Effektivitätssteigerung der Datenauswertung durch Implementierung einer Datevschnittstelle im firmeninternen Data Warehouse mit anschließender Datenaufbereitung

Projektarbeit I

Vorgelegt am:	01.05.2019
Studienbereich:	Technik
Studienrichtung	Praktische Informatik
Kurs:	
Von:	
Matrikelnummer:	
Ausbildungsstätte:	dotSource GmbH
Betreuer Praxisbetrieb:	
Gutachter Duale Hochschule:	

Sperrvermerk

Die vorliegende Arbeit beinhaltet interne und vertrauliche Informationen der Firma dotSource GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften – auch in digitaler Form – gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma dotSource GmbH.

Inhaltsverzeichnis

Inhaltsverzeichnis I
Abbildungsverzeichnis II
Tabellenverzeichnis III
Abkürzungsverzeichnis IV
1 Einleitung 1
2 Grundlagen 2
2.1 Microsoft Azure 2
2.1.1 Konzept Cloud Computing 2
2.1.2 Data Factory und Logic Apps 3
2.1.3 Azure Functions 4
2.1.4 Microsoft SQL-Server 4
2.2 DATEVconnect 5
2.3 Power BI 5
3 Theoretischer Aufbau eines Data Warehouses 7
3.1 Allgemeiner Aufbau 7
3.2 Aufbau laut Microsoft Azure 9
4 Ausgangssituation 11
4.1 Systemarchitektur des dotSource Data Warehouse 11
4.2 Bestehende Quellsysteme 13
5 Anforderungen 15
6 Umsetzung 16
6.1 Laden der Daten unter Verwendung der REST API und Azure 16
6.1.1 Grundaufbau DATEVconnect in der dotSource 16
6.1.2 Requests zum Laden der Anforderungen 17
6.1.3 Automatisierung des Imports durch eine Azure Logic App 19
6.2 SQL-seitige Verarbeitung der Daten 21
6.3 Reporting Layer bevölkern 26
7 Probleme während der Umsetzung 27
8 Fazit 28
Literaturverzeichnis V
Anhangsverzeichnis VII
Ehrenwörtliche Erklärung

Abbildungsverzeichnis

Abbildung 1: Systemarchitektur Azure 2
Abbildung 2: DATEVconnect Setup 5
Abbildung 3: Power BI-Beispieldashboard 6
Abbildung 4: DWH-Konstrukt 9
Abbildung 5: Azure Modernes DWH 10
Abbildung 6: Aufbau DSDWH 11
Abbildung 7: Beispiel Buchungstabellen 13
Abbildung 8: Kostenstellen Response 19
Abbildung 9: Logic App erstellen 20
Abbildung 10: Logic App Trigger Recurrence 20
Abbildung 11: SQL-Skript erstellen der Stagetabelle 22
Abbildung 13: SQL-Skript erstellen der Satellitentabelle 22
Abbildung 14: SQL-Skript erstellen der Hubtabelle 22
Abbildung 15: SQL-Skript erstellen der Säuberungssicht 23
Abbildung 16: SQL-Skript erstellen der Prozedure zum Laden der Stage 24
Abbildung 17: SQL-Skript erstellen der Prozedur zum Laden des Hubs 24
Abbildung 18: SQL-Skript erstellen der Prozedure zum Laden der Detaitabelle 25

Tabellenverzeichnis

Tabelle 1: Anforderungen, die vom *Datascientist* bestimmt wurden 15

Abkürzungsverzeichnis

API	Application Programming Interface
Blob	Binary large object
CSV	Comma-Separated Values
dS	dotSource
DSDWH	dotSource Data Warehouse
DWH	Data Warehouse
ETL	Extract Transform Load
HaSA	Hub-and-Spoke-Architektur
HTTP	Hyptertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
LA	Logic App
PK	Primary Key
REST API	Representational State Transfer API
SaaS	Software as a Service
SID	Secure Identifier
SQL	structured query language
TSQL	Transact-SQL
VPN	Virtual Private Network

1 Einleitung

Ein *Data Warehouse* (DWH) ist eine spezifische Softwarearchitektur, die Unternehmen bei der Bündelung und Auswertung ihrer Daten unterstützen soll. Diese Daten werden aus den verschiedenen Quellsystemen (z.B. Jira, Gitlab, DATEV, Buisness Cordination Software, …) importiert, welche firmenintern genutzt werden.

Gerade bei einer großen Anzahl an Quellsystemen oder einer enormen Anzahl an zu verarbeitenden Daten erweist sich das Konzept eines DWH als äußerst nützlich, da es die gebündelten Daten nicht nur zentral an einem Ort zusammenführt, sondern auch so aufbereitet und logisch verknüpft, dass mit Hilfe von Analyseprogrammen, wie beispielsweise Power BI, ohne weiteres Zusammenhänge ersichtlich werden können. Dadurch kann dem Unternehmen eine effizientere Arbeitsweise ermöglicht werden. So können Probleme wie die Inkonsistenz der Daten oder unterschiedliche Arbeitsweisen der Mitarbeiter überwunden werden, indem eine einheitliche Datenbasis für die Arbeit geschaffen wird.

Auch die dotSource nutzt ein hauseigenes DWH, um ihre Daten aus den verschiedenen Quellsystemen zusammenzuführen und auszuwerten. Allerdings fehlt es momentan an Finanzdaten aus dem DATEV, wodurch keine aussagekräftige Kosten- und Finanzanalyse beim Teamcontrolling gegeben ist.

Im Zuge dieser Arbeit soll das dotSource *Data Warehouse* (DSDWH) um die Funktionalität einer Datevschnittstelle erweitert werden. Neben dem Zusammenführen der Finanzdaten aus der Finanzabteilung werden diese Daten auch aufbereitet, sodass sie mit Buchungs- und Verwaltungsdaten verknüpfbar sind.

Hierzu wird sowohl der grundlegende Aufbau eines DWHs dargestellt als auch die letztendliche Realisierung innerhalb der dotSource. Dazu zählt auch die Vorstellung der einzelnen Tools, aus denen sich das DWH grundlegend zusammensetzt, sowie die Anforderungen an die Datevschnitstelle. Danach folgt die technische Umsetzung der Vorbetrachtung mit einer anschließenden Analyse der Problematiken während der Umsetzung und ein Ausblick auf einen möglichen zukünftigen Nutzen für die Firma.

2 Grundlagen

2.1 Microsoft Azure

2.1.1 Konzept Cloud Computing

Bei dem vom amerikanischen Technologiekonzern Microsoft bereitgestellten Dienst *Azure* handelt es sich um eine Cloud-Computing-Lösung für kleinere bis größere Unternehmen. Dabei werden IT-Ressourcen zur Verfügung gestellt, welche sonst eigenständig bereitgestellt werden müssten. Die *Azure Cloud* bringt nicht nur *Virtual Machines*, sondern auch umfangreiche Analysetools und Grafikoberflächen, mit denen Geschäftsprozesse visualisiert werden können, mit.

Um die Anforderungen ihrer Kunden zu erfüllen, bedient sich der Clouddienst dreier fundamentaler Servicemodelle.[1] So haben *User* die Möglichkeit, auf visualisierte Hardware und Server für ihre Businesslösungen zuzugreifen und deren Rechnerleistung zu nutzen. Dies umfasst auch *Cloud*-Speicherplatz sowie die Rechnerkapazität im Allgemeinen. Dadurch wird Enterprise-Kunden eine kostengünstige und einfach skalierbare IT-Lösung ermöglicht. Der wie in Abbildung 1 als *Infrastructure as a Service* betitelte Service wird jedoch fast ausschließlich in Verbindung mit den weiteren zwei Modellen genutzt.

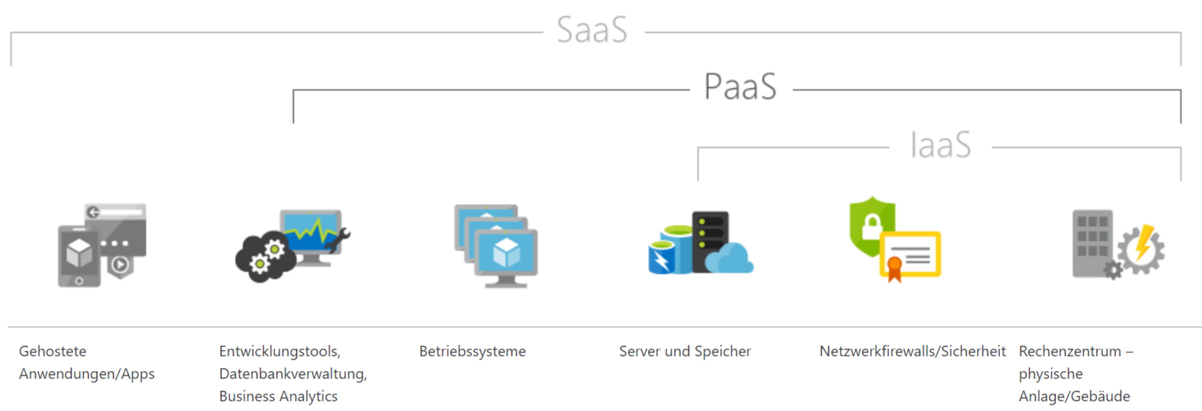


Abbildung 1: Systemarchitektur *Azure*[2]

---

[1] siehe Abbildung 1
[2] [Mic19e]

Platform as a Service ist der Teil in der Cloud, durch den Kunden „Zugang zu den erforderlichen Ressourcen erhalten, um verschiedenste Lösungen“[3] nutzen zu können. Der Benutzer spart dadurch Zeit und Aufwand beim Erstellen von Code für neue *Applications*, was eine einfachere und schneller zu erlernende Bedienung ermöglicht.[4]

Das dritte verwendete Servicemodell setzt mit seinen Funktionen auf den vorrangegangenen Services auf. *Software as a Service* (SaaS) stellt cloudbasierte *Applications* zur Verfügung. Insbesondere „E-Mail-, Kalender- und Office-Tools.“[5] Neben den offensichtlichen Kosteneinsparungen muss, aufgrund der Tatsache, dass nur das abgerechnet wird, was auch wirklich genutzt wird, nicht zwangsläufig Hardware erworben werden. Dies ermöglicht eine einfache Skalierung der *Applications,* bei wachsenden Anforderungen im sich ständig verändernden Internet.

2.1.2 Data Factory und Logic Apps

In der *Azure Cloud* existieren für die Datenverarbeitung und Automatisierung unter dem SaaS Modell zwei wesentliche Elemente, die *Data Factory* und sogenannte *Logic Apps* (LA). Letztere werden hauptsächlich zum Automatisieren von z.B. E-Mail-Prozessen verwendet. Ihr Vorteil ist, dass sie mit vielen Drittanbietern *Application Programming Interfaces* (API) und vielen hauseigenen APIs breit aufgestellt sind. Darüber hinaus bieten sie eine leicht zu bedienende Grafikoberfläche, in welcher der spätere Prozessablauf übersichtlich dargestellt wird[6].

Zum anderen besitzt auch die *Data Factory* eine einfach verständliche grafische Oberfläche und weist mit ihren *Pipelines* eine Methode auf, um Geschäfts- bzw. Datenprozesse zu modellieren und ihre Verarbeitung genau zu steuern und zu überwachen. Es gibt zwar keinerlei Drittanbieter APIs, da die *Data Factory* noch recht neu ist und bisher keine Integration von externen Anbietern ermöglicht, aber es existieren sogenannte *Databricks*. Diese Blöcke im grafischen Ablaufplan der LA enthalten selbstgeschriebenen Code und unterstützen eine weite Variation an Programmiersprachen für die Datenverarbeitung, wie „Python, Scala, R, [und] SQL“[7]. Microsoft versucht mit der *Data Factory* Anwender von *Machine* und *Deep Learning*

---

[3] Ebenda
[4] vgl. Ebenda
[5] [Mic19f]
[6] siehe Anhang 1
[7] [Mic19b]

für sich zu begeistern, indem sie *Frameworks* wie *TensorFlow, Pytorch* und *Scikit-learn* unterstützen.[8]

Der genaue Unterschied zwischen diesen zwei Elementen liegt im eigentlichen Einsatzgebiet. Die *Data Factory* dient hauptsächlich dem Duplizieren, Erstellen und Verarbeiten von größeren Datenmengen.[9] Hingegen sind LA mit einem *Timeout* nach 60 Sekunden eher für einfachere Prozesse, wie Bestätigungsemails oder kleinere Abfragen auf SQL-Servern, gedacht.

2.1.3 Azure Functions

Die *Azure Functions* sind ein weiterer Baustein, welcher innerhalb von LA oder *Data Factory Pipelines* verwendet werden kann. Im Prinzip besitzen sie eine Funktionsweise wie *Databricks* der *Data Factory*, mit dem Unterschied, dass sie noch mehr Sprachen unterstützen bzw. unterstützen sollen.[10]

In Bezug auf die unterstützten Programmiersprachen zeigt sich, dass *Functions* eher eine allgemeine Aufgabe erfüllen sollen, unter anderem *Requests* aus dem *Frontend* verarbeiten oder gar das gesamte *Backend* einer *Application* oder SaaS Anwendung darstellen[11] und nach der Abarbeitung ihrer Logik auch Daten wieder zurückgeben könnten. *Databricks* hingegen besitzen nicht die Möglichkeit einen *Response* mit Daten zu senden, sondern können nur ihre Ergebnisse in einer gewählten Speicherlösung ablegen.

2.1.4 Microsoft SQL-Server

Der Microsoft *structured query language* (SQL) -Server ist ein weiterer Teil der *Azure Cloud*. Hierbei handelt es sich um einen vollständigen relationalen Clouddatenbankdienst, welcher eine einfache Migration von SQL Server-Datenbanken ermöglicht und lückenlos in die restlichen *Azure* Dienste eingebunden ist. Er baut auf der *Query Language Transact*-SQL (TSQL), eine proprietäre Erweiterung des SQL-Standards, auf. Für die meisten *Applications* ist

[8] vgl. Ebenda
[9] vgl. [Sef18]
[10] vgl. [Mic18d]
[11] vgl. [Mic18b]

die Clouddatenbank der Hauptspeicherort für Backenddaten oder fungiert als Knotenpunkt für den Datenaustausch und die Verarbeitung.[12]

2.2 DATEVconnect

Um automatisiert auf seine eigenen Datevdaten außerhalb der Standard Datevsoftware zuzugreifen, wurde von DATEV eine Schnittstelle mit dem Namen DATEVconnect bereitgestellt (siehe Abbildung 2). Diese wurde auf Basis der bestehenden Software entwickelt und setzt auf diese auf. Sie bedient sich der softwareseitigen Funktionen, um Zugriff auf Datevdaten mit Hilfe einer *Representational State Transfer* API (REST API) zu gewährleisten.

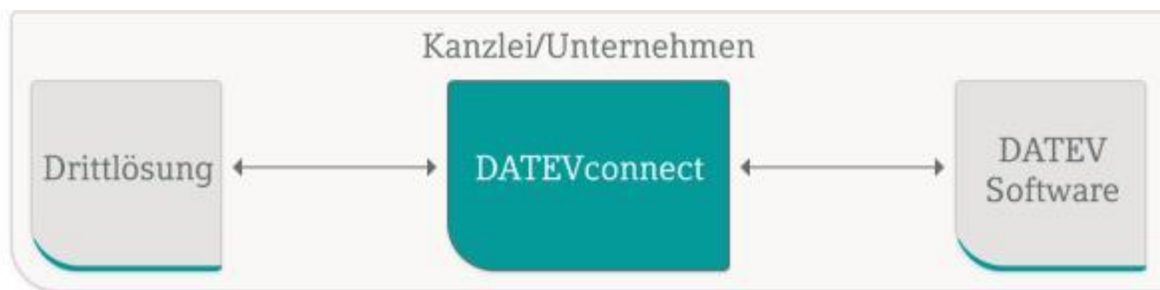


Abbildung 2: DATEVconnect Setup[13]

2.3 Power BI

Microsoft bietet für alle möglichen Datensätze ein eigenes Analysetool an. Mit Power BI können Daten über die verschiedensten Wege importiert werden. So kann der *User* Daten mit Hilfe von Excel, aber auch direkt aus einer SQL-Datenbank, durch Anbindung von *Azure* laden. Der Benutzer ist jedoch nicht auf einfache *Comma-Separated Value* (CSV) Dateien und *Azure* Datenbanken beschränkt, sondern kann frei seine Datenquellen wählen. So können eigene Datenserver oder gar Web-APIs mit Power BI verbunden werden, um seine Daten zu visualisieren.

Importierte Daten können dann mit verschiedenen Aggregatfunktionen verarbeitet und logisch verknüpft werden.[14] Die Umsetzung erfolgt dabei per Drag and Drop, wodurch auf einfache Weise aussagekräftige Graphen und Diagramme erstellt werden können. Hinzu kommt die Möglichkeit eigene Rechnungen und komplexere Abfragen unter Zuhilfenahme von *Data*

[12] vgl. [Mic19d]
[13] [DAT18]
[14] vgl. [Mic18a]

Analysis Expressions zu erstellen. Die sogenannten *Measures* können wie in Abbildung 3 zu sehen mit den gegebenen Graphen kombiniert werden, um eine genauere und auf die eigenen Probleme zugeschnittene Analyse zu ermöglichen.

Auch die Anwendung von Programmiersprachen, wie Python oder R, können genutzt werden bei der Verwirklichung solcher Grafiken und Diagramme.

Aufgrund dieser Funktionen ist Power BI meist das Endglied in einem DWH oder einer Datenpipeline. Dort können gesammelte und verarbeitete Daten visualisiert werden, ohne die technischen Hintergründe der Beschaffensprozesse verstehen zu müssen. Daher wird es hauptsächlich von Projektmanagern, Finanzmitarbeitern und Verwaltungsangestellten eines Unternehmens verwendet.

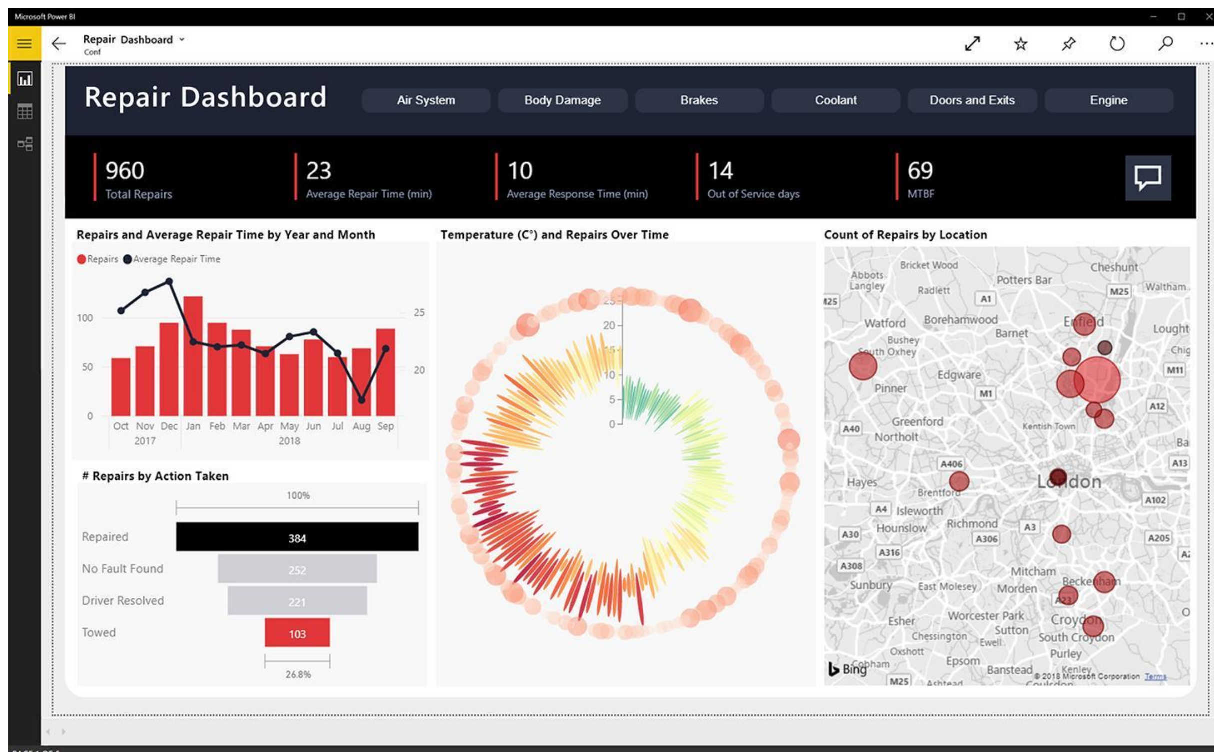


Abbildung 3: Power BI-Beispieldashboard[15]

[15] [Mic19c]

3 Theoretischer Aufbau eines Data Warehouses

3.1 Allgemeiner Aufbau

Im Allgemeinen dient ein DWH dazu, eine homogene Sicht auf vorangegangene, heterogene und operative Systeme zu ermöglichen. Zu diesem Zweck müssen die Daten der Quellsysteme im DWH zusammengeführt und soweit angepasst bzw. in Zusammenhang gebracht werden, dass eine Homogenität gegeben ist. Nur durch diese Einheit erfolgt eine logische Bewertung von Daten mit Hilfe von Analysetools.

Grundsätzlich gibt es drei große Hauptabschnitte in einem DWH (siehe Abbildung 4). Das Fundament ist der Import der verschiedenen Daten aus den Quellsystemen. Es wird wiederum zwischen externen Daten, wie Excel- oder CSV-Dateien, und operativen Vorsystemen wie zum Beispiel Ticketsystemen,[16] sowie Finanz- oder Zeitbuchungssoftware, unterschieden.

Vorhandene Daten werden durch *Extract Transform Load* (ETL) Prozesse in eine Datenbank geladen. ETL-Systeme können Cronjob-Scheduler oder Cloud-Services sein, wie die von Microsoft angebotenen LAs oder die *Data Factory.*[17]

Innerhalb des DWH gibt es nicht nur das ETL-System, sondern auch noch das Metadatenbanksystem, welches im Idealfall der Steuerung des DWH-Betriebs dient, und das Archivierungssystem, was die wichtige Aufgabe der Datensicherung auf sich nimmt.

Neben den verschiedenen Arbeitssystemen gibt es im Grunde zwei verschiedene Datenbanken. Zum einen die Hauptdatenbank des DWH, welche dem Laden, Aufbereiten und Speichern, sowie der Ausgabe der aufbereiteten Daten dient. Zum anderen existieren die für die Weiterverarbeitung durch andere Tools gebildeten Datenextrakte, die sogenannten *Data Marts*. Meist werden diese dann auch auf physischer Ebene vom eigentlichen Datenspeicher getrennt. Trotz ihrer Bedeutung sind sie nicht Pflicht in einem DWH und können je nach Größe der Tabellen weggelassen werden,[18] da es einfachen Sichten möglich ist die Aufgabenbereiche der

[16] vgl. [Glu19]
[17] vgl. [Mic18c]
[18] vgl. [Glu19]

Data Marts abzudecken. Auch besteht die Möglichkeit, bei geringer Größe des DWH, Daten direkt aus der Speicherungsschicht zu nehmen.

Die gesamten DWH-Datenbanken können noch einmal, meist durch Sichten, Tabellen, Prozeduren und Schemas genauer aufgeteilt werden. Meist bilden Schemas die unterschiedlichen Schichten der Datenbank ab und gruppieren alle Datenbankobjekte.

Der Import von Daten endet immer in der *Staging Area*. Dort werden die einzelnen Fakten roh und so wie sie aus den Quellsystemen kommen hin exportiert. Entweder werden die Daten sofort verarbeitet oder vorher noch einmal zwischengespeichert, um dann als *Bulk* (Sammlung mehrerer gleich strukturierter Datensätze, die als eine Masse verarbeitet werden) im *Integration Layer* verarbeitet zu werden. Diese Schicht dient zur Harmonisierung und Homogenisierung der Daten. Dort werden somit „Sammel- und Integrationsfunktionen“[19] ausgeführt, um die Grundlagen für spätere Schichten zu legen. Meist wird die sogenannte *Hub-and-Spoke-Architektur* (HaSA) zur Strukturierung angewendet. Dort werden Daten nach Gebieten oder Funktionen in Satelliten gegliedert und mit Hubs, welche auf die Satelliten verweisen, verbunden. Die HaSA kann auch im *Reporting Layer* zur Anwendung kommen oder gegebenenfalls dort von einer Linkstruktur ersetzt werden. Satelliten und Hubs werden mit permanenten Tabellen realisiert, während Links aus der Linkstruktur über *Views* dargestellt werden. Im *Propagation Layer* werden Geschäftslogiken und Geschäftsregeln erstellt und eingeführt. Diese Schicht kann jedoch je nach Anwendungsfall weggelassen werden. Wichtiger ist das *Reporting Layer*, dass die Aufgabe erfüllt, einzelne Daten oder Datengruppen für die Analyse zusammenzuführen.[20] Dort werden Daten entweder direkt aus den Hub- und Satellitentabellen abgegriffen oder sie werden durch Links, Fakten- und Dimensionstabellen in die *Views* des *Reporting Layers* geladen.

[19] Ebenda
[20] vgl. Ebenda

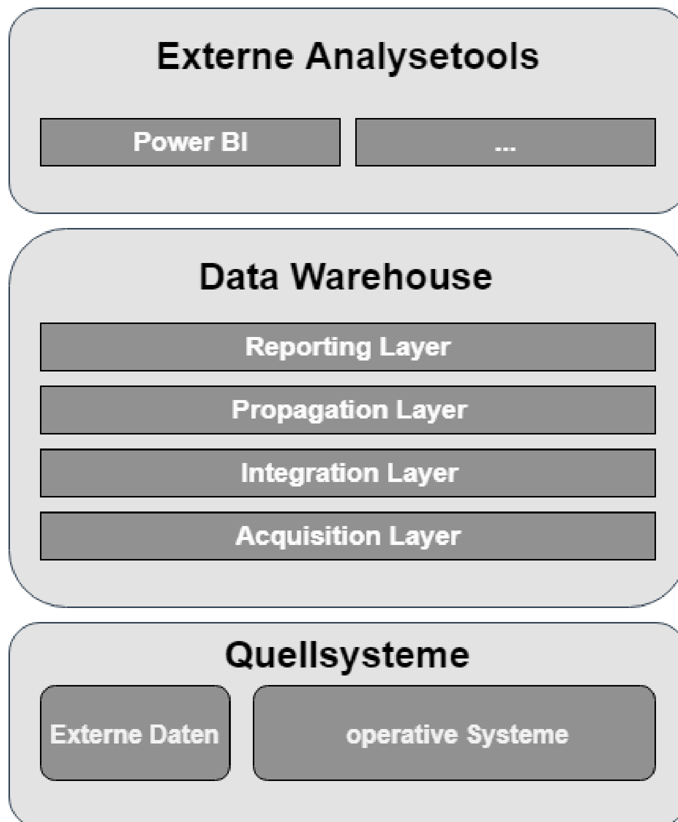


Abbildung 4: DWH-Konstrukt

3.2 Aufbau laut Microsoft Azure

Microsoft setzt keine Richtlinien fest wie der Aufbau innerhalb der Datenbank erfolgen soll. Es gibt jedoch Hilfestellungen, wie ein DWH am effektivsten und je nach Anforderung mit den Ressourcen des hauseigenen Clouddienst *Azure* umgesetzt werden kann.

Die universellste Lösungsidee, hinsichtlich der Nutzung von *Azure* Ressourcen, ist Microsofts „Modernes Data Warehouse“[21], welches in Abbildung 5 dargestellt ist.

[21] [Mic19a]

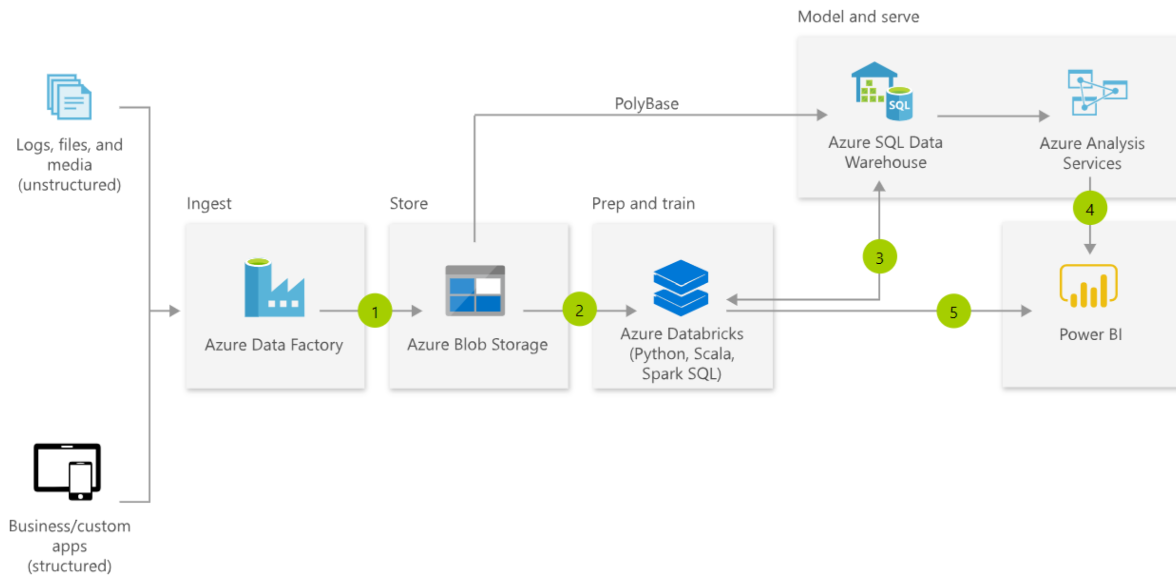


Abbildung 5: Azure Modernes DWH[22]

Zwei Punkte müssen allerdings hervorgehoben werden. Ein DWH mit geringen Datenmengen im Import kann auf den Schritt mit den *Azure Binary large object* (Blob) *Storage* verzichten, da es den Sinn hat einen Bulkimport im SQL DWH zu ermöglichen. Der Bulkimport ist fest in TSQL integriert und von Microsoft bzw. Sybase, mit dem Hintergrund von größtmöglicher Effizienz, optimiert worden. Bei großen Datenmengen ist somit eine erhebliche Zeitersparnis zu erwarten. Es besteht auch die Möglichkeit, alle Importdaten roh abzuspeichern und falls nötig neu zu importieren oder zu Dokumentationszwecken zu historisieren und zu archivieren.

Darüber hinaus ähnelt das *Azure* SQL DWH dem SQL Server. Auch hier empfiehlt sich das Nutzen des SQL DWH nur bei großen Datenmengen, wohingegen bei kleineren Datensätzen der SQL Server zu bevorzugen ist. Ein grundlegender Unterschied stellt die grundsätzliche Optimierung der Server dar. Somit ist der SQL-Server für *Create, Read, Update* und *Delete* Operationen optimiert. Das hauseigene Data Warehouse besticht vor allem durch seine Performance bei Datenanalyseaufgaben und bringt die Möglichkeit zum *Massive Parallel Processing* mit sich.[23]

[22] Ebenda
[23] vgl. [Lui18]

4 Ausgangssituation

4.1 Systemarchitektur des dotSource Data Warehouse

Das Design des DSDWH basiert auf der Konstruktion eines klassischen DWH (siehe Abbildung 6). Es wurde jedoch an die Bedürfnisse des Unternehmens angepasst, da in der dotSource (dS) nicht mehrere Millionen Zeilen an Datensätze gibt. Insbesondere war es nicht die *Data Factory* mit der die ETL-Prozesse des DWH automatisiert wurden, sondern LAs. Darüber hinaus wird nicht der Weg über den *Azure* Blob *Storage* durchlaufen, sondern Rohdaten direkt in die *Azure* SQL-Datenbank importiert. Diese Datenbank befindet sich wiederum auf einem *Azure* SQL-Server und nicht in einem *Azure* SQL DWH, da keine umfangreichen Analysen direkt auf dem Server ausgeführt werden, sondern im ausgelagerten Power BI.

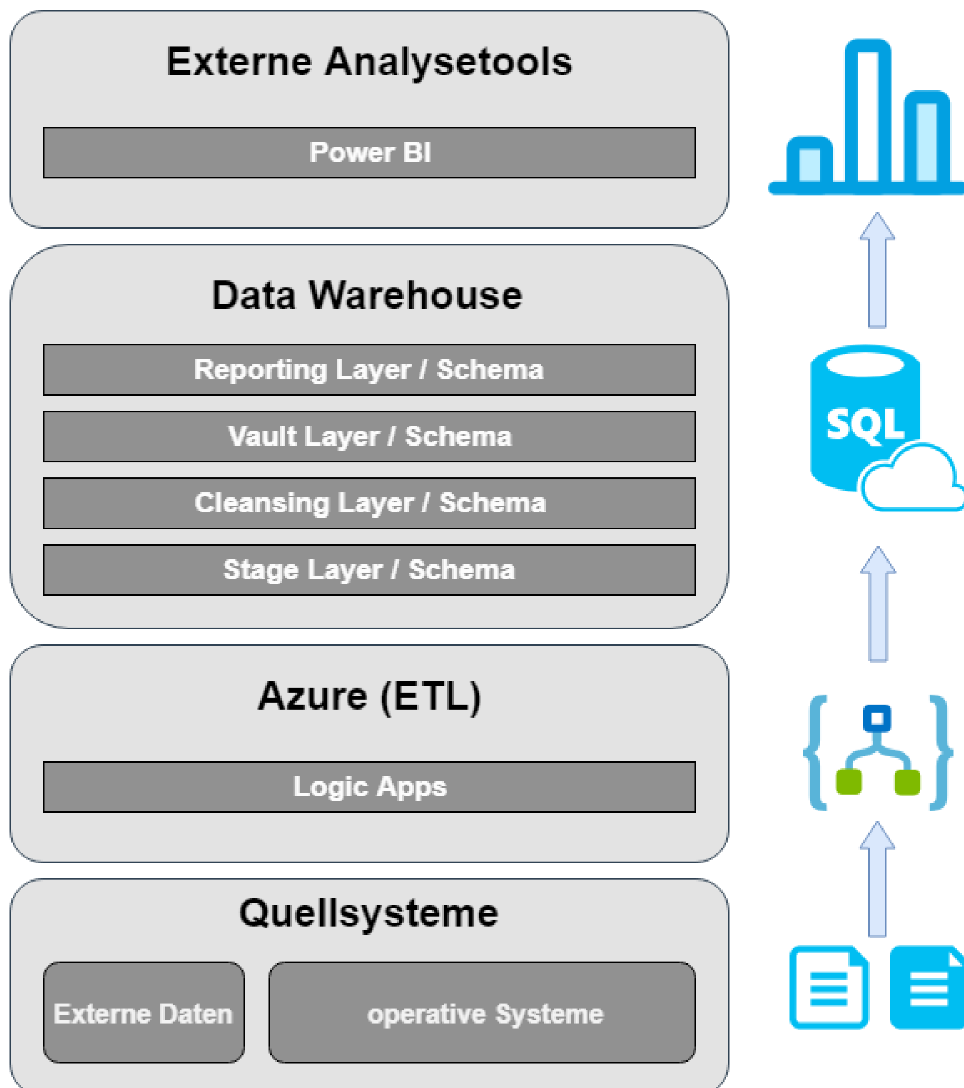


Abbildung 6: Aufbau DSDWH

Eine Besonderheit in der unternehmenseigenen Anpassung ist, dass jede Schicht in der Datenbank nicht, wie oft üblich, physikalisch sondern nur durch Schemas getrennt.

Als Einstiegspunkt für die unbearbeiteten Rohdaten gibt es die *Stage*-Ebene. Die dahinterstehenden Tabellen werden mit verschiedenen Prozeduren beladen. Sollte es bei der weiteren Verarbeitung der Daten zu Fehlern kommen werden diese Tabellen nicht gelöscht, sondern dienen als Zwischenspeicher bis sich das Problem entweder von selbst behoben hat oder es durch einen Entwickler korrigiert wurde.

Die *Cleansing*-Ebene im DSDWH besteht aus keinen permanenten Tabellen, sondern nur aus verschiedenen Sichten (engl. *View*), die sich aus den einzelnen Tabellen in der *Stage* bedienen. Dort werden die Datensätze auf ein einheitliches Bild angepasst. Es korrigiert beispielsweise das Datum, konvertiert Buchungszeiten zu Stunden, korrigiert Nullwerte bzw. begrenzt Dezimalzahlen auf zwei Nachkommastellen. Diese Sichten werden durch Prozeduren aufgerufen. Erfolgreich bereinigte Daten werden dann in die permanenten Tabellen des *Vault Layers* gespeichert.

Dort werden die einzelnen logisch zusammenhängenden Datensätze erneut unterteilt. In Anlehnung an HaSA werden die Daten in einzelne logisch verbundene Tabellengruppen, sogenannte Satelliten, unterteilt, die alle über einen Hubtabelle verbunden sind. Am Beispiel der Buchungstabelle (siehe Abbildung 7) lässt sich diese Struktur besonders gut veranschaulichen.

Die hier aufgeführten *Efforts* sind einzelne Buchungen aus dem Buchungssystem der dS, dem BCS. Der Hub enthält nur die Fakten zum Generieren des 32-stelligen Hashwertes, dem Secure Identifier (SID), als auch das Datum und ob der *Effort* überhaupt noch im BCS existiert oder gelöscht wurde. Mit Hilfe des *Primary Key* (PK) wird auf die Satelliten verlinkt. Dort wurden die einzelnen Daten einer Buchung in logisch zusammenhängende Blöcke unterteilt. Es gibt also eine Tabelle, in der alle Daten zu den Kosten bzw. Einnahmen stehen. In einer anderen stehen zwei weitere SIDs, welche auf andere Tabellen verweisen können. Dort kann in Erfahrung gebracht werden, worauf gebucht wurde und, falls vorhanden, wie das dazu passende Ticket heißt. Die Ticket- oder Projekttabellen zeigen ebenfalls die typische Hub-Satelliten-Struktur auf.

Anhang 2 bietet eine komplette Übersicht der Tabellen und der Schemas vor der Dateverweiterung.

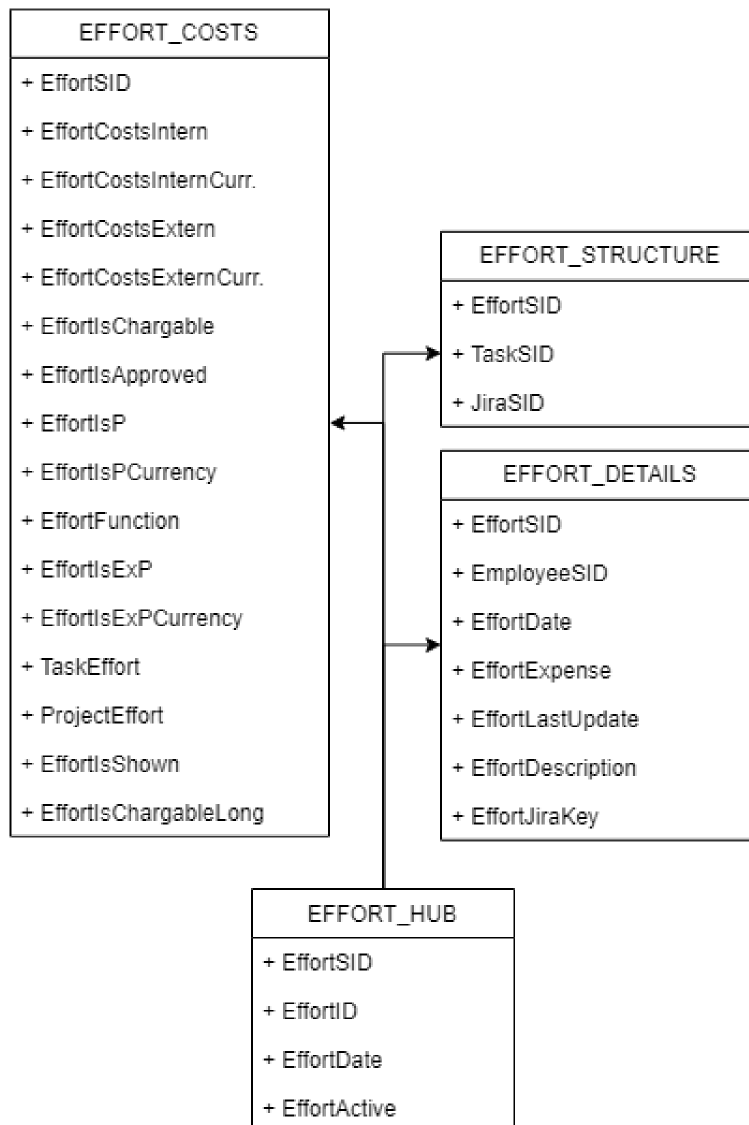


Abbildung 7: Beispiel Buchungstabellen

Über dem *Vault* befindet sich das *Reporting Layer*. Dort existieren ebenfalls keine permanenten Tabellen, sondern nur Dimensions- und Faktensichten. Diese werden für jeden Nutzer einzeln von Power BI abgerufen und geladen.

4.2 Bestehende Quellsysteme

Es gibt drei vorhandene Quellsysteme. Zum einen werden Projekt-, Buchungs-, Angestellten- und Rechnungsdaten aus dem BCS geladen. Auf der anderen Seite werden Projekt- und Ticketdaten aus dem Jira gezogen. Als letzte Quelle für Datensätze fungieren mehrere CSV-

Dateien, mit deren Hilfe BCS-Projekte mit Jira-Projekten verknüpft werden, um eine gesunde Basis für die spätere Auswertung zu erhalten. Die CSV-Dateien befinden sich im SharePoint von Microsoft und sind so für die Projektmanager der dS zugänglich.

5 Anforderungen

Die Erweiterung des DWH um eine Datevschnittstelle stellt einen von drei Teilen in einem größeren Erweiterungsprozess dar. So wurde in mehreren Gesprächen mit dem hauseigenen *Datascientist* und der Verwaltung geklärt, welche Daten abseits der bereits vorhandenen noch benötigt werden (siehe Tabelle 1).

Es wurde festgestellt, dass nicht nur der *BCS-Webservice* erweitert werden muss, sondern auch noch vorhandene Exceldateien in das DWH zusammen mit verschiedenen Datevdaten eingegliedert werden müssen.

BCS-Anforderungen	**DATEV-Anforderungen**	**weitere Anforderungen (meistens als Excelimport)**
Abrechnungsfaktor	Kostenstellen	Arbeitstage und Feiertage
Soll-Arbeitszeit	Datum	Verknüpfung von Kostenstellen und Projekten (Mappingtabelle)
Ist-Arbeitszeit	Leistungsdatum	Arbeitslohn
Abrechnungsfähige Zeit von Ist		
Elternzeit		

Tabelle 1: Anforderungen, die vom *Datascientist* bestimmt wurden

Im Zuge dieser Arbeit soll es dS-Analysten ermöglicht werden alle nötigen Datevdaten zur Verfügung gestellt zu bekommen, um mit Hilfe des Power BI z.B. *Forecasts* erstellen zu können.

6 Umsetzung

6.1 Laden der Daten unter Verwendung der REST API und Azure

6.1.1 Grundaufbau DATEVconnect in der dotSource

Da es sich bei DATEVconnect um eine RESTful API handelt, die *on top* der Standard Datevsoftware konstruiert wurde, werden Abfragen an die DATEV Datenbank mit Hilfe von *Hypertext Transfer Protocol Requests* ausgeführt. Insbesondere werden sogenannte *GET-Requests* verwendet, mit denen nur Daten ausgelesen und nicht verändert oder hinzugefügt werden können. Die genaue Struktur eines solchen Requests ist in der DATEVconnect Dokumentation definiert. Dabei muss ein genaues Schema befolgt werden:

```
<scheme>://<ip-addresse>:<port>/datev/api/<domain>/<domain-
version>/<resource-path>
```

Das *<scheme>* stellt dabei das Verbindungsprotokoll dar. Je nachdem ob die Datevsoftware *local* angesprochen oder ob ein *Request* zu einem Server geschickt wird muss das *Hypertext Transfer Protocol* (HTTP) oder HTTP *Secure* (HTTPS) verwendet werden. Innerhalb der dotSource befindet sich die Schnittstelle auf einem *Laptop* installiert. Was dazu führt, dass HTTPS zur Anwendung kommt und nicht HTTP zusammen mit dem *localhost*.

Für die Verbindung zur Schnittstellt ist auch noch die IP-Adresse und der *Port* des *Laptops* von Nöten. Diese lauten: *jds2271:58452*.

Durch IP-Adresse, Port und Transportprotokoll kann nun die Grunddomain für DATEVconnect erstellt werden.

```
https://jds2271:58452/datev/api/
```

Nun muss nur noch die *<domain>* und *<domain-version>* bestimmt werden. Diese kann z.B. *master-data/v1* lauten, wo *master-data* unter anderem durch *accounting* ersetzt werden kann. Was einen abschließenden Link ergibt.

```
https://jds2271:58452/datev/api/master-date/v1/
```

Sollte nun über diesen Link ein *Request* gestellt werden, muss ebenso noch eine Authentifizierung erfolgen. Bei der dS erfolgt die Freischaltung mit Hilfe der *Windows-Authentication* und der firmenintern genutzten *Domain.* So wurde einerseits innerhalb der DATEV Verwaltung ein *User* für die Schnittstelle angelegt als auch ein Nutzer innerhalb der dS Domain.

Kommen Anfragen allerdings von außerhalb des dS Netzwerks, gibt es zwei mögliche Verfahren, um die Authentifizierung zu ermöglichen. Es können unter Verwendung eines *Proxys*, welcher sich im IP-Verbund der dS befindet, *Requests* auf den *Proxy* zur Schnittstelle umgeleitet werden. Oder es wird der Authentifizierungsprozess mit Hilfe der *Basic Authentication*, welche nur einen *Username* und ein Passwort verlangt, konfiguriert. Allerdings stellt die *Basic Authentication* ein höheres Sicherheitsrisiko als die Verwendung eines *Proxys* dar.

6.1.2 Requests zum Laden der Anforderungen

Bei den folgenden Anfragen wird der Teil des Links, welcher in Kapitel 6.1.1 definiert wurde, weggelassen, da er sich nicht verändert.

Innerhalb von DATEVconnect ist ein fester Aufbau der *Requests* vorgeschrieben. So steht als Wurzelelement der Klient der Firma, von welchem einzelne Objekte innerhalb eines Fiskaljahres, wie z.B. Inventurdaten, Sachkonten und Kreditoren, abrufen werden können. Um alle Kostenstellen mit dem dazugehörigen Datum und Leistungsdatum ausgeben zu lassen, werden alle IDs der Klienten benötigt. Dabei wird bei allen Abfragen immer die gleiche Struktur verwendet.

```
<object>/[<objectid>]
```

Entweder kann z.B. durch das Ersetzen von *<object>* mit *clients* eine Liste von all diesen Objekten als *Response* zurückkommen. Oder es wird mit Hilfe der ID ein bestimmtes Datenobjekt herausgefiltert.

Zur Ermittlung aller Kostenstellen müssen zunächst alle Kunden abgerufen werden.

```
clients/
```

Danach wird von jedem einzelnen Klienten die zu ihm passenden Fiskaljahre ausgelesen.

```
clients/<client-id>/fiscal-years/
```

So wird unter jedem Kunden eine Liste von Fiskaljahr-IDs ausgegeben, welche dazu genutzt wird, Fehlern vorzubeugen, die dadurch entstehen, dass Request z.B. mit der ID eines Fiskaljahres an die Schnittstelle geschickt werden, welche gar nicht für den Kunden existiert. Mit den Fiskaljahr-IDs ist es nun möglich die verschiedenen KOSTS-Systeme eines Jahres von einem bestimmten Kunden abzurufen.

```
clients/<client-id>/fiscal-years/<fiscal-year-id>/cost-
systems/
```

Die KOSTS-Systeme stellen die letzte Instanz vor den eigentlichen Kostenstellen da. Durch Verwendung ihrer IDs kann nun eine Liste von Kostenstellen abgerufen und ihre ID für die genauen Eigenschaften der Kostenstelle genutzt werden.

```
clients/<client-id>/fiscal-years/<fiscal-year-id>/cost-
systems/<cost-system-id>/cost-centers/
```

```
clients/<client-id>/fiscal-years/<fiscal-year-id>/cost-
systems/<cost-system-id>/cost-centers/<cost-centers-id>/
```

Nach jedem *Request* wird immer ein *Response* mit einem *Array* von Elementen zurückgegeben. Diese sind abhängig von den Abfrage Informationen zum Klienten, Fiskaljahr, KOSTS-System oder zur Kostenstelle. Im unteren Beispiel Response ist ein möglicher Rückgabewert bei der *Request* für eine Kostenstelle zu sehen (Abbildung 8).

```
{
  "id": "1000",
  "properties": [
    {
      "id": "1",
      "characteristic_id": 1
    }
  ],
  "cost_rates": [
    {
      "valid_from": 20161201,
      "valid_to": 20161231,
      "rate": 1234567.12
    }
```

```
  ],
  "creation_date": "2016-05-24T08:10:49.000+02:00",
  "date_last_modification": "2016-05-24T08:10:49.000+02:00",
  "email": "Hans.Mustermann@muster.de",
  "long_name": "Verwaltung/Vertrieb",
  "note": "Das ist eine Notiz",
  "short_name": "Verw./Vertr.",
  "postable_from": "2016-05-31T00:00:00.000+02:00",
  "postable_to": "2016-12-31T00:00:00.000+01:00",
  "reference_value": "km",
  "responsible": "Meier"
}
```


Abbildung 8: Kostenstellen Response

Der nächste Schritt ist die Automatisierung des Datenimports über die *Azure* Cloud mit Hilfe von einer LA.

6.1.3 Automatisierung des Imports durch eine Azure Logic App

Bei der Konstruktion der LA wurde die Methode der *Basic-Authentication* verwendet. Des Weiteren wurde sich explizit für die LA entschieden, da ein *Response* von über einem *Megabyte* bei Kostenstellen nicht unüblich ist. Eine *Data Factory Activity* kann hingegen nur einen Rückgabewert von maximal einem *Megabyte* übergeben. Auch ist es innerhalb einer *Pipeline* nicht möglich verschachtelte Schleifen zu verwenden, weshalb ein *Workaround* mit mehreren *Pipelines* konstruiert werden musste. Zwecks der Lesbarkeit des Automatisierungsdienstes wurde sich somit gegen die *Data Factory* entschieden. Es sei als Nachtrag zu erwähnen, dass die neue *Data Factory* Funktion „*Copy Data*" keine Limitierung im Rückgabewert vorsieht und die *Data Factory* nun als Alternative neu bewertet werden müsste. Dies wurde aber erst nach Fertigstellung dieser Arbeit bekannt und so widmet sich die weitere Ausführung der Implementierung der LA.

Da schon eine *Azure Subscription* und eine *Resource group* vorhanden ist, muss als erster Schritt nur eine LA erstellt werden. Dabei muss das Namensschema und der Standort beachtet werden, wie es in Abbildung 9 zu sehen ist.

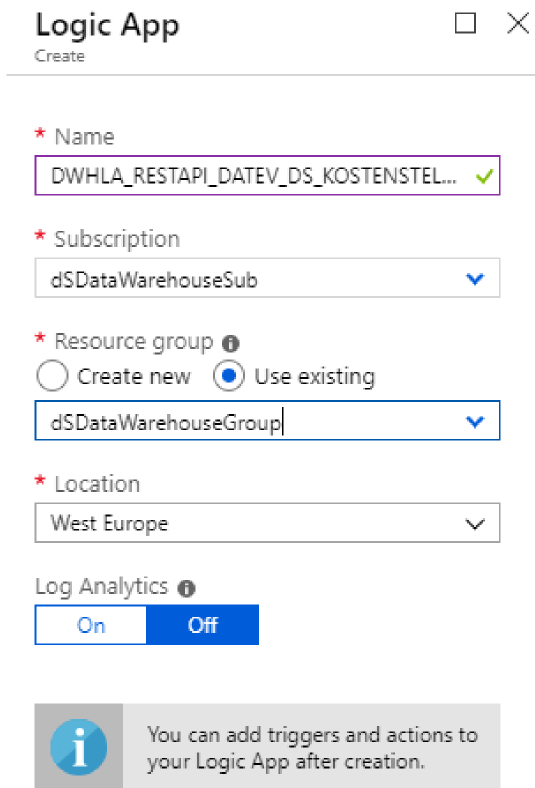


Abbildung 9: Logic App erstellen

Am Anfang einer LA steht immer ein *Trigger,* um die restliche Ausführungslogik einzuleiten und zu starten. In diesem Fall handelt es sich um ein alle sechs Stunden laufenden *Scheduler* mit dem Namen *Recurrence* (siehe Abbildung 10). Der Verständlichkeit halber wird bei den einzelnen Schritten in einer LA ab sofort von Blöcken gesprochen.

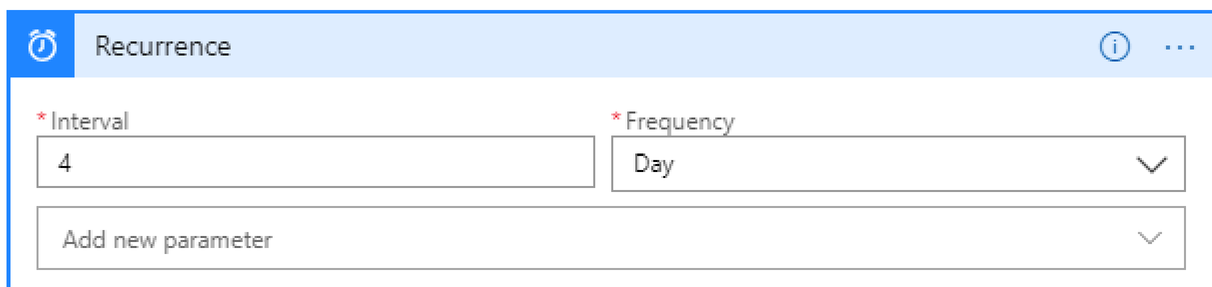


Abbildung 10: Logic App Trigger Recurrence

Zu Anfang der eigentlichen Abarbeitungslogik steht ein RESTAPI *GET-Request* eines HTTP-Blocks. Dort wird die bekannte URL aus Kapitel 6.1.2 eingegeben und die Authentifizierung auf *Basic* mit passendem *Username* und Passwort gestellt (siehe Anhang 3).

Danach folgt die Auswertung der IDs mit Hilfe einer *Forloop* und ein weiterer HTTP-Block, dessen Ergebnisse wieder in einer *Forloop* abgearbeitet werden. Dieses Muster aus Abfragen setzt sich für alle im Kapitel 6.1.2 besprochenen Elemente fort. Sodass eine Reihe aus Schleifen und HTTP-Blöcken entsteht, wie in Anhang 4 zu sehen ist. Damit wird die nach der Schnittstelle definierte Reihenfolge abgearbeitet. Es wäre auch möglich gewesen mit einer eigenen LA nur die Klienten und die meisten Zwischenschritte in eigene Tabellen zu laden und dann mit einer SQL-Abfrage diese zu holen und für weitere LAs als Requestgrundlage zu nutzen.

Allerdings sind die einzelnen *Requests* noch nicht abgesichert. Es kann immer noch der Fall eintreten, dass ein *Response* ein leeres *Array* zurückgibt. Dieser Fall muss nach jeder Anfrage mit einer If-Anweisung abgefangen werden (siehe Anhang 5). Sollte nun ein leeres *Array* zurückkommen, wird entweder die komplette LA terminiert oder das betreffende Element übersprungen und mit dem nächsten fortgefahren.

Ab diesem Punkt dient die restliche Logik der LA nur noch um SQL-Prozeduren und -Abfragen auszuführen. So muss einerseits die Stagetabelle durch eine Prozedur geladen werden. Die nächste Prozedur lädt Daten in die Cleaningview und legt diese dann im *Vault* ab. Dies passiert pro Hub und Satellit einzeln. Danach muss die Stagetabelle gesäubert werden. Dabei wird nicht der *DELETE* Befehl, sondern der *TRUNCATE TABLE* Befehl benutzt, da dieser keine *Logs* speichert und somit schneller läuft (siehe Anhang 6).

6.2 SQL-seitige Verarbeitung der Daten

Alle Tabellen, Sichten und Prozeduren bedienen sich der verschiedenen Schemas des DSDWH. Beispielsweise steht *dwh_vault* für die Vaultschicht, *dwh_cls* für die Cleaningschicht und *dwh_stage* definiert Elemente der Stageschicht. Es gibt außerdem noch *dwh_rpt* als Schema für die *Reports* und der eigentlichen Analysegrundlage für das Power BI.

Für den Import in das DWH wird eine Stagetabelle angelegt (siehe Abbildung 11), in der die Daten roh und unbearbeitet zwischengespeichert werden. Die Bezeichner der einzelnen Spalten orientieren sich innerhalb der Stagetabelle an den Namen aus der Quelle, um Verwechslungen vorzubeugen und das *Refactoring* zu erleichtern.

```
CREATE TABLE dwh_stage.DATEV_COST_CENTER
(id                     VARCHAR(50)  PRIMARY KEY,
 date_last_modification VARCHAR(MAX) NULL,
 creation_date          VARCHAR(MAX) NULL,
 long_name              VARCHAR(MAX) NULL,
 short_name             VARCHAR(MAX) NULL,
 postable_from          VARCHAR(MAX) NULL,
 postable_to            VARCHAR(MAX) NULL,
 rate                   VARCHAR(MAX) NULL,
 reference_value        VARCHAR(MAX) NULL
);
```

Abbildung 11: SQL-Skript erstellen der Stagetabelle

Als nächstes werden die verbleibenden Vaulttabellen und die *Cleaningview* erstellt. Zuerst werden die Vaulttabellen erstellt, um die Felder für die *Cleaningview* zu definieren.

Der Hub beinhaltet nur die erstellte SID und die Elemente, aus denen diese berechnet wird. Dabei handelt es sich um die ID der Kostenstellen (siehe Abbildung 13). Es zeigt sich auch, dass sich die Bezeichnung der einzelnen Spalten geändert hat. Die Namensgebung ergibt sich daher aus dem gemeinsamen Namensteil der Satelliten und des Hubs und einigen durch andere Quellsysteme gewachsenen Bezeichnungen im *CamelCase*.

```
CREATE TABLE dwh_vault.COSTCENTER_DETAILS
(CostcenterSID             CHAR(32)     PRIMARY KEY,
 CostcenterInsDate         DATE         NOT NULL,
 CostcenterLastUpdate      DATE         NOT NULL,
 CostcenterLongName        VARCHAR(MAX) NULL,
 CostcenterShortName       VARCHAR(MAX) NULL,
 CostcenterRate            VARCHAR(MAX) NULL,
 CostcenterRateValue       VARCHAR(max) NULL,
 --Datum
 CostcenterDate            DATE         NULL,
 --Leistungsdatum
 CostcenterPerformanceDate DATE         NULL
);
```

Abbildung 12: SQL-Skript erstellen der Satellitentabelle

```
CREATE TABLE dwh_vault.COSTCENTER_HUB
(CostcenterSID             CHAR(32)     PRIMARY KEY,
 CostcenterID              VARCHAR(MAX) NOT NULL
);
```

Abbildung 13: SQL-Skript erstellen der Hubtabelle

Die Satellitentabelle hat die Aufgabe, die Informationen in einer logischen Einheit zu speichern (siehe Abbildung 12). Jedoch sind es nur wenige Werte, die aus dem DATEV gezogen werden. Deshalb wird einzig und allein eine Tabelle erstellt, welche alle Details beinhalten soll. Bei zukünftigen Erweiterungen werden die Satelliten entweder erweitert oder komplett neue hinzugefügt.

```sql
CREATE VIEW dwh_cls.V_DATEV_COSTCENTER
AS
select
	dwh_utl.fn_GetSID(id, NULL, NULL) as CostcenterSID,
	id as CostcenterID,
	LEFT(date_last_modification, 10) as CostcenterLastUpdate,
	LEFT(creation_date, 10) as CostcenterInsDate,
	long_name as CostcenterLongName,
	short_name as CostcenterShortName,
	LEFT(postable_from, 10) as CostcenterDate,
	LEFT(postable_to, 10) as CostcenterPerformanceDate,
	rate as CostcenterRate,
	reference_value as CostcenterRateValue
from dwh_stage.DATEV_COST_CENTER
```

Abbildung 14: SQL-Skript erstellen der Säuberungssicht

Zwischen den Vaulttabellen und der Stagetabelle steht eine Sicht (siehe Abbildung 14), um die Daten zu vereinheitlichen. Dort werden *Datetime*-Werte zu normalen Datumsangaben konvertiert und aus der Kostenstellenid wird eine SID generiert, welche durchweg im DSDWH verwendet werden. Datevdaten verwenden viele ISO-Regeln, was den Vorteil mit sich bringt, dass sie recht wenig in der Cleaningebene bearbeitet werden müssen. Andere Sichten, wie die *Issueview* im Anhang 7, sind nicht so schlank.

Nachdem nun alle Tabellen und Sichten definiert und auf dem Server erstellt wurden, können auch die dazugehörigen Prozeduren programmiert werden. Diese haben die Aufgabe, die jeweiligen Schichten zu laden. Es werden somit drei verschiedene Prozeduren benötigt. Eine zum Laden der Stagetabelle (Abbildung 15) und zwei weitere für den Hub (Abbildung 16) und den Satelliten. Die Funktion der Prozeduren ist auf den *MERGE INTO* Befehl ausgelegt.

Da aus der LA ein *JavaScript Object Notation* (JSON) *Array* zurückkommt, muss dieses auch an die Prozedur übergeben werden. TSQL bietet mit der *OPENJSON* Funktion eine integrierte Möglichkeit, um JSON effektiv zu verarbeiten und auszulesen. Mit Hilfe des *MERGE INTO* Befehls ist eine schnelle Speicherung der einzelnen Felder in eine Tabelle möglich. Dafür werden die vorher ausgelesenen Werte verwendet. Es muss eine Übereinstimmungsbedingung geben oder ein *Match-Case* vorliegen. Dies kann allerdings auch „eins gleich zwei" (1=2) oder sonst ein Ausdruck sein, der logisch falsch ist, weil die LA das Säubern der Tabelle mit dem *TRUNCATE* Befehl übernimmt und somit keine zutreffende Bedingung nötig ist. In die Stagetabelle werden somit nur Werte eingelesen allerdings nicht aktualisiert, da diese Daten nur nicht gelöscht werden, wenn ein Fehler in einem höheren Verarbeitungsschritt erfolgt ist. Wenn bei der restlichen Verarbeitung ein Problem auftreten sollte, bleiben die Daten in der Stagetabelle erhalten, um feststellen zu können ob der Fehler aufgrund noch nicht abgefangener ungültiger Werten erfolgte.

```sql
CREATE PROCEDURE [dwh_stage].[usp_datev_dotSource_Cost_Center] @json nvarchar(max)
AS
BEGIN
	MERGE INTO dwh_stage.DATEV_COST_CENTER dat USING(
		SELECT
			JSON_VALUE(c.value, '$.id') as id,
			JSON_VALUE(c.value, '$.creation_date') as creation_date,
			JSON_VALUE(c.value, '$.date_last_modification') as
			date_last_modification,
			JSON_VALUE(c.value, '$.long_name') as long_name,
			JSON_VALUE(c.value, '$.short_name') as short_name,
			JSON_VALUE(c.value, '$.postable_from') as postable_from,
			JSON_VALUE(c.value, '$.postable_to') as postable_to,
			JSON_VALUE(c.value, '$.reference_value') as reference_value,
			JSON_VALUE(c.value, '$.cost_rates.rate') as rate
		FROM OPENJSON(@json) as c
	) sub
	ON (sub.id = dat.id)
	WHEN NOT MATCHED THEN INSERT VALUES
		(sub.id,
		 sub.date_last_modification,
		 sub.creation_date,
		 sub.long_name,
		 sub.short_name,
		 sub.postable_from,
		 sub.postable_to,
		 sub.rate,
		 sub.reference_value);
END;
```

Abbildung 15: SQL-Skript erstellen der Prozedure zum Laden der Stage

Die letzten zwei Prozeduren zum Laden der Hub- und Satellitentabelle bedienen sich ebenfalls dem *MERGE INTO Statement*. Als Quelle für den *Merge* dient die vorher erstellte *Cleaningview*. Dieses Mal enthält der *MERGE INTO* Befehl eine *MATCHED* Bedingung, um vorhandene Datensätze zu aktualisieren.

```sql
CREATE PROCEDURE dwh_vault.usp_loadClsCostcenter
AS
BEGIN
	MERGE INTO dwh_vault.COSTCENTER_HUB dat USING(
		SELECT
			CostcenterSID,
			CostcenterID
		FROM dwh_cls.V_DATEV_COSTCENTER
	) sub
	ON (sub.CostcenterSID = dat.CostcenterSID)
	WHEN NOT MATCHED THEN INSERT VALUES
		(sub.CostcenterSID,
		 sub.CostcenterID)

	WHEN MATCHED THEN UPDATE SET
		dat.CostcenterSID = sub.CostcenterSID,
		dat.CostcenterID = sub.CostcenterID;
END;
```

Abbildung 16: SQL-Skript erstellen der Prozedur zum Laden des Hubs

Es sollte beachtet werden, dass zwar bei der Hubtabelle ein *MATCHED* Zweig existiert, dieser allerdings auch weggelassen werden kann, da sich weder SID oder ID jemals ändern können. Dieser Zweig existiert also mehr aus einheitlichen Gründen. So gibt es auch Hubtabellen (z.B. *EFFORT_HUB*), wo sich Werte verändern können und dementsprechend aktualisiert werden. Die *MATECHED*-Bedingung ist auch vorhanden, da zukünftige Erweiterungen sie unter Umständen mal benötigen könnten. Es könnte eine neue Spalte geben, in die geschrieben wird ob diese Kostenstelle noch existiert oder nicht, auch wenn es eher unwahrscheinlich ist ob Kostenstellen jemals gelöscht werden.

```
CREATE PROCEDURE dwh_vault.usp_loadClsCostcenter_DETAILS
AS
BEGIN
	MERGE INTO dwh_vault.COSTCENTER_DETAILS dat USING(
		SELECT
			CostcenterSID,
			CostcenterLastUpdate,
			CostcenterInsDate,
			CostcenterLongName,
			CostcenterShortName,
			CostCenterDate,
			CostCenterPerformanceDate,
			CostcenterRate,
			CostcenterRateValue
		FROM dwh_cls.V_DATEV_COSTCENTER
	) sub
	ON (sub.CostcenterSID = dat.CostcenterSID)
	WHEN NOT MATCHED THEN INSERT VALUES
		(sub.CostcenterSID,
		 sub.CostcenterInsDate,
		 sub.CostcenterLastUpdate,
		 sub.CostcenterLongName,
		 sub.CostcenterShortName,
		 sub.CostcenterRate,
		 sub.CostcenterRateValue,
		 sub.CostCenterDate,
		 sub.CostCenterPerformanceDate)

	WHEN MATCHED THEN UPDATE SET
		dat.CostcenterSID = sub.CostcenterSID,
		dat.CostcenterInsDate = sub.CostcenterInsDate,
		dat.CostcenterLastUpdate = sub.CostcenterLastUpdate,
		dat.CostcenterLongName = sub.CostcenterLongName,
		dat.CostcenterShortName = sub.CostcenterShortName,
		dat.CostcenterRate = sub.CostcenterRate,
		dat.CostcenterRateValue = sub.CostcenterRateValue,
		dat.CostCenterDate = sub.CostCenterDate,
		dat.CostCenterPerformanceDate = sub.CostCenterPerformanceDate;
END;
```

Abbildung 17: SQL-Skript erstellen der Prozedure zum Laden der Detaitabelle

Beim Laden der Detailtabelle (siehe Abbildung 17) ist der *Update*-Zweig von größerer Bedeutung. Es kann immer Situationen geben, in denen sich Werte von einem *Load* zum anderen verändern und überarbeitet werden müssen.

6.3 Reporting Layer bevölkern

Wenn alle Daten in die Vaulttabellen geladen wurden, gilt der *Import* als abgeschlossen. Diese Daten gelten danach als verlässliches Spiegelbild der Quelldaten und werden für Analyseaufgaben verwendet. Der letzte Schritt, um sie verwenden zu können, stellt dabei der Aufbau von Fakten- und Dimensionssichten dar, wobei es sich hier um eine Faktensicht handeln würde. Am Anfang des DSDWH Projektes lag die Aufgabe zum Bau dieser *Reportingviews* bei den Entwicklern. Die Zuständigkeit wurde allerdings nun auf die firmeninternen *Data Scientists* übertragen. Entwickler übernehmen nur noch eine Hilfestellung, falls keine soliden SQL-Kenntnisse bei den Analysten vorhanden sind.

7 Probleme während der Umsetzung

Es gab keine Probleme bei der Tiefe der Datevdokumentation oder bei der Erfahrung im Umgang mit *Azure* und SQL-Servern. Die Probleme traten eher bei der technischen Einrichtung der Schnittstelle auf. Die DATEVconnect-Schnittstelle ist um den Verzeichnisdienst *Active Directory* von Microsoft aufgebaut. Innerhalb der dS wird allerdings eine eigene *Domain* verwendet und nicht die *Active Directory Domain Services*. Dies führt zu Authentifizierungsproblemen mit der Schnittstelle. Da ein Benutzer auf dem Datevserver erstellt werden muss und dieser sich mit seinem *Domain Username* und Passwort anmelden bzw. seine Anmeldedaten bei einem *Request* aus Kapitel 6.1.1 mit übertragen muss. Da dieser jedoch nicht von einer *Active Domain* verwaltet wird, sondern von einer hauseigenen, kommt es zu Authentifizierungsfehlern. Der Nutzer wird somit nicht erkannt und es kann keine Abfrage stattfinden.

Die Schnittstelle lief auch auf einem Rechner, wo die nötige Software schon installiert war und von einem Finanzmitarbeiter genutzt wurde. Dieser Computer musste permanent eingeschaltet und erreichbar sein. Es wurde sich für diese Lösung entschieden, weil die benötigte Datevsoftware normalerweise nicht auf einem Server installiert ist. Auf lange Sicht sollte allerdings ein Server bereitgestellt werden, da dieser auch nach außen hin geöffnet werden muss, um einen Zugriff durch *Azure* zu ermöglichen. Es besteht aber auch die Möglichkeit die *Azure* Cloud durch ein *Virtual Private Network* (VPN) anzubinden und so die Abfragen umzuleiten.

Diese Probleme hatten zur Folge, dass *Responses* gemockt werden mussten. Dies war jedoch aufgrund der ausführlichen Dokumentation mit Beispiel *Requests* und *Responses* und dem Zugang zu den eigentlichen dS Daten leicht umzusetzen. Außerdem bieten die LAs direkt bei jedem Block die Möglichkeit ein Mockergebnis festzulegen, um den Ablauf testen zu können. DATEV direkt bietet eine virtuelle und lokale Testumgebung, in die Daten der dS eingelesen werden konnten, um dann die Abfragen dagegen zu testen.

Als Nachtrag bleibt zu erwähnen, dass nach Besprechung der Erkenntnisse dieser Arbeit mit der IT unter Zuhilfenahme des DATEV-Supports ein Server mit benötigter Software eingerichtet wurden ist. Gegen diesen Server konnten erneut die *Requests* aus Kapitel 6.1.2 und die Funktion der LA geprüft werden. Beide Tests ergaben die volle Funktionstüchtigkeit der in der Arbeit beschriebenen Module.

8 Fazit

Die in dieser Arbeit erstellte Schnittstelle ermöglicht eine Ausweitung der Analysegrundlagen von *Projektmanagern*, *Teamleads*, Verwaltungspersonal und *Data Scientists*.

Es sollten jedoch für die zukünftige Verwendung alle Probleme, wie in Kapitel 7 beschrieben, beseitigt werden, um einen richtigen Nutzen aus der Schnittstelle ziehen zu können. Da sie momentan nicht in Gebrauch ist und somit weiterhin auf den *Import* von Exceldateien zurückgegriffen werden muss.

Trotzdem konnte man wichtige Erkenntnisse gewinnen wie das eine Umstellung auf eine *Active Domain* und ein VPN nötig ist. Beide Voraussetzungen für eine erfolgreiche Nutzung der Schnittstelle sind fest eingeplant und in der Strategieplanung der dS enthalten und sollen zukünftig umgesetzt werden. So wird die *Azure* im Zugang der Anbindung von externen Jiras mit einem VPN versorgt. Des Weiteren soll bis spätestens im Sommer des Jahres 2019 eine Umstellung auf die *Active Domain* von Microsoft erfolgen. Vor allem letzteres ist ein großes Unterfangen und muss zeitnah angegangen werden. Bei einer falschen Umstellung kann es unter Anderem zur Zugangssperrung der Computer eines jeden Mitarbeiters führen.

Eine Implementierung eines Servers mit einer *Basic Authentification* könnte auch den Zugriff auf die Daten ermöglichen und so zusätzliche Kosten durch einen VPN Dienst von Microsoft verringern. Des Weiteren bietet ein *dedicated* Server die Möglichkeit auf DATEV-Daten vor Umstellung auf die *Active Domain* zuzugreifen.

Künftig wäre auch eine Umstellung auf die *Data Factory* möglich. Danach würde es nur noch zur Erweiterung des ETL-Prozesses kommen. So steht momentan schon fest, dass künftig auch Gehaltsabrechnungen, Löhne, Haben-Soll- und Haben-Ist-Werte von Projekten und *Teams* ausgelesen werden sollen.

Literaturverzeichnis

[DAT18] DATEV: „DATEVconnect“, 2018. https://www.datev.de/web/de/top-themen/unternehmer/weitere-themen/neue-schnittstellentechnologie/datevconnect/ Abruf: 2019.04.02

[Glu19] Gluchowski Peter: „Data Warehouse“, 2019. http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/daten-wissen/Business-Intelligence/Data-Warehouse Abruf: 2019.04.01

[Lui18] Luijbregts Barry: „Compare Azure SQL Database vs. Azure SQL Data Warehouse: Definitions, Differences and When to Use“, 2018. https://stackify.com/azure-sql-database-vs-warehouse/ Abruf: 2019.04.02

[Mic18a] Microsoft: „Arbeiten mit Aggregaten (Summe, Mittelwert usw.) im Power BI-Dienst - Power BI“, 2018. https://docs.microsoft.com/de-de/power-bi/service-aggregates Abruf: 2019.04.02

[Mic18b] Microsoft: „Azure Functions – serverlose Architektur | Microsoft Azure“, 2018. https://azure.microsoft.com/de-de/services/functions/ Abruf: 2019.04.02

[Mic18c] Microsoft: „Extrahieren, Transformieren und Laden (ETL)“, 2018. https://docs.microsoft.com/de-de/azure/architecture/data-guide/relational-data/etl Abruf: 2019.04.02

[Mic18d] Microsoft: „In Azure Functions unterstützte Sprachen“, 2018. https://docs.microsoft.com/de-de/azure/azure-functions/supported-languages Abruf: 2019.04.02

[Mic19a] Microsoft: „Architektur moderner Data Warehouses | Microsoft Azure“, 2019. https://azure.microsoft.com/de-de/solutions/architecture/modern-data-warehouse/ Abruf: 2019.04.02

[Mic19b] Microsoft: „Azure Databricks | Microsoft Azure“, 2019. https://azure.microsoft.com/en-us/services/databricks/ Abruf: 2019.04.02

[Mic19c] Microsoft: „Power BI | Interaktive BI-Tools für die Datenvisualisierung“, 2019. https://powerbi.microsoft.com/de-de/ Abruf: 2019.04.02

[Mic19d] Microsoft: „SQL-Datenbank – Database-as-a-Service-Lösung für die Cloud | Microsoft Azure“, 2019. https://azure.microsoft.com/de-de/services/sql-database/ Abruf: 2019.04.02

[Mic19e] Microsoft: „Was ist PaaS? Platform-as-a-Service | Microsoft Azure“, 2019. https://azure.microsoft.com/de-de/overview/what-is-paas/ Abruf: 2019.04.01

[Mic19f] Microsoft: „Was ist SaaS? Software-as-a-Service | Microsoft Azure“, 2019. https://azure.microsoft.com/de-de/overview/what-is-saas/ Abruf: 2019.04.01

[Sef18] Seferlis, C.: „Azure Data Factory vs Logic Apps“, 2018. http://blog.pragmaticworks.com/azure-data-factory-vs-logic-apps Abruf: 2019.04.02

[Wei16] Weigel Frank: „Announcing Azure Logic Apps general availability“, 2016. https://azure.microsoft.com/en-us/blog/announcing-azure-logic-apps-general-availability/ Abruf: 2019.04.01

Anhangsverzeichnis

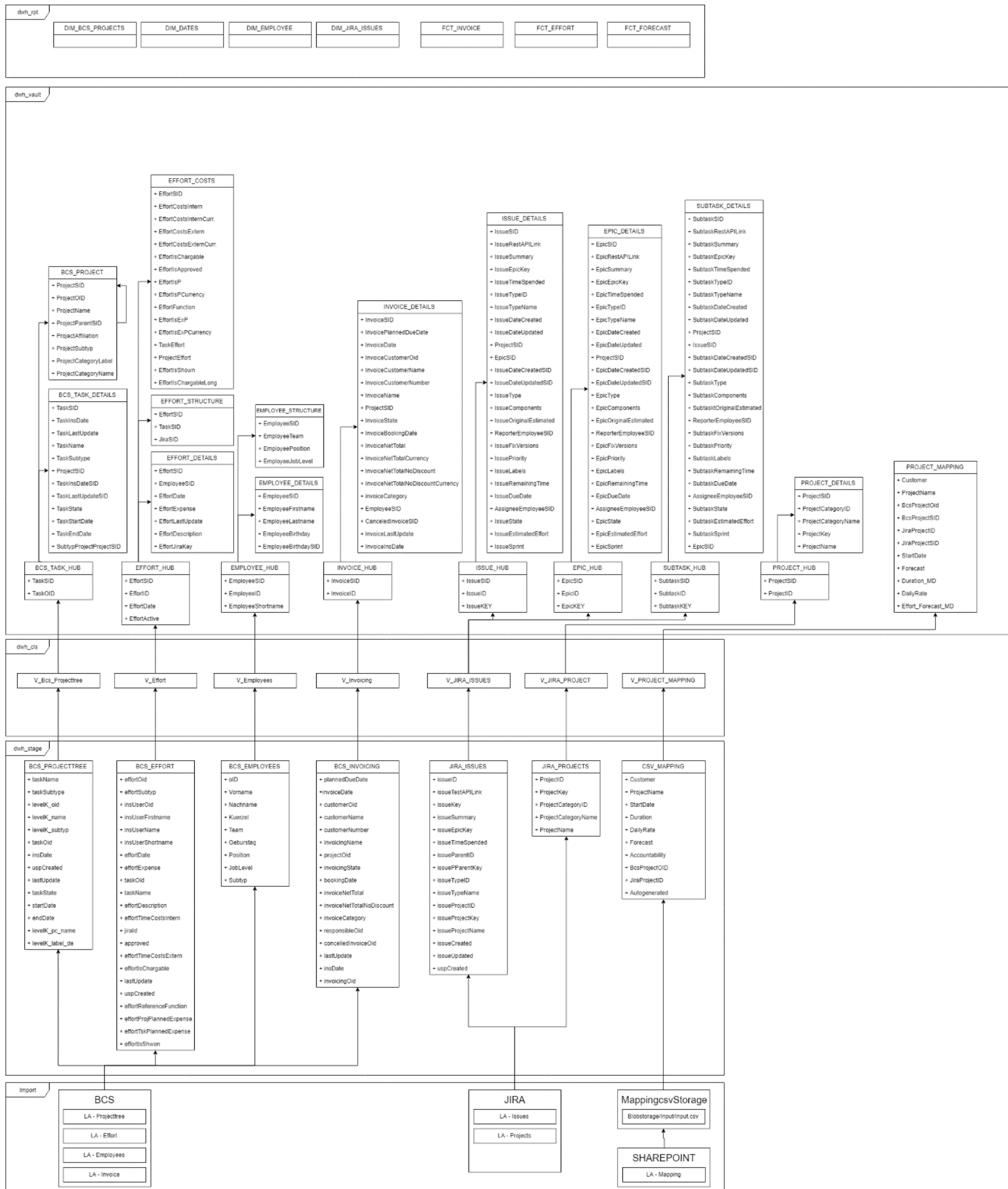
Anhang 1: Beispielhafter Aufbau einer Logic App VIII
Anhang 2: Struktur Tabellen und Sourcesysteme IX
Anhang 3: HTTP-Block X
Anhang 4: Requestschleifen X
Anhang 5: If-Anweisung XI
Anhang 6: Prozeduren XI
Anhang 7: Issueview (Jira) XII

Anhänge

Anhang 1: Beispielhafter Aufbau einer Logic App[24]

[24] [Wei16]

Anhang 2: Struktur Tabellen und Sourcesysteme

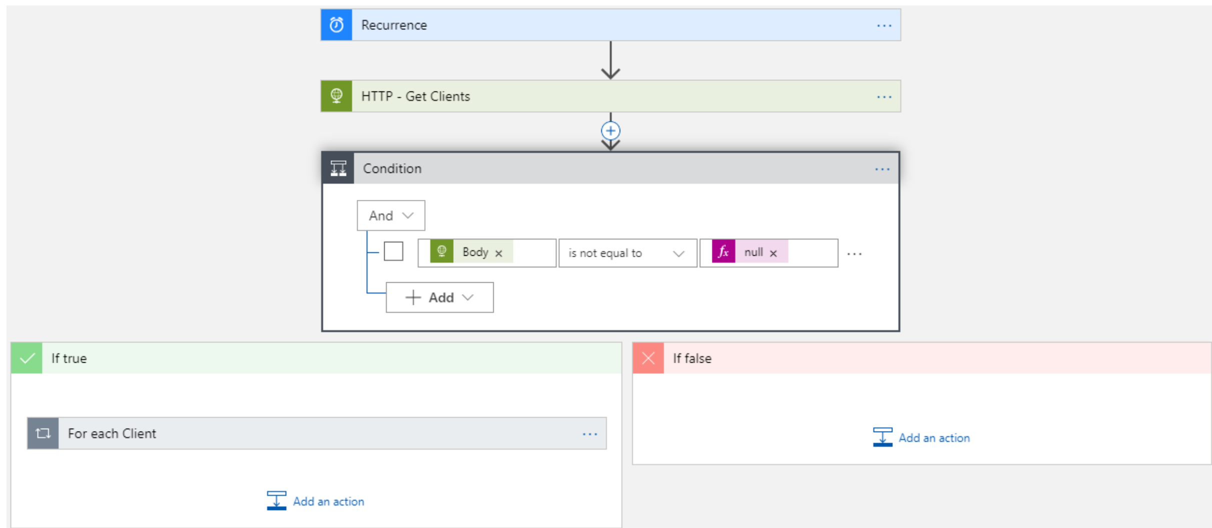


Anhang 3: HTTP-Block

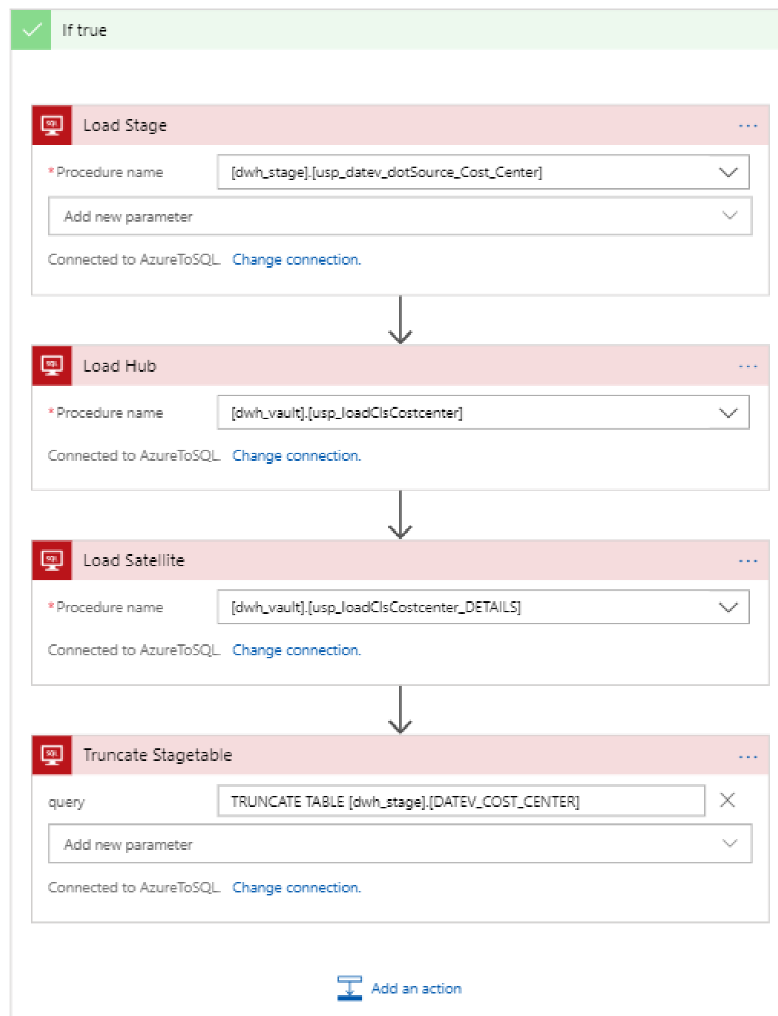
{entfernt wegen Internas}

Anhang 4: Requestschleifen

Anhang 5: If-Anweisung



Anhang 6: Prozeduren



Anhang 7: Issueview (Jira)

```
CREATE VIEW [dwh_cls].[V_JIRA_ISSUES]
AS
     WITH issues
          AS (SELECT [dwh_utl].[fn_GetSID](JIRA_ISSUES.issueID, NULL, NULL)
IssueSID,
                     JIRA_ISSUES.issueID AS issueID,
                     JIRA_ISSUES.issueRestAPILink AS RestAPILink,
                     JIRA_ISSUES.issueKey AS 'issueKey',
                     JIRA_ISSUES.issueSummary AS Summary,
                     JIRA_ISSUES.issueEpicKey AS EpicKey,
                     JIRA_ISSUES.issueTimeSpended AS TimeSpended,
                     JIRA_ISSUES.issueParentID AS ParentID,
                     JIRA_ISSUES.issueParentKey AS ParentKey,
                     JIRA_ISSUES.issueTypeID AS TypeID,
                     CASE
                         WHEN JIRA_ISSUES.issueTypeName = 'Epos'
                         THEN 'Epic'
                         ELSE JIRA_ISSUES.issueTypeName
                     END TypeName,
                     JIRA_ISSUES.issueProjectID AS ProjectID,
                     JIRA_ISSUES.issueProjectKey AS ProjectKey,
                     JIRA_ISSUES.issueProjectName AS ProjectName,
                     LEFT(JIRA_ISSUES.issueCreated, 10) AS DateCreated,
                     LEFT(JIRA_ISSUES.issueUpdated, 10) AS DateUpdated,
                     issueType,
                     issueComponents,
                     CONVERT(decimal(38,2), (CONVERT(decimal(38,4),
issueOriginalEstimated)/60/60)) as issueOriginalEstimated,
              (
                  SELECT max(EmployeeSID) EmployeeSID
                  FROM dwh_vault.EMPLOYEE_HUB
                  WHERE EmployeeShortname = issueReporterShortname
                          and employeeid not like 'ds%'
              ) AS ReporterEmployeeSID,
                     issueFixVersions,
                     issuePriority,
                     issueLabels,
                     CONVERT(decimal(38,2), (CONVERT(decimal(38,4),
issueRemainingTime)/60/60)) as issueRemainingTime,
                     issueDueDate,
              (
                  SELECT max(EmployeeSID) EmployeeSID
                  FROM dwh_vault.EMPLOYEE_HUB
                  WHERE EmployeeShortname = issueAssigneeShortname
                          and employeeid not like 'ds%'
              ) AS AssigneeEmployeeSID,
                     issueState,
                     issueEstimatedEffort,
                     issueSprint
              FROM dwh_stage.JIRA_ISSUES),
          epic2subtask
          AS (SELECT a.issueID,
                     a.ParentID,
                     a.ParentKey,
                     CASE
                         WHEN b.EpicKey IS NULL
                         THEN det.IssueEpicKey
                         ELSE b.EpicKey
                     END AS EpicKey,
                     det.IssueEpicKey
```

```
                FROM issues a
                     LEFT JOIN issues b ON a.ParentID = b.issueID
                     INNER JOIN dwh_vault.ISSUE_HUB iss ON a.ParentID = iss.IssueID
                     INNER JOIN dwh_vault.ISSUE_DETAILS det ON iss.IssueSID =
det.IssueSID
                WHERE a.TypeName = 'Sub-task')
            SELECT issues.IssueSID,
                   issues.issueID,
                   issues.RestAPILink,
                   issues.issueKey,
                   issues.Summary,
                   CASE
                       WHEN issues.TypeName = 'Sub-task'
                       THEN epic2subtask.EpicKey
                       WHEN issues.TypeName = 'Epic'
                       THEN issues.issueKey
                       ELSE ISNULL(issues.EpicKey, issues.ProjectKey+'-0')
                   END EpicKey,
                   issues.TimeSpended,
                   issues.ParentID,
                   issues.ParentKey,
                   issues.TypeID,
                   issues.TypeName,
                   issues.ProjectID,
                   issues.ProjectKey,
                   issues.ProjectName,
                   issues.DateCreated,
                   issues.DateUpdated,
                   issues.issueType,
                   issues.issueComponents,
                   issues.issueOriginalEstimated,
                              issues.ReporterEmployeeSID,
                              issues.issueFixVersions,
                              issues.issuePriority,
                              issues.issueLabels,
                              issues.issueRemainingTime,
                              issues.issueDueDate,
                              issues.AssigneeEmployeeSID,
                              issues.issueState,
                              issues.issueEstimatedEffort,
                              issues.issueSprint
            FROM issues
                 LEFT OUTER JOIN epic2subtask ON issues.issueID = epic2subtask.issueID
                         and issues.EpicKey = epic2subtask.EpicKey;
GO
```

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Projektarbeit/Studienarbeit/Bachelorarbeit mit dem Thema:

Empfehlungen und Hinweise zur Erstellung wissenschaftlicher Arbeiten

ohne fremde Hilfe angefertigt habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und

3. dass ich meine Projektarbeit/Studienarbeit/Bachelorarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Jena, 01.05.2019

Ort, Datum	Unterschrift