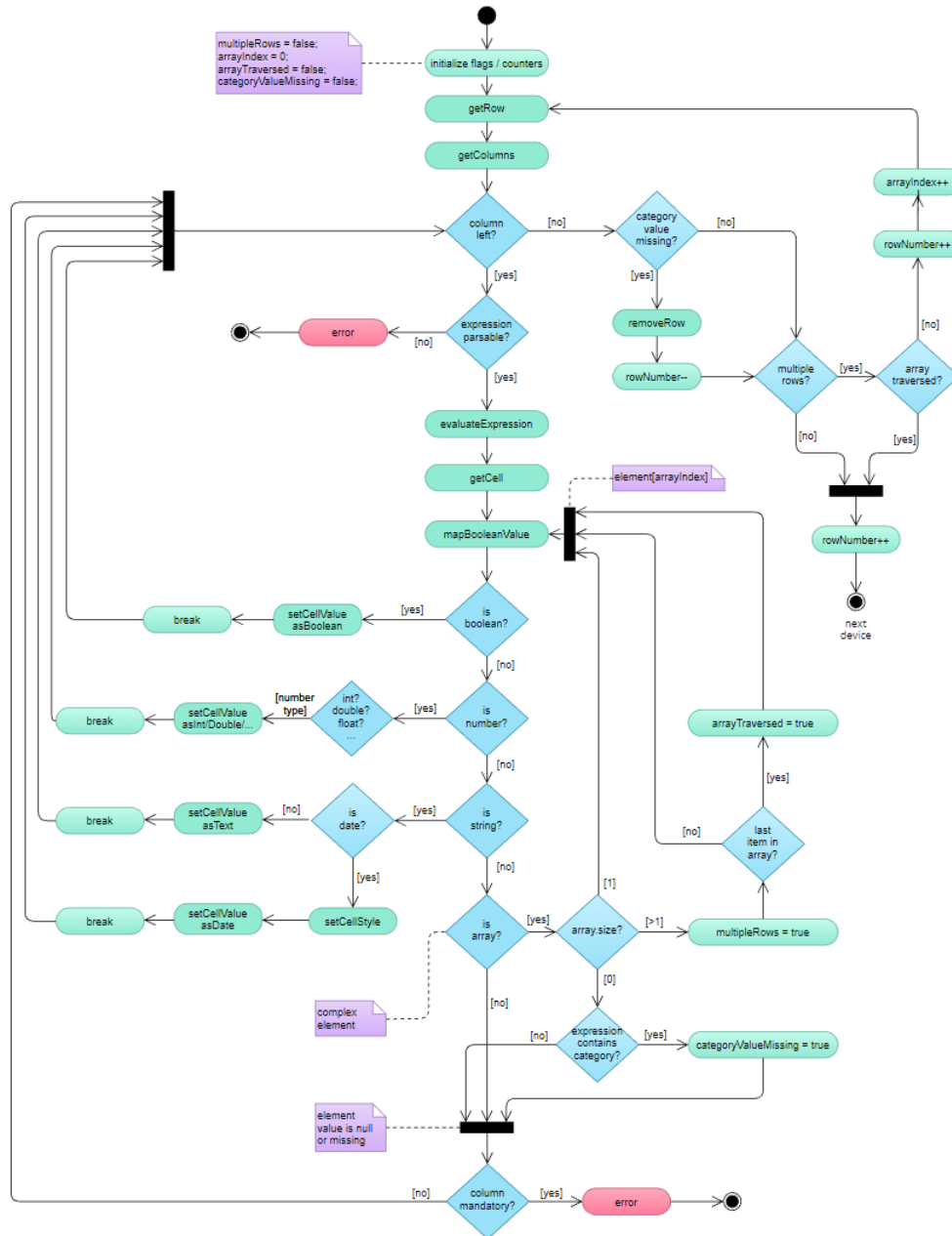


Abbildung 4.3: Aktivitätsdiagramm für writeOneDevice

Danach wird zunächst die aktuelle Zeile anhand der **rowNumber** im aktuellen Arbeitsblatt zur Bearbeitung ausgewählt. Die Liste der Spalten für das jeweilige Arbeitsblatt entsprechend der Mapping-Datei wird geladen und im Folgenden durchlaufen. Dabei wird spaltenweise der **JSONata**-Ausdruck evaluiert (falls dies möglich ist), sowie die aktuelle Zelle ausgewählt und gefüllt. Beim Füllen werden zunächst mögliche Wahrheitswerte in behör-

denspezifische Werte umgewandelt und dann wird per Switch-Statement über den Datentyp die Zelle typgerecht mit dem Ergebnis der **JSONata**-Auswertung beschrieben. Es werden primitive Datentypen, wie Booleans und die unterschiedlichen Zahlenwerte, sowie Strings unterstützt. Liegen bei Strings spezielle Datums- oder Zeitangaben vor, muss die Excel-Zelle zusätzlich in ihrem Style formatiert werden. Ansonsten kann die Zelle über die entsprechende **Apache POI**-Methode **setCellValueAs<Type>** beschrieben werden. Eine Ausnahme ergibt sich jedoch, wenn der **JSONata**-Ausdruck ein Array zurückliefert. In dem Fall handelt es sich um ein komplexes Datenelement – hier muss überprüft werden, ob überhaupt Einträge vorliegen. Wenn dies der Fall ist, wird über den **arrayIndex** iteriert und für jeden Eintrag eine neue Zeile geschrieben, bis das gesamte Array durchlaufen ist. Die restlichen simplen Dateneinträge aus anderen Spalten werden einfach wiederholt. Ist das komplexe Datenelement allerdings leer, wird geprüft, ob es mit der Kategorie übereinstimmt und ggf. die Zeile gelöscht. Dies trifft zu, wenn ein Produkt beispielsweise gar nicht sterilisierbar ist, sodass auch keine Sterilisationsmethoden vorliegen können. In diesem Fall soll das entsprechende Produkt auch nicht im zugehörigen Arbeitsblatt aufgelistet werden. Liefert ein **JSONata**-Ausdruck kein Ergebnis oder beträgt **null**, obwohl die Spalte verpflichtend ist, kommt es zur Fehlermeldung. Ansonsten werden alle Spalten durchgegangen und beim erfolgreichen Ausfüllen wird die aktualisierte **rowNumber** für das nächste Produkt zurückgegeben.

Auch hier ist bei Arrays bei der Wahl des **JSONata**-Ausdrucks Vorsicht geboten. Es muss bedacht werden, dass nicht vorhandene Datenelemente von komplexen Eltern explizit mit **null** besetzt werden müssen, anstatt sie implizit wegzulassen. Der **JSONata**-Ausdruck wird dadurch ein wenig komplizierter, was man in den Zeilen 29 und 30 des **YAML**-Codes 3.1 sieht. **TradeNames.TradeName** muss beispielsweise zu **TradeNames.(\$exists(TradeName) ? TradeName : null)** erweitert werden. Dadurch wird gewährleistet, dass in jeder Zeile die einander zugehörigen Informationen stehen und sich in den relationalen Arbeitsblättern keine Verschiebungen ergeben, sondern die Zellen der unterschiedlichen Spalten zueinander ausgerichtet sind.

Sind alle Produktdaten erfolgreich in alle Arbeitsblätter geschrieben worden, wird wie in Abschnitt 4.3 im letzten Schritt die ausgefüllte Excel-Datei erzeugt und als **OutputStream** zurückgegeben.

Kapitel 5

Evaluation

Das Ziel dieser Arbeit ist es, verschiedene Prototypen für die Transformationsengine zu implementieren, um daraufhin denjenigen Ansatz zu identifizieren, der am Besten für die Einbindung in die UDI Plattform geeignet ist.

Im folgenden Kapitel werden daher die vorgestellten Lösungsansätze analysiert und miteinander verglichen. Die Grundlinie stellt dabei die spezifische Version aus Kapitel 3.2 dar – ihr gegenüber stehen die generischen Ansätze aus Kapitel 3.3 und 3.4. Unabhängig davon wird zusätzlich die Wahl der JSON-Abfragesprache evaluiert und abschließend auf die Eignung der Transformationsengine für den Massen-Upload eingegangen.

5.1 Gegenüberstellung der drei Ansätze

Die Evaluation untersucht den Nutzen bzw. die Eignung der Transformationsengine-Varianten, indem die verschiedenen Ansätze anhand Kriterien bzgl. Funktionalität und Benutzerfreundlichkeit empirisch bewertet werden, mit dem Zweck sie zu vergleichen, um die beste Lösung für die UDI Plattform zu finden. Näheres zu Evaluationen beschreiben [DGE16] und [Heg03].

Die drei Varianten der Transformationsengine werden mit dem Kapitel, in dem ihr Konzept vorgestellt wurde, nummeriert. Es wird also der spezifische Ansatz 3.2 sowie die generischen Versionen, Ansatz 3.3 mit der Abbildung in der Excel-Datei und Ansatz 3.4 mit der zusätzlichen Mapping-Datei, evaluiert.


Die Kriterien basieren auf den funktionalen und nicht-funktionalen An-

forderung an die Transformationsengine, welche in Tabelle 3.1 spezifiziert wurden, wobei bzgl. der Benutzerfreundlichkeit A–6 noch differenzierter unterteilt wird. In einem Soll/Ist-Vergleich wird die Erfüllung der Ziele überprüft.

Zur Bewertung wird ein einfaches Ampelsystem mit den Stufen

Gut  Kriterium in vollem Maße / gut erfüllt

Mittel  Kriterium teilweise / rudimentär erfüllt

Schlecht  Kriterium nicht / schlecht erfüllt

verwendet, wenngleich einige Kriterien nur  oder  zulassen.

Insgesamt ergibt sich die folgende Evaluationsmatrix:

























Nr.	Kriterium	Ansatz		
		3.2	3.3	3.4
A–1	Datentransformation nach Excel			
A–2	Mapping von Wahrheitswerten			
A–3	Mapping von Datumsformaten			
A–4	Mapping von Element-Wertelisten			
A–5	Aufwand bei Änderung/Erweiterung			
A–6.1	Übersichtlichkeit für Benutzer			
A–6.2	Fehleranfälligkeit für Benutzer			
A–7	Massen-Upload möglich			

Tabelle 5.1: Evaluationsmatrix für die Transformationsengine

Da Ansatz 3.2 lediglich skizziert anstatt in der Praxis implementiert wurde, werden die Kriterien bzgl. ihrer Machbarkeit aus theoretischer Sicht bewertet. Außerdem ist anzumerken, dass das Mapping für Datumsangaben in Ansatz 3.3 nur aus Zeitgründen eingespart wurde, aber durchaus realisiert werden kann. Im Folgenden wird die obige Bewertung noch genauer erläutert. Auf den Massen-Upload wird gesondert in Abschnitt 5.3 eingegangen – die verschiedenen Ansätze unterscheiden sich diesbezüglich nicht, da er vor allem von der Verwendung der Excel-Bibliothek abhängig ist.

Der spezifische **Ansatz 3.2**, welcher auf Hartkodierung setzt, hat den Vorteil, dass die Implementierung absolut unkompliziert ist. Die einzige Herausforderung ist der richtige Umgang mit der Excel-Bibliothek **Apache POI**.

Hier wirkt sich allerdings jede kleine Änderung direkt aus. Nicht nur bei neuen Behörden muss das Package erweitert werden, sondern es ergeben sich auch Codeänderungen, sobald eine Behörde ihre Anforderungen und damit ihre Vorlage minimal abändert. Ebenso wenn intern bzw. beim Hersteller Datenelemente überarbeitet werden. Aufgrund der Hartkodierung müssen Änderungen oder Erweiterungen jedes Mal manuell eingepflegt und ausgerollt werden. Die stets neuen Versionen und ihr Deployment führen zu erheblichem Aufwand, zumindest wenn auf regulatorischer Seite häufiger Anpassungen auftreten bzw. die Plattform kontinuierlich durch neue Märkte erweitert wird.

Daher werden stattdessen **generische Ansätze** zur Lösung angestrebt, die unabhängig von Behörden, Herstellern und Produktinformationen sind. Hier muss bei einer externen oder internen Änderung durch die Behörde oder p36 lediglich die neue (in Ansatz 3.3 überarbeitete) Excel-Vorlage im System hinterlegt und in Ansatz 3.4 außerdem die gespeicherte YAML-Datei aktualisiert werden. Dabei ist entscheidend, dass für diese Änderungen keine speziellen, tiefergehenden Programmierkenntnisse nötig sind, sondern dass auch jemand mit nur oberflächlichem Wissen die Umsetzung ausführen, d. h. die Transformationsengine bedienen, kann. Ansatz 3.2 bietet hingegen keine Benutzung in diesem Sinn.

Der Vorteil von **Ansatz 3.3**, bei dem man die Abbildung in den jeweiligen Excel-Arbeitsblättern verankert, besteht darin, dass nur eine einzige zusätzliche Datei nötig ist und der Nutzer visuell genau vor Augen hat, welche Daten in welche Spalten geschrieben werden. Beim Erstellen des Mappings müssen also nicht zwei Dateien miteinander abgeglichen werden, wodurch Tippfehler minimiert werden. Ein Nachteil ist allerdings, dass die Bearbeitung von Excel-Dateien unbequem ist und insbesondere längere Ausdrücke bei der JSON-Abfrage in den Excel-Zellen nicht mehr leserlich dargestellt werden können.

Ansatz 3.4 geht einen kleinen Umweg über eine zusätzliche Datei zur Beschreibung des Mappings. Im YAML-Format ist sie besonders schlicht, schlank und übersichtlich. Während man die JSONata-Ausdrücke auf den ersten Blick in voller Gänze lesen kann und auch mehrzeilige Funktionen unterstützt werden, muss natürlich die Abbildung als solche zunächst definiert werden. Das heißt, die Namen aller Tabellenblätter, die jeweiligen Startreihen, sowie die Spalten und ihr gewünschter Inhalt müssen abge-

tippt bzw. angegeben werden. Das bringt einen gewissen Aufwand und Fehleranfälligkeit mit sich. Zusätzliche Flexibilität und Struktur erhält man durch die unkomplizierte Definition von Mappings für Wahrheitswerte, Datumsformate oder einzelne Datenelemente. Die YAML-Datei kann hierbei auf unkomplizierte Weise beliebig erweitert werden, falls neue Funktionen implementiert werden – zum Beispiel die Angabe einer Maximalanzahl an Zeilen. Ist der Maximalwert erreicht, werden alle weiteren Produkte in eine neue Arbeitsmappe geschrieben, um Overflow zu vermeiden.

Insgesamt kristallisiert sich der letzte Ansatz als der Beste heraus. Die gewonnene Übersicht durch die klare und intuitive Struktur der YAML-Datei rechtfertigt die zusätzliche Datei und mögliche Übertragungsfehler. Für die UDI Plattform wird daher Ansatz 3.4 empfohlen.

5.2 JSON-Abfragesprache

Neben der bereits angesprochenen Unterscheidung, wo und wie die Mapping Informationen gespeichert werden, also konkret in welcher Datei, ist auch die Wahl der Abfragesprache von Bedeutung. Sie legt zu einem gewissen Teil Form und Umfang der Abbildung zwischen der Excel-Vorlage und den Produktdaten fest. Einen detaillierten Überblick über eine Auswahl verschiedener JSON-Abfragesprachen und ihrer Eigenschaften bzw. Vor- und Nachteile vermittelt das Grundlagenkapitel ??.

Als Einstieg in die Thematik wurde anfänglich `JSONPath` verwendet, wobei sich schnell die ersten Grenzen aufgezeigt haben. Daher wurde in Ansatz 3.3 zunächst auf `JMESPath` gesetzt und später in Ansatz 3.4 auf `JSONata` umgestellt. Die JSON-Abfragesprachen sind unabhängig von den vorgestellten Ansätzen und entsprechend beliebig kombinierbar: Variante 3.3 mit `JSONata`- anstatt `JMESPath`-Ausdrücken im Excel-Arbeitsblatt wäre z. B. genauso möglich.

Die Kriterien für die Evaluation, um die für p36 am besten geeignete Abfragesprache zu ermitteln, betreffen zum einen die Sprache selbst und zum anderen deren Anwendung innerhalb der Transformationsengine. Es wird die Mächtigkeit der Sprache begutachtet – im konkreten Fall heißt das, ob mit ihr die Werte der Datenelemente in die typischerweise von den Behörden gewünschte Form transformiert werden können und darüber hinaus ggf. noch komplexere Abfragen möglich wären. Daneben wird die Erweiterbarkeit durch selbst definierte Funktionen beurteilt, und wie übersichtlich und intuitiv die Syntax ist, sowie das Verhalten im Anwendungsfall

bei komplexen Datenelementen. Die vorhandenen Java-Implementierungen werden bzgl. Umfang, Verbreitung und Aktualität beurteilt. Außerdem wird der Aufwand gemessen, den die Benutzer zur Einarbeitung in bzw. im Umgang mit der Sprache haben.

Unter Verwendung des Ampelsystems aus Abschnitt 5.1 ergibt sich die folgende Evaluationsmatrix, deren Einträge im Nachgang noch genauer erläutert werden.



















Kriterium	JSONPath	JMESPath	JSONata
Mächtigkeit			
Erweiterbarkeit			
Übersichtlichkeit der Syntax			
Abfrage komplexer Elemente			
Java-Bibliothek			
Einarbeitungsaufwand			

Tabelle 5.2: Evaluationsmatrix für die JSON-Abfragesprache

JSONPath ist weit verbreitet, sehr populär und dient sicher als erste Anlaufstelle. Im Vergleich ist es allerdings auch die simpelste und am wenigsten mächtigste Abfragesprache von den hier verwendeten. Die Implementierung von **JSONPath** in Java durch Jayway bietet über die reine Spezifikation hinaus Zusatzfunktionalitäten, wie z. B. einige eingebaute Funktionen und diverse Filtermöglichkeiten, siehe [jay22]. Komplexere Funktionen, wie zum Beispiel den ternären Operator oder **contains** zur Abfrage von Array-Inhalten, werden allerdings nicht unterstützt und können auch nicht innerhalb der Abfrage selbst definiert werden, sondern nur in der Implementierung der Engine. Es gibt hier den großen Vorteil, dass mit der Konfigurationsoption **DEFAULT_PATH_LEAF_TO_NULL** automatisch Nullwerte für fehlende Pfade bzw. Blätter zurückgegeben werden, was bei den relationalen Arbeitsblättern wichtig ist. Wie bereits in Kapitel 4.3 und 4.4 angesprochen, muss hierfür sowohl bei **JMESPath** als auch **JSONata** auf einen Workaround zur absichtlichen Erzeugung von Nullwerten gesetzt werden, der die Ausdrücke etwas aufbläht.

Die Abfragesprache **JMESPath** besticht besonders durch den im Vergleich zu **JSONPath** deutlich angestiegen Umfang und ihre ausführliche Dokumentation, auch wenn die Sprache in ihrer Komplexität zugenommen hat. Mit der Hintereinanderausführung per Pipe-Symbol sind vielfältige Array-manipulationen möglich, die zum Teil Arrays aufblähen und durch `[]` ggf.

wieder abflachen, worunter die Übersichtlichkeit etwas leidet. Außerdem können eigene Funktionen definiert werden, wenn auch nicht in der Abfrage selbst. Insgesamt konnten mit **JMESPath** aber alle bisher aufgetretenen Abfragen in der Transformationsengine realisiert werden. **JMESPath** ist bestrebt in vielen Programmiersprachen als Schnittstelle zur Verfügung zu stehen, so auch in Java. Sie erfreut sich wachsender Popularität, wobei sie allerdings hinter **JSONPath** zurückfällt.

JSONata ist noch mächtiger, allerdings ohne dass die Syntax darunter leidet. Die Sprache hat den großen Vorteil, dass die Nutzer, das heißt die Mitarbeiter des Solution Management, schon Erfahrung mit ihr gesammelt haben, da **JSONata** bereits an anderer Stelle verwendet wird. Es ist also keine größere Einarbeitung oder Umstellung notwendig. Obwohl **JSONata** ursprünglich nur für TypeScript und NodeJS entwickelt wurde, schafft die API **JSONata4Java** Abhilfe, vgl. [IBM22]. Die Bibliothek ist unscheinbar und hat vergleichsweise wenig Nutzer, wird jedoch kontinuierlich gepflegt. Vereinzelte Funktionen wurden bisher nicht implementiert, allerdings kommen sie in der Engine nicht zum Tragen, sodass es im vorliegenden Anwendungsfall kein wirklicher Nachteil ist. Dem gegenüber besteht hier ebenso die Möglichkeit, die Sprache durch eigene Funktionsdefinitionen zu erweitern, und zwar auch innerhalb des **JSONata**-Ausdrucks.

Insgesamt fällt die Wahl der Abfragesprache daher eindeutig auf **JSONata**, deren Umfang und kompakte Schreibweise überzeugen, während sie die Benutzer nicht mit der Wissensaneignung einer zusätzlichen Sprache belastet. Anzumerken ist, dass die Performanz der Sprachen bei der Betrachtung außer Acht gelassen wurde. Diese ist nicht ausschlaggebend für die Transformationsengine, da sie sich in jedem Fall im annehmbaren Bereich bewegt (wie im anschließenden Abschnitt erläutert wird), allerdings schneidet dies bezüglich **JSONata** vermutlich weniger gut ab, vgl. [Zus20] und [mt22]¹.

5.3 Massen-Upload

Ein wichtiges Kriterium ist die Skalierbarkeit über der Anzahl der Produkte, die in die Excel-Vorlage geschrieben werden sollen. Die Hersteller registrieren gerade beim Einstieg in einem neuen Markt bzw. als Neukun-

¹ In den Benchmarks werden die JavaScript-Bibliotheken verglichen, nicht Java. Daher kann für den vorliegenden Fall keine konkrete Aussage bzgl. der Performanz getroffen werden.

de von p36 nicht nur einzelne Produkte, sondern ihr gesamtes Portfolio. Hierzu muss die Transformationsengine in der Lage sein, Daten in der Größenordnung von 1.000 bis 10.000 Produkten zu übermitteln. Für noch größere Uploads besteht keine Anforderung, da man den Prozess ggf. auch wiederholt ausführen kann, im Folgenden werden allerdings die maximalen Grenzen ausgetestet.

Dazu wurden beim finalen Ansatz 3.4 Modultests mit unterschiedlich vielen Produkten ausgeführt. Die Tests haben ergeben, dass bis zu 16.000 Mal das Beispielprodukt in die Vorlage geschrieben werden kann. Bei dem Versuch die Excel-Datei mit 17.000 Produkten abzuspeichern, kommt es zum Heap-Overflow und der `OutOfMemoryError` wird geworfen. Man muss hierbei bedenken, dass in den relationalen Arbeitsblättern für jedes Produkt nicht nur eine Zeile erzeugt wird, sondern mehrere. In den Tests wurde mit dem Faktor 10 gerechnet, das heißt ein Produkt erzeugt zehn Zeilen in der Excel-Datei.

Zur Überprüfung, ob eine Arbeitsmappe erfolgreich ausgefüllt wurde, wird diese nach dem Schreiben erneut geladen, um so die Anzahl der Zeilen auszulesen. Hierbei ergeben sich schon früher Probleme. Ab 10.000 Produkten, das heißt also 100.000 Zeilen, konnte die Excel-Datei nicht mehr über die Bibliothek **Apache POI** eingelesen werden, sondern es kam zum Überlauf des Speichers und der `OutOfMemoryError`-Ausnahme. Daher wurde ab dieser Größe ein Test schon als erfolgreich gezählt, wenn die Datei nur erzeugt wurde und existiert, ohne sie erneut zu öffnen und genauer zu untersuchen. Manuell ist das Öffnen natürlich ohne Probleme möglich und die Daten wurden auch korrekt gefüllt.

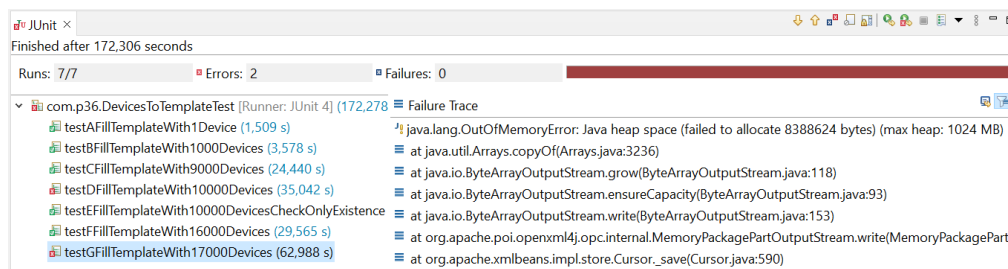


Abbildung 5.1: Testergebnisse zum Massen-Upload

Die Ergebnisse eines Testdurchlaufs sind in Abbildung 5.1 festgehalten. Hier sieht man die Fehlermeldung bei 17.000 Devices. Auch im ersten Test mit 10.000 Devices, bei dem die Datei zur Überprüfung eingelesen wird, kommt es zu einem kleinen Heap-Overflow. Die anderen Tests für bis zu 16.000

Produkten sind grün unterlegt und laufen erfolgreich durch. Der Massen-Upload ist also mit der Transformationsengine realisierbar. Die Dauer des Schreibvorgangs bis zur erfolgreichen Excel-Datei-Erzeugung ist mit nur wenigen Sekunden sehr angenehm. Auch bei vielen Produkten bleibt die Zeit im Bereich von etwa 30 bis 60 Sekunden. Hierbei ist anzumerken, dass die Zeiten im Normalfall (wenn die Testreihenfolge nicht zur besseren Übersicht fixiert ist) stärker variieren, je nachdem welcher Test zuerst durchgeführt wird und damit ohne vorheriges Aufwärmen der JVM am längsten braucht. Trotz dieser Ungenauigkeiten verdeutlichen die Tests gut die Größenordnung der Geschwindigkeit des Transformationsprozesses.

Kapitel 6

Zusammenfassung und Ausblick

Die Bachelorarbeit hat sich damit beschäftigt, Lösungsansätze für eine generische Transformationsengine zu entwickeln und zu evaluieren. Diese hat den Zweck, ausgewählte Daten medizinischer Produkte im JSON-Format in von Gesundheitsbehörden vorgegebene Excel-Dateien zu schreiben. Die finale Version der Transformationsengine wurde bereits in die bestehende Struktur der UDI Plattform des Unternehmens p36 eingebunden und kommt dort bald zum Einsatz.

Zunächst wurde besonders in Kapitel ?? das regulatorische Umfeld in den Life-Sciences näher beleuchtet, in dem die Transformationsengine operiert. Dies befindet sich zur Zeit weltweit in einem Umbruch. Viele Behörden sind dabei, einen Umstrukturierungsprozess zu starten, um die sogenannte UDI, eine eindeutige Produktkennung, verpflichtend einzuführen. Die Hersteller von Medizinprodukten sind damit aktuell vor neue Herausforderungen gestellt, bei denen das Unternehmen p36 sie mit ihrer UDI Plattform unterstützt, um die neuen Regularien umzusetzen und die geforderten Produktdaten UDI-konform bei den Behörden zu hinterlegen. Hier kristallisiert sich der Sinn hinter der Transformationsengine heraus – während große Gesundheitsministerien eine Machine-to-Machine-Schnittstelle zu ihrer Datenbank anbieten, muss bei kleinen Ländern der Upload via Excel-Datei erfolgen.

Da die Transformationsengine für verschiedene Behörden mit teilweise

unterschiedlichen Anforderungen gleichermaßen nutzbar sein soll, muss die Lösung vor allem generisch sein. Um dies zu gewährleisten, müssen die Informationen darüber, welche Daten genau in welcher Spalte von welchem Arbeitsblatt der Excel-Vorlage eingefügt werden sollen, ausgelagert werden und dürfen kein spezifischer Teil der Engine sein. Die Abbildung bzw. Projektion von den Spalten zu den Produktdaten ist dabei der behördenspezifische Teil, der individuell von den Solution Managern bei p36 angepasst werden kann.

Die erste Idee bestand darin, die Abbildung direkt in die Excel-Vorlage zu verlagern. Das Konzept wurde in Kapitel 3.3 detailliert beschrieben und anschließend in Kapitel 4.3 implementiert. Da jedoch längere Ausdrücke in den kleinen Excel-Zellen schnell unleserlich werden, entstand die zweite Idee einer externen Mapping-Datei, in der die Abbildung übersichtlich definiert werden kann. Dabei hat sich letztendlich das YAML-Format durchgesetzt. Das Konzept ist in Kapitel 3.4 erläutert, die Umsetzung erfolgte ausführlich in Kapitel 4.4. Für eine unkomplizierte Integration in die Plattform wurde dabei besonderen Wert darauf gelegt, bereits eingebundene Bibliotheken wiederzuverwenden.

Um die gewünschten Produktdaten zu selektieren, wird sich einer JSON-Abfragesprache bedient. Diese Ausdrücke werden in der Transformationsengine ausgewertet und dann in die jeweiligen Arbeitsblätter geschrieben. In Kapitel ?? werden die bekanntesten Abfragesprachen vorgestellt, von denen im weiteren Verlauf der Entwicklung einige ausgetestet wurden. Am Ende hat sich dabei JSONata als die optimale Sprache für p36 herausgestellt (vgl. Kapitel 5.2), die durch großem Umfang trotz kompakter Syntax besticht. Außerdem bestehen bei den Mitarbeitern schon Vorkenntnisse im Umgang mit JSONata aus anderen Projekten.

Nachdem das grobe Konzept feststand und implementiert war, konnten verschiedene Komplexitäten hinzugefügt werden. Zum einen müssen zusätzliche behördenspezifische Mappings für Wahrheitswerte oder auch Wertelisten einzelner Datenelemente ermöglicht werden. Zum anderen ergab sich eine Schwierigkeit bei fehlenden Pfaden von komplexen Datenelementen, die in einer 1:n-Beziehung zum Produkt stehen und in weiteren Arbeitsblättern mehrzeilig abgebildet werden. Auch Datumsformate werden in Excel wie gewünscht dargestellt. Die YAML-Datei wird per JSON-Schema validiert. Diese und weitere Anforderungen und Funktionalitäten wurden sukzessive und inkrementell in die Transformationsengine eingebaut – ent-

sprechend dem methodisches Vorgehen bei der Entwicklung nach dem Prinzip von Scrum (worauf Kapitel ?? näher eingegangen ist).

Darüber hinaus wurden Modultests für die Funktionalitäten der einzelnen Klassen geschrieben. Im Zuge dessen konnte auch der Massen-Upload erfolgreich getestet werden, denn die Hersteller sind daran interessiert bis zu 10.000 Produkte gleichzeitig hochzuladen, siehe Kapitel 5.3.

Insgesamt ist eine generische Transformationsengine entstanden, die als Eingabe die Produktdaten in Form eines JSON-Strings sowie die Mapping-Datei und die Excel-Vorlage als InputStream nimmt, und die ausgefüllte Excel-Datei als Ausgabe liefert. In den letzten Wochen wurde sie sogar schon in die UDI Platform eingebaut und die umgebende Logik zur Benutzung implementiert, sodass der Integration von neuen Behörden wie Taiwan und Saudi-Arabien nahezu nichts mehr im Weg steht – sofern diese ihre Excel-Templates zeitnah veröffentlichen.

Grundsätzlich befinden sich enorm viele medizinische Regulierungsbehörden aktuell in einer Umbruchphase und weltweit werden in den nächsten Jahren immer mehr Länder auf ein UDI-System umstellen – langfristig vermutlich auf ein global einheitliches System. Bis dahin wird die Transformationsengine aber bis auf Weiteres für viele kleinere Märkte zum Einsatz kommen.

Ich bin gespannt, wie sie sich dabei mit echten Daten innerhalb der UDI Platform verhält und auch, ob die Abbildung in der Mapping-Datei in der Praxis so einfach definiert werden kann wie erhofft. Dies konnte bisher nur mit theoretischen, selbstgeschriebenen Excel-Vorlagen und Produktdaten ohne weiteren Kontext getestet werden.

Als Ausblick kann zusätzlich herausgestellt werden, dass sich die Mapping-Datei beliebig erweitern lässt. Hier ist beispielsweise die Angabe einer maximalen Anzahl an Zeilen pro Arbeitsblatt oder -mappe denkbar, bei deren Überschreiten automatisch eine weitere Datei generiert wird.

Man könnte auch die YAML-Datei als Form in Frage stellen und stattdessen für mehr Benutzerfreundlichkeit die Definition der Abbildung über eine graphische Benutzeroberfläche eingeben, die dann zum Beispiel per Rest-API direkt in der Plattform landet. Hierbei stellt sich letztendlich die Frage, ob der Aufwand in Relation zum Nutzen steht.

Einige Behörden bieten Schnittstellen zu ihrer Datenbank an, die mit XML arbeiten. Anstatt Excel-Dateien könnten durch eine Adaption der Transformationsengine auch XML-Dateien entsprechend der behördlichen

Vorgaben erzeugt werden. Die EUDAMED lässt zum Beispiel den XML-Massen-Upload bestehend aus bis zu 300 einzelnen Datensätzen zu, wobei in diesem Fall natürlich die Verwendung der vorhandenen M2M-Schnittstelle effektiver ist, vgl. [Hou21].

Betrachtet man die in dieser Bachelorarbeit erarbeiteten Themen in einem übergeordneten Zusammenhang, lässt sich anmerken, dass obwohl sich JSON großer Beliebtheit erfreut und vielfältig eingesetzt wird, es zum Beispiel keine einheitliche Standardabfragesprache dafür gibt, genauso wenig wie eine native Schnittstelle in Java, die sich gegen die Drittanbieter-Bibliotheken durchsetzen kann, vgl. [Vit22] und [HDB21]. Die Vielzahl der Sprachen und Bibliotheken, die sich über die Zeit entwickelt haben, ist enorm. Das Positive daran ist, dass die Chance groß ist, dass es für jeden Anwendungsfall bereits die optimale Bibliothek gibt. Andererseits verkümmern viele dieser Bibliotheken wieder, wenn die Entwickler die Projekte nicht mehr weiterverfolgen. Reutter et al. visieren in ihren Fachpublikationen stattdessen eine Vereinheitlichung an, basierend auf der Definition eines formalen JSON-Datenmodells inklusive Abfragesprache [BRV20] sowie einem entsprechend formal verankerten JSON-Schema [PRS16]. Es bleibt abzuwarten, ob sich dieser Vorschlag durchsetzen kann oder nur als weitere Möglichkeit in die bestehende Vielfalt einreicht. Noch einen Schritt weiter wird dagegen in [SKLS21] gegangen mit der Beschreibung eines universelleren Schemas, das JSON als Spezialfall von allgemein durch Knoten strukturierte Datenformate validiert und verarbeitet.

Literaturverzeichnis

- [Abt22] ABT, Dietmar: *Masterkurs Client/Server-Programmierung mit Java: Anwendungen entwickeln mit Standard-Technologien*, 6. Auflage. Wiesbaden: Springer Vieweg, 2022. – DOI: 10.1007/978-3-658-37200-2. – ISBN: 978-3-658-37199-9
- [Ant16] ANTOLOVIC, Miroslav: *Einführung in SAPUI5*, 2. Auflage. Rheinwerk Verlag, 2016. – ISBN: 978-3-8362-8901-6
- [App22a] APPGYVER: *App logic*. 2022. – URL: <https://docs.appgyver.com/docs/app-logic> [abgerufen am 29.11.2022]
- [App22b] APPGYVER: *Community Announcement*. 2022. – URL: <https://forums.appgyver.com/t/community-announcement/19453> [abgerufen am 22.11.2022]
- [App22c] APPGYVER: *View Components*. 2022. – URL: <https://docs.appgyver.com/docs/view-components> [abgerufen am 29.11.2022]
- [AS22] ANDREAS SCHAFFRY, Directorate-COMPUTERWOCHE: *Studie No-Code/Low-Code 2022 Licht und Schatten*. 2022. – URL: <https://www.computerwoche.de/a/licht-und-schatten,3553554> [abgerufen am 11.10.2022]
- [BRV20] BOURHIS, Pierre; REUTTER, Juan L.; VRGOČ, Domagoj: JSON: Data model and query languages. In: *Information Systems* Bd. 89 (2020), Nr. 101478. – DOI: 10.1016/j.is.2019.101478
- [Cen21] CENTER, SAP N.: *SAP übernimmt No-Code-Pionier AppGyver*. 2021. – URL: <https://news.sap.com/germany/2021/02/sap-uebernimmt-no-code-pionier-appgyver/> [abgerufen am 11.10.2022]

- [Cod22a] CODE, Visual S.: *Overview*. 2022. – URL: <https://code.visualstudio.com/docs> [abgerufen am 01.12.2022]
- [Cod22b] CODE, Visual S.: *Terminal Basics*. 2022. – URL: <https://code.visualstudio.com/docs/terminal/basics> [abgerufen am 01.12.2022]
- [Com22a] COMMUNITY, SAP: *SAPUI5*. 2022. – URL: <https://community.sap.com/topics/ui5> [abgerufen am 25.11.2022]
- [Com22b] COMMUNITY, SAP: *Sneak Peek on SAP AppGyver Integration Into SAP BTP*. 2022. – URL: <https://groups.community.sap.com/t5/devtoberfest/sneak-peek-on-sap-appgyver-integration-into-sap-btp/ec-p/8958#M17> [abgerufen am 13.10.2022]
- [Con22] CONSULTING, Hecker: *Die Zukunft der Software Entwicklung: Low-Code-Plattformen*. 2022. – URL: <https://www.hco.de/blog/die-zukunft-der-software-entwicklung-low-code-plattformen> [abgerufen am 11.11.2022]
- [CVE22] COEN VAN EENBERGEN, Directorate-TECHZINE: *ServiceNow fully enters low-code market with App Engine Studio*. 2022. – URL: <https://www.techzine.eu/blogs/applications/56875/servicenow-fully-enters-low-code-market-with-app-engine-studio/> [abgerufen am 17.11.2022]
- [DGE16] DEGEVAL, Gesellschaft für Evaluation (Hrsg.): *Standards für Evaluation – Erste Revision*. Mainz, 2016. – ISBN: 978–3–941569–06–5
- [Doc22a] DOCUMENTATION, SAPUI5: *Developing Apps with SAP Fiori Elements*. 2022. – URL: <https://sapui5.hana.ondemand.com/#/topic/03265b0408e2432c9571d6b3feb6b1fd> [abgerufen am 28.11.2022]
- [Doc22b] DOCUMENTATION, SAPUI5: *Using SAP Fiori Elements Floorplans*. 2022. – URL: <https://sapui5.hana.ondemand.com/#/topic/03265b0408e2432c9571d6b3feb6b1fd> [abgerufen am 28.11.2022]

- [doc22c] DOCUMENTATION, ServiceNow P.: *App Engine Studio release notes*. 2022. – URL: <https://docs.servicenow.com/bundle/store-release-notes/page/release-notes/store/platform-app-engine/store-rn-plat-app-engine-aes.html> [abgerufen am 17.11.2022]
- [Dro22] DROETTBOOM, Michael u. a.: *Understanding JSON Schema – Release 2020-12*. Space Telescope Science Institute, 2022. – URL: <https://json-schema.org/understanding-json-schema/UnderstandingJSONSchema.pdf> [abgerufen am: 24.08.2022]
- [EH11] ERIKSSON, Malin; HALLBERG, Victor: *Comparison between JSON and YAML for Data Serialization* (Bachelorarbeit). Stockholm, Schweden: KTH, School of Computer Science and Communication (CSC), 2011. – URN: <https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-130815>
- [Ele22] ELEMENTS, SAP F.: *SAP Fiori Design Guidelines*. 2022. – URL: <https://experience.sap.com/fiori-design-web/smart-templates/> [abgerufen am 13.10.2022]
- [Eng20] ENGLBRECHT, Michael: *SAP Fiori Implementierung und Entwicklung*, 3. Auflage. Rheinwerk Verlag, 2020. – ISBN: 978–3–8362–7268–1
- [Fas22] FASTERXML, LLC: *Jackson Project Home @github*. 2022. – URL: <https://github.com/FasterXML/jackson> [abgerufen am 07.07.2022]
- [Fri19] FRIESEN, Jeff: *Java XML and JSON: Document Processing for Java SE*, 2. Auflage. Berkeley, CA: Apress, 2019. – DOI: 10.1007/978–1–4842–4330–5. – ISBN: 978–1–4842–4329–9
- [G222] G2: *Salesforce Platform Reviews and Product Details*. 2022. – URL: <https://www.g2.com/products/salesforce-platform/reviews> [abgerufen am 18.11.2022]
- [GG22] GARTNER GLOSSARY, Directorate-Gartner: *Citizen Developer*. 2022. – URL: <https://www.gartner.com/en/information-technology/glossary/citizen-developer> [abgerufen am 11.11.2022]

- [GHK14] GALLARDO, Raymond; HOMMEL, Scott; KANNAN, Sowmya; GORDON, Joni; ZAKHOUR, Sharon B.: *The Java® Tutorial: A Short Course on the Basics*, 6. Auflage. Upper Saddle River, NJ: Addison-Wesley, 2014. – ISBN: 978-0-13-403408-9
- [Gmb22] GMBH p36: *p36 Überblick*. 2022. – URL: <https://p36.io/professional-services/sap-consulting/?lang=de> [abgerufen am 11.10.2022]
- [Gui22] GUIDELINES, SAP Fiori D.: *SAP Fiori Launchpad*. 2022. – URL: <https://experience.sap.com/fiori-design-web/launchpad/> [abgerufen am 24.11.2022]
- [HDB21] HARRAND, Nicolas; DURIEUX, Thomas; BROMAN, David; BAUDRY, Benoit: The Behavioral Diversity of Java JSON Libraries. In: *32nd International Symposium on Software Reliability Engineering (ISSRE) – Proceedings*, S. 412–422. Wuhan, China: IEEE, 2021. – ISBN: 978-1-6654-2587-2. – DOI: 10.1109/ISSRE52982.2021.00050
- [Heg03] HEGNER, Marcus: *Methoden zur Evaluation von Software* (IZ-Arbeitsbericht Nr. 29). Bonn: Informationszentrum Sozialwissenschaften, 2003. – ISSN: 1431-6943
- [Hou21] HOULIHAN, Richard: *EUDAMED: Add my device data*. 2021. – URL: <https://eudamed.com/index.php/2021/08/21/eudamed-database/> [abgerufen am 11.08.2022]
- [HV22] HEINRICH VASKE, Directorate-COMPUTERWOCHE: *Low-Code-Plattformen auf einen Blick*. 2022. – URL: <https://www.computerwoche.de/a/low-code-plattformen-auf-einen-blick,3544905> [abgerufen am 10.10.2022]
- [IBM22] IBM, International Business Machines: *JSONata4Java*. 2022. – URL: <https://github.com/IBM/JSONata4Java> [abgerufen am 20.07.2022]
- [ISPE22] ISPE, International Society for Pharmaceutical Engineering (Hrsg.): *GAMP® 5: A Risk-Based Approach to Compliant GxP Computerized Systems*, 2. Auflage. Tampa, FL, 2022. – ISBN: 978-1-946964-57-1

- [Jac16] JACKSON, Wallace: *JSON Quick Syntax Reference*, 1. Auflage. Berkeley, CA: Apress, 2016. – DOI: 10.1007/978-1-4842-1863-1. – ISBN: 978-1-4842-1862-4
- [jay22] *Jayway JsonPath*. 2022. – URL: <https://github.com/json-path/JsonPath> [abgerufen am 10.08.2022]
- [Kle19] KLEUKER, Stephan: *Qualitätssicherung durch Software-tests: Vorgehensweisen und Werkzeuge zum Testen von Java-Programmen*, 2. Auflage. Wiesbaden: Springer Vieweg, 2019. – DOI: 10.1007/978-3-658-24886-4. – ISBN: 978-3-658-24885-7
- [Lea22] LEARNING, SAP: *Getting Started With Low-Code/No-Code and SAP Build*. 2022. – URL: https://learning.sap.com/learning-journey/utilize-sap-build-for-low-code-no-code-applications-and-automations-for-citizen-developers/getting-started-with-low-code-no-code-and-sap-build_afaa85b8-93eb-4607-94b8-221c80aa6479 [abgerufen am 18.11.2022]
- [Len22] LENTZ, Bethaney: *Quick reference guide - global medical device UDI requirements and timelines*. 2022. – URL: <https://www.rimsys.io/blog/quick-reference-guide-global-udi-requirements-and-timelines> [abgerufen am 22.08.2022]
- [Mar22] MARKETPLACE, Visual S.: *SAP CDS Language Support*. 2022. – URL: <https://marketplace.visualstudio.com/items?itemName=SAPSE.vscode-cds> [abgerufen am 19.12.2022]
- [mt22] *JSON Query Languages*. 2022. – URL: <https://www.measurethat.net/Benchmarks/Show/20654> [abgerufen am 27.08.2022]
- [NMA21] DÖT NET, Ingy; MÜLLER, Tina; ANTONIOU, Pantelis; ARO, Eemeli; SMITH, Thomas: *YAML Ain't Markup Language (YAML™) Version 1. – Revision 1.2.2*. 2021. – URL: <https://yaml.org/spec/1.2.2/> [abgerufen am 12.08.2022]
- [OB22] OLIVER, Andrew C.; BAROZZI, Nicola K.: *POI-HSSF and POI-XSSF/SXSSF – Java API To Access Microsoft Excel Format Files*. 2022. – URL: <https://poi.apache.org/components/spreadsheet/index.html> [abgerufen am 07.07.2022]

- [Ove22a] OVERVIEW, G2: *228 Listings in Low-Code Development Platforms Available*. 2022. – URL: <https://www.g2.com/categories/low-code-development-platforms> [abgerufen am 11.11.2022]
- [Ove22b] OVERVIEW, G2: *289 Listings in No-Code Development Platforms Available*. 2022. – URL: <https://www.g2.com/categories/no-code-development-platforms> [abgerufen am 11.11.2022]
- [Por22] PORTAL, SAP H.: *What Is SAP Business Application Studio*. 2022. – URL: <https://help.sap.com/docs/SAP%20Business%20Application%20Studio/9d1db9835307451daa8c930fbd9ab264/8f46c6e6f86641cc900871c903761fd4.html> [abgerufen am 29.11.2022]
- [PRS16] PEZOA, Felipe; REUTTER, Juan L.; SUAREZ, Fernando; UGARTE, Martín; VRGOČ, Domagoj: *Foundations of JSON Schema*. In: *WWW '16: Proceedings of the 25th International World Wide Web Conference, Montréal, Kanada*, S.263–273. Genf, Schweiz: International World Wide Web Conferences Steering Committee, 2016. – ISBN: 978–1–4503–4143–1. – DOI: 10.1145/2872427.2883029
- [QOS22] QOS.CH: *Simple Logging Facade for Java (SLF4J)*. 2022. – URL: <https://www.slf4j.org/index.html> [abgerufen am 07.07.2022]
- [Raj21] RAJPUT, Varsha: *Top 2 Alternatives To Apache POI Library That Can Actually Make Your Life Better*. 2021. – URL: <https://automationqahub.com/top-2-alternatives-to-apache-poi-library-that-can-actually-make-your-life-better/> [abgerufen am 27.08.2022]
- [Ren21] RENAUD, Fabien: *Benchmark of Java JSON libraries*. 2021. – URL: <https://github.com/fabienrenaud/java-json-benchmark> [abgerufen am 26.08.2022]
- [RJR22] RICHARDSON, Clay; JOHN RYMER, Directorate-FORRESTER: *New Development Platforms Emerge For Customer-Facing Applications*. 2022. – URL: <https://www.forrester.com/report>

- /New-Development-Platforms-Emerge-For-CustomerFacing-Applications/RES113411 [abgerufen am 10.10.2022]
- [Sai22] SAILPOINT TECHNOLOGIES, Inc.: *GxP-Compliance – ein kurzer Überblick*. 2022. – URL: <https://www.sailpoint.com/de/identity-library/gxp-compliance-ein-kurzer-uberblick/> [abgerufen am 09.08.2022]
- [SAP22a] SAP: *About CAP*. 2022. – URL: <https://cap.cloud.sap/docs/about/> [abgerufen am 01.12.2022]
- [SAP22b] SAP: *AppGyver*. 2022. – URL: <https://www.appgyver.com/> [abgerufen am 10.10.2022]
- [SAP22c] SAP: *Generator-easy-ui5*. 2022. – URL: <https://github.com/SAP/generator-easy-ui5> [abgerufen am 01.12.2022]
- [SDW20] SCHAAF, Sebastian; DÖRPINGHAUS, Jens; WEIL, Vera: *Java für die Life Sciences: Eine Einführung in die angewandte Bioinformatik*, 1. Auflage. Heidelberg: O'Reilly, 2020. – ISBN: 978-3-96009-125-7
- [SKLS21] SPIVAK, Iryna; KREPYCH, Svitlana; LITVYNCHUK, Mykola; SPIVAK, Serhii: Validation and Data Processing in JSON Format. In: *IEEE EUROCON 2021: 19th International Conference on Smart Technologies – Conference Proceedings*, S. 326–330. Lviv, Ukraine: IEEE, 2021. – ISBN: 978-1-6654-3299-3. – DOI: 10.1109/EUROCON52738.2021.9535582
- [Vit22] VITZ, Michael: JSON in Java verarbeiten – Ein Einstieg in vier JSON-Bibliotheken für Java. In: *JavaSPEKTRUM* Bd. 1 (2022), S. 54–58. – URL: <https://www.innoq.com/de/articles/2022/02/java-json/>
- [VK18] VANURA, Jan; KRIZ, Pavel: Performance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats. In: FERREIRA, João E.; SPANOUDAKIS, George; MA, Yutao; ZHANG, Liang-Jie (Hrsg.): *Services Computing – SCC 2018, 15th International Conference, Seattle, WA* (Lecture Notes in Computer Science Nr. 10969), S. 166–175. Cham, Schweiz: Springer, 2018. – ISBN: 978-3-319-94376-3. – DOI: 10.1007/978-3-319-94376-3_11

- [Wes06] WESTPHAL, Frank: *Testgetriebene Entwicklung mit JUnit & FIT – Wie Software änderbar bleibt*, 1. Auflage. Heidelberg: dpunkt.verlag, 2006. – ISBN: 978-3-89864-996-4
- [Wik22a] WIKIPEDIA: *Node.js*. 2022. – URL: <https://en.wikipedia.org/wiki/Node.js> [abgerufen am 14.12.2022]
- [Wik22b] WIKIPEDIA: *SQLite*. 2022. – URL: <https://de.wikipedia.org/wiki/SQLite> [abgerufen am 14.12.2022]
- [Wik22c] WIKIPEDIA: *Visual Studio Code*. 2022. – URL: https://de.wikipedia.org/wiki/Visual_Studio_Code [abgerufen am 01.12.2022]
- [ZS22] ZWITSERLOOT, Reinier; SPILKER, Roel: *Project Lombok – features*. 2022. – URL: <https://projectlombok.org/features/> [abgerufen am 24.08.2022]
- [Zus20] ZUSCHLAG, Cody: *The JSONata Performance Dilemma*. 2020. – URL: <https://www.nearform.com/blog/the-jsonata-performance-dilemma/> [abgerufen am 24.08.2022]

Anhang A

JSON-Schema

JSON-Schema wird verwendet, um die Struktur von JSON-Dateien zu validieren. Bei der Transformationsengine wird dieses Prinzip bei der Mapping-Datei aus Kapitel 3.4 angewendet – besonders in Abschnitt 3.4.1 wird detailliert auf JSON-Schema eingegangen. Das verwendete Schema ist in der Datei `MappingJsonSchema.json` gespeichert und wie folgt definiert:

```
1 {
2   "$schema": "https://json-schema.org/draft-06/schema#",
3   "title": "jsonataExcelMapping",
4   "description": "A mapping for JSONata input into excel template",
5   "type": "object",
6   "properties": {
7     "SheetMappings": {
8       "description": "mappings for the sheets in the excel template",
9       "type": "object",
10      "additionalProperties": {
11        "description": "different sheetNames",
12        "type": "object",
13        "properties": {
14          "row": {
15            "description": "start row in excel sheet where the device
16                          data shall be written",
17            "type": "integer",
18            "minimum": 1
19          },
20          "category": {
21            "description": "category of the sheet (complex data
22                          element key)",
23            "type": "string"
24          },
25          "columns": {
26            "description": "different columns to be filled",
```

```

25     "type": "object",
26     "patternProperties": {
27         "[a-zA-Z]{1,2}$": {
28             "description": "the key is the column letter in excel
                               format, the value is the JSONata expression to
                               filter the right device data for this column",
29             "type": "string"
30         }
31     },
32     "additionalProperties": false,
33     "minProperties": 1
34 },
35     "mandatoryColumns": {
36         "type": "array",
37         "description": "a list of those columns that are required
                           to be filled",
38         "default": [],
39         "items": {
40             "type": "string",
41             "pattern": "[a-zA-Z]{1,2}$"
42         }
43     }
44 },
45     "required": [ "row", "columns" ]
46 }
47 },
48     "ElementMappings": {
49         "description": "mappings for booleans, date formats and/or
                           specific data elements",
50         "type": "object",
51         "properties": {
52             "Boolean": {
53                 "description": "mapping for boolean values in this template
                                   ",
54                 "type": "object",
55                 "properties": {
56                     "true": {
57                         "description": "mapping of true in this agency",
58                         "type": ["number", "string", "boolean"]
59                     },
60                     "false": {
61                         "description": "mapping of false in this agency",
62                         "type": ["number", "string", "boolean"]
63                     }
64                 },
65                 "additionalProperties": false,
66                 "required": ["false", "true"]
67             },
68             "Date": {
69                 "description": "mapping for date or time formats",
70                 "type": "object",
71                 "additionalProperties": {

```

```
72         "description": "key: date format for p36 / value: date  
73             format in excel for agency",  
74         "type": "string"  
75     }  
76 },  
77     "additionalProperties": {  
78         "description": "data element key",  
79         "type": "object",  
80         "additionalProperties": {  
81             "description": "key: element value for p36 / value: element  
82                 value for agency",  
83             "type": ["number", "string", "boolean"]  
84         }  
85     }  
86 },  
87     "required": [ "SheetMappings" ]  
88 }
```

Quelltext A.1: JSON-Schema für die Mapping-Datei