THM | CAMPUS GIESSEN | MNI
TECHNISCHE HOCHSCHULE MITTELHESSEN
Mathematik, Naturwissenschaften und Informatik

Bachelorarbeit

Analyse und Implementierung einer Depotverwaltung auf der Salesforce-Plattform

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

vorgelegt dem
Fachbereich Mathematik, Naturwissenschaften und Informatik
der Technischen Hochschule Mittelhessen
von

Max Euler

im August 2021

Referent: Prof. Dr. Frank Kammer
Korreferent: Prof. Dr. Harald Ritz

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 10. August 2021

Zusammenfassung

Das Ziel der vorliegenden Arbeit ist die Implementierung einer Depotverwaltung auf der Salesforce-Plattform. Dabei soll analysiert werden, ob eine Veröffentlichung der Applikation technisch möglich und wirtschaftlich sinnvoll ist. Die Idee entstand aus einem ähnlichen Projekt, welches auf Fonds basierte. Daraus ergab sich die Frage, ob dies auch mit der Verwendung von Aktien umsetzbar ist.

Für die Beantwortung der Frage mussten die verschiedenen Möglichkeiten zur Umsetzung der Anforderungen an die Applikation verglichen und diskutiert werden. Außerdem spielten bewährte Verfahren für solche Implementierungen bei der potenziellen Einsatzfähigkeit der Depotverwaltung eine entscheidende Rolle.

Die Implementierung zeigte, dass die Umsetzung der Applikation in einer einfachen Form zwar möglich ist, jedoch sind einige Funktionalitäten, welche in solch einer Applikation erwartet werden, auf der Salesforce-Plattform auf die geplante Art und Weise nicht umsetzbar sind.

Anhand des Ergebnisses lässt sich ableiten, dass die Bereitstellung von Daten in Echtzeit und die Erstellung einer generischen und jedoch gleichzeitig auch flexiblen Applikation die Hindernisse einer erfolgreichen Veröffentlichung sind.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Die Erstellung einer umfangreichen und lückenlosen Darstellung der Kunden eines Finanzunternehmens ist eine der großen wirtschaftlichen Herausforderungen der letzten Jahre, welche auch durch die fortschreitende Digitalisierung der Unternehmensprozesse an Bedeutung gewonnen hat.[1] Unternehmen aus der Finanzbranche sind dafür bekannt, neue Technologien, speziell im Hinblick auf das sogenannte *Cloud-Computing*[1], frühzeitig zu übernehmen.[2] Durch den Einsatz verschiedenster Systeme und der Gewinnung von Daten aus unterschiedlichsten Quellen, entsteht eine große Komplexität in der Zusammenführung dieser Daten. Eine Arbeit ist effizienter und einfacher, wenn Kundendaten an einem Ort gesammelt, abgerufen und bearbeitet werden können. Diese Möglichkeit bietet die Firma Salesforce, welche ihre sogenannten *CRM-Funktionalitäten*, was für *Customer Relationship Management* steht, als Plattform zur Verfügung stellt.[2] Doch auch dieses Produkt wird nur in einer Basis-Version ausgeliefert, welche zwar viele umfangreiche Funktionalitäten enthält, doch gerade für spezifische Themen von Unternehmen verschiedener Branchen individuell erweitert werden muss. Diese Erweiterungen finden durch den Einsatz von spezialisierten Entwicklern oder in Form von Applikationen statt, welche über den sogenannten *AppExchange* installiert werden können. Dabei handelt es sich um ein Produkt von Salesforce, welches kostenfreie und kostenpflichtige Lösungen zur Erweiterung der Salesforce Funktionalitäten bietet. Oftmals ist es jedoch der Fall, dass für bestimmte Anforderungen eine generische Lösung nur schwer oder in manchen Fällen auch überhaupt nicht umsetzbar ist. Deshalb bietet sich der *AppExchange* nicht für alle Projekte eine optimale Lösung. Allgemein sind Anforderungen an solche Lösungen immer von individuellen Kundenvorstellungen abhängig, wodurch es nicht immer möglich ist, eine generell zutreffende Applikation zur Verfügung zu stellen. In solchen Fällen müssen andere Wege zum Vertrieb der Lösung eingesetzt werden. Ein interessantes Beispiel für solch eine Applikation ist die Möglichkeit zur Depot- und Aktienverwaltung innerhalb der Salesforce-Plattform. Die Erwartungen an solch eine Lösung unterscheiden sich in der Regel in der Anzahl der Aktien[3], der Aktualität der Preis-Daten oder auch generell der Herkunft der Daten. Diese Art von Applikation wird in dieser Arbeit implementiert.

Diese Arbeit wird sich damit beschäftigen, einen wirtschaftlichen und fachlich sinnvollen Weg zu finden, eine solche Applikation implementieren und veröffentlichen zu können. Dabei steht die Implementierung im Vordergrund. Im Hinblick auf diese muss entschieden werden, welche Konzepte und Technologien eingesetzt werden müssen, um eine robuste

[1] Wird in Abschnitt 2.1 besprochen
[2] Wird in Abschnitt 2.1 besprochen
[3] Wird in Abschnitt 2.1 besprochen

und skalierbare Lösung zu entwickeln. Die Salesforce-Plattform bietet hier zahlreiche Möglichkeiten, zwischen welchen abgewogen werden muss. Außerdem gibt es Limits und Begrenzungen der Plattform, die berücksichtigt werden müssen. All diese Themen werden in einem separaten Abschnitt besprochen und evaluiert.

Jedoch muss ebenfalls besprochen werden, in welcher Form die Applikation am Ende veröffentlicht werden kann. Der bereits erwähnte *AppExchange* bietet dafür eine Möglichkeit. Das zentrale Problem hierbei ist jedoch die Bereitstellung der Daten. Anhand eigener Erfahrungen ist bereits bekannt, dass hier auch vertragliche und rechtliche Aspekte bedacht werden müssen. Eine möglicher Lösung wäre eine Auslieferung ohne Daten oder der Einsatz von Lizenzen. Das Ziel der Arbeit ist es neben der implementierten Applikation eine Antwort auf die Frage „Auf welche Art und Weise eine solche Lösung vertrieben werden?“ zu liefern.

Bevor mit der Implementierung begonnen werden kann, muss zuerst die Ausgangslage analysiert werden. Es ist wichtig zu verstehen, welche Rolle heutzutage der Einsatz von *Cloud-Computing Technologien*, mit Hinblick auf Salesforce und die Finanzbranche spielt. Außerdem werden die Aktien als Finanzprodukt und deren Bedeutung betrachtet. Das alles auch mit Fokus der aktuellen Entwicklung im Zuge der Corona-Pandemie. Als Grundlage der Implementierung werden darauffolgend Entscheidungen zu den eingesetzten Technologien getroffen. Das betrifft bestimmte Interfaces in der Code-Entwicklung, Möglichkeiten zur Datenverarbeitung und -darstellung, sowie eine Abwägung zwischen dem Einsatz von Code und einer deklarativen Lösung, welche auf der Benutzeroberfläche implementiert werden könnte. Während der Implementierung werden die eingesetzten Technologien und Wege beschrieben, eventuelle Fehltritte und Probleme erklärt und die implementierten Lösungen präsentiert und beschrieben. Der Abschluss der Arbeit gibt einen Ausblick darüber, ob und in welcher Form die Applikation vertrieben werden kann und in welchen Bereichen noch Verbesserungen oder Erweiterungen möglich sind.

Die Arbeit besitzt einen praktischen Schwerpunkt. Im Mittelpunkt steht die Implementierung einer Depotverwaltung auf der Salesforce-Plattform. Dies war die grundlegende Idee, welche zur Wahl des Themas dieser Arbeit führte. Dabei liegt der Fokus auf dem Einsatz der passenden Salesforce Technologien und dem Entwurf einer ansprechenden und effektiv einsetzbaren Darstellung der Informationen.

2 Grundlagen

2.1 Ausgangssituation

Die Welt des sogenannten *Cloud-Computings* gewinnt seit einigen Jahren erheblich an Wichtigkeit und dringt zunehmend in immer mehr Bereiche ein. Rimal et al. definieren Cloud Computing als, *a model of service delivery and access where dynamically scalable and virtualized resources are provided as a serviceover the Internet.*[3] Es geht also darum, dass Rechenleistung, Speicherplatz und auch Service Leistungen sich nicht mehr im eigenen System befinden, sondern über das Internet bereitgestellt werden. Diesen Aufschwung nutzt das Unternehmen Salesforce, um die Vorteile der Cloud in das Management von Kundenbeziehungen einzubringen. Mithilfe von Salesforce können Kundenbeziehungen gepflegt und verwaltet, verschiedene Systeme können eingebunden und zugeschnittene Applikationen entwickelt werden. All das ist möglich, ohne Software auf dem eigenen Rechner installieren zu müssen. Benötigt wird lediglich eine Internetverbindung. Mit *force.com* bietet Salesforce eine Architektur auf Metadaten-Basis, die es Entwicklern ermöglicht jegliche Art von anspruchsvollen und anpassbaren Geschäftsapplikationen in der Cloud zu entwickeln und darüber zur Verfügung zu stellen.[4] Der Vorteil dieses Modells besteht darin, dass die gesamte komplexe Architektur zum Betrieb dieser Plattform im Hintergrund liegt, wodurch sich der Entwickler selbst sich nur auf seine Implementierung konzentrieren muss. Alle anderen Themen werden von der Plattform übernommen.[4]

Weitere Vorteile von Cloud-Computing können Tabelle 2.1 entnommen werden. Neben des einfacheren Zugriffs auf die Software über das Internet, fällt auch die Verwaltung von Aktualisierungen und einer Versionsverwaltung der Software weg. Salesforce aktualisiert die Software regelmäßig und automatisch, ohne dass der Verwender eingreifen muss. Ebenfalls entfallen alle Aspekte hinsichtlich der Beschaffung und Bewahrung von Hardware. Durch den Einsatz von Cloud-Computing muss lediglich für die verwendete Hardware bezahlt werden, nach dem sogenannten *pay for what you need* Prinzip.

Die Corona Pandemie hat den Vormarsch des Cloud-Computings nochmal signifikant verstärkt. Die Digitalisierung wird noch stärker vorangetrieben. Zum einen wird in Firmen immer mehr auf Arbeiten von Zuhause aus gesetzt. Diese Arbeitsweise setzt einen Standort unabhängigen Zugriff auf notwendige Systeme voraus. Wie bereits erwähnt, reicht bei Cloud-Computing Systemen ein Internetzugang, da Software nicht mehr lokal installiert werden muss. Ein anderer Punkt ist die Leistungsqualität von Unternehmen gegenüber von Kunden. Hier muss ein positives Auftreten der Firma auch über eine distanzierte Betreuung gewährleistet werden. Auch in der Finanzbrache ist dieser Wandel erkennbar.

Function	On Premise	Cloud Computing
Software	Installed on your computer	Delivered via service over Web
Access	Through your computer	Through the Internet
Upgrades	Manual and complex	Automatic and easy
Versions	Multiple versions to maintain	Single code base with no infrastructure to maintain
Hardware	Purchase, Maintain and Manage	Pay for what you need

Tabelle 2.1: Vergleich zwischen On Premise und Cloud Computing[5]

Banken setzen vermehrt auf die Cloud und viele greifen dabei auf Salesforce zurück. Dabei werden die unterschiedlichsten Geschäftsbereiche auf die Salesforce-Plattform verlagert und immer mehr Abteilungen der Unternehmen verbringen ihren Arbeitstag in der Cloud. Dies ergaben eigene Erfahrungen durch die Arbeit in der Finanzbranche.

Ein weiterer Trend, der während der Pandemie stark zugenommen hat, ist der Handel mit Aktien. Gerade jüngere Menschen setzen sich zunehmend mit dem Thema auseinander. Abbildung 2.1 zeigt auf, dass über 20% der Befragten einer Studie während der Corona-Pandemie verstärkt Aktien gekauft haben. Währenddessen trifft das bei Investmentfonds auf nur ungefähr sechs Prozent zu.

Bei Aktien handelt es sich um Eigentumsanteile an Unternehmen. Sie werden in der Regel an der Börse gehandelt. Aktiengesellschaften, also Unternehmen von denen Aktien gehandelt werden können, teilen ihr Eigenkapital in eine bestimmte Anzahl an Aktien auf und geben diese zum Handel frei. Durch den Kauf von Aktien eines Unternehmens wird man Miteigentümer der Aktiengesellschaft, da man im anteiligen Besitz des Vermögens des Unternehmens ist. Daraus ergeben sich oftmals einige Rechte, wie beispielsweise die Teilnahme an einer Hauptversammlung mit zugehörigem Stimmrecht.[7]

Zusätzlich zur Pandemie bestärkt ebenfalls die Diskussion zur gesetzlichen Rentenversicherung den Trend zu einer eigenen Vorsorge. Junge Menschen können damit rechnen, dass das aktuelle Rentensystem in seiner heutigen Funktion bald nicht mehr gewährleisten kann, dass eine sichere und ausreichende Rente ausgezahlt werden kann.[8]

Anhand dieser Ausgangssituation ergibt sich dementsprechend ein Bedarf einer Applikation zur Verwaltung von Aktien-Depots auf der Salesforce-Plattform.

2.2 Projekt-Idee

Die Grundidee der Applikation war es, Finanzdienstleistern eine 360 Grad Ansicht ihrer Kunden zu ermöglichen. Dabei soll es hier um den Bereich des Aktienhandels und der Depotverwaltung gehen. Da der Handel selbst nicht über Salesforce geschehen wird, soll die Applikation es jedoch ermöglichen, die entscheidenden Daten dazu in Salesforce integrieren

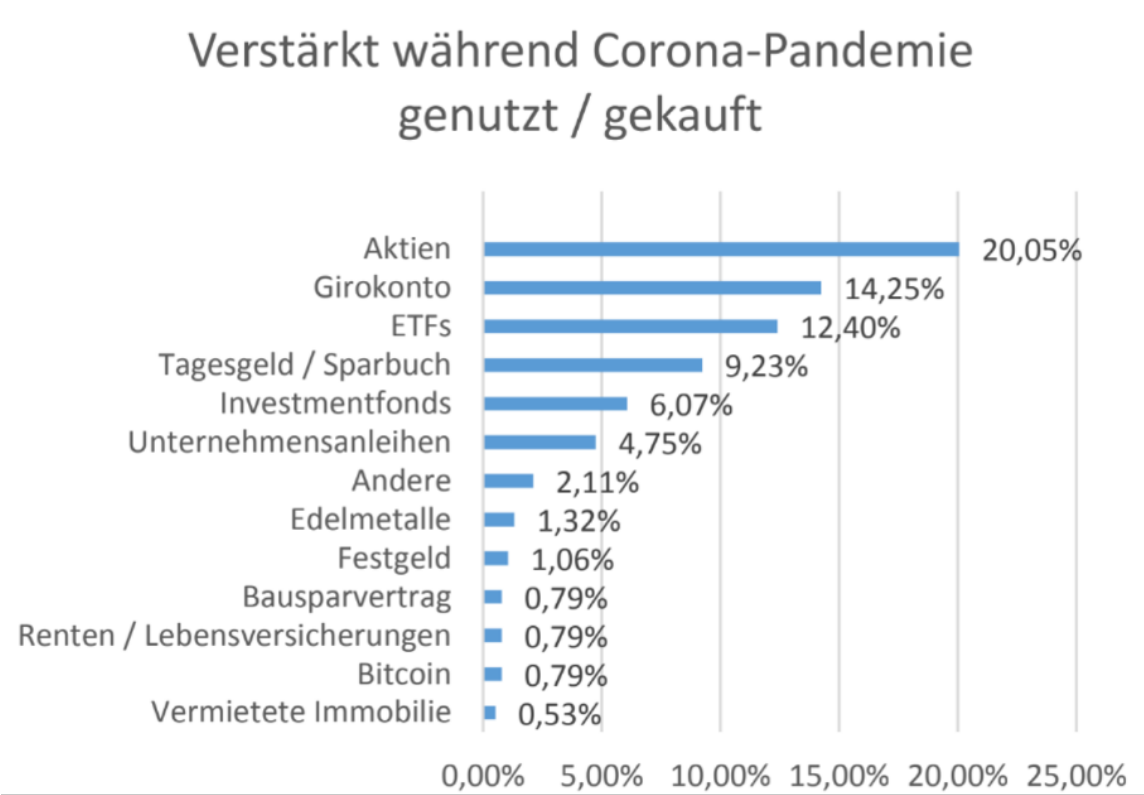

Abbildung 2.1: Verstärkt genutzt/gekauft während der Corona-Pandemie[6]

zu können. Genau dies ist auch das Grundprinzip, welches hinter Salesforce steckt. Die verschiedenen eingesetzten Systeme eines Kunden werden mit Salesforce integriert, um dort alle Daten an einer zentralen Stelle sammeln und verwenden zu können. Die Applikation wird eine Schnittstelle liefern, welche die Daten entgegennehmen und korrekt verarbeiten wird. Außerdem bringt sie die Möglichkeit zum automatischen Einlesen von Aktienpreisen mit, die aber je nach Vorstellungen des Anwenders individualisiert oder ersetzt werden kann. Ziel ist es wirklich nur das grundlegende Datenmodell und Automatisierungen zu liefern, die in die verschiedensten vorhandenen Salesforce Umgebungen der Benutzer integriert werden können.

Mit den Informationen in Salesforce können die Verwalter beispielsweise Berichte, besprochen in Abschnitt 3.10, und sogenannte *Dashboards* über alle für das Unternehmen entscheidenden Kennzahlen erzeugen. Mit Dashboards können Daten aus verschiedenen Berichten zusammen auf einer Oberfläche in Form verschiedener Diagrammtypen präsentiert werden. [9]

Durch eine mögliche Integration der Daten in das Tableau CRM[1], eine Plattform zur Datenvisualisierung und Business Intelligence (BI), sind noch mehr analytische Vorgänge möglich, bei denen auch künstliche Intelligenz eingesetzt werden kann. Dafür müssen die Daten

[1] https://www.salesforce.com/de/products/einstein-analytics/overview/

aber vorerst in Salesforce verfügbar sein. Die zahlreichen Analyse- und Darstellungsfunktionen können in verschiedenen Ebenen der Firmen wichtige Fragen beantworten. Für das Management oder die Controlling Abteilung könnten folgende Informationen interessant sein:

- Welches Gesamtvermögen verwalten wir für unsere Kunden?
- Wie entwickelt sich die Rate der Depoteröffnungen?

Die Depots werden in Salesforce immer einem zuständigen Verwalter zugeordnet, lassen sich aber auch manuell anhand der internen Kriterien des jeweiligen Unternehmens zuordnen. So können Manager mehr Überblick über ihr Team erlangen und beispielsweise Antworten auf folgende Fragen erhalten:

- Wie viele Depots verwaltet jeder Mitarbeiter?
- Welchen Gesamtbetrag verwaltet jeder Mitarbeiter?
- Die Depots welcher Mitarbeiter entwickeln sich am besten?

Die Antworten können über Berichte und Dashboards für die passenden Mitarbeiter zugänglich gemacht werden.

Ein Problem, welches sich bei solchen Applikationen immer ergibt, ist die Beschaffung der Daten. In diesem Fall die Aktienpreise und Informationen zu den Unternehmen. Für die Implementierung ist es kein Problem, einen Anbieter zu finden, der eine nützliche Menge an Daten zur kostenlosen Nutzung bereitstellt. Aus diesem Grund wurde über diese Problematik während der Entstehungs- und Ausarbeitungsphase dieser Idee nicht detaillierter nachgedacht. Für die kommerzielle Nutzung ist dies jedoch nicht so einfach umsetzbar. Hier müssen entweder Verträge direkt zwischen dem Endnutzer und dem Datenanbieter oder eine Art „Rahmenvertrag“ zwischen dem Entwickler der Applikation und dem Datenanbieter abgeschlossen werden. Solch ein „Rahmenvertrag“ würde eine grundlegende Vereinbarung zur Zusammenarbeit beinhalten und müsste dann für jeden neuen Endbenutzer erweitert werden. Dieses Verfahren würde im Zusammenspiel mit einem Vertrieb durch Lizenzen zum Einsatz kommen können. Der Preis der Lizenz würde die Kosten der Datenlieferung beinhalten. Andernfalls kann die Applikation natürlich auch ohne Lizenz installiert und genutzt werden, dafür ist dann aber die Einrichtung einer eigenen Datenquelle notwendig. Die Logik würde so entwickelt werden, dass eine Beschaffung über einen vorgegebenen Anbieter ohne Anpassungen möglich ist. Gleichzeitig soll es mithilfe weniger Anpassungen möglich sein, die Logik an einen eigens ausgewählten anderen Anbieter anzupassen. Der Prozess zur Abfrage der Daten und das gelieferte Format unterscheiden sich zwischen den Anbietern nur geringfügig. Das ermöglicht eine einfache und schnelle Anpassung.

Die Grundidee entstand während eines Projekts in der Zeit als Werkstudent in der Saleforce Entwicklung. Dabei ging es um die Einspielung von Fondsdaten in eine vorhandenes Salesforce Umgebung eines Kunden. Dieser Kunde war ein Vermögensverwalter, der das Geld seiner Kunden in verschiedene Fonds angelegt. Zur Verwaltung seiner Endkunden

setzt das Unternehmen Salesforce ein. Das System sollte nun um tagesaktuelle Preisdaten der Fonds und weitere Informationen über diese ergänzt werden. Aus diesem Projekt entstand die Idee zum Einlesen und Verwalten von Aktien Depots in Salesforce. Wie der Kunde auch, sollen hier die Aktien nicht in Salesforce selbst gehandelt werden, sondern nur die Zusammenführung der Informationen ist Aufgabe der Applikation.

Salesforce bietet für die Implementierung eines Webservices zwei Protokolle an: REST und SOAP. Dabei ist Ersteres die favorisierte Variante. Der Plan ist der Einsatz des sogenannten APEX REST, welches in Abschnitt 3.5 im Kapitel zu den Salesforce Konzepten besprochen wird. Da die Depotverwaltung zwei unterschiedliche Kategorien von Daten, die Tagespreise der Aktien und die Informationen zu den Aktien benötigt, werden hier mehrere Lösungen zur Bereitstellung implementiert werden müssen. Ein wichtiger Punkt betrifft den zeitlichen Einsatz dieser Lösungen. Es muss besprochen und festgelegt werden, welche Daten zu welchem Zeitpunkt abgefragt werden müssen.

Während des Einspielens von Daten in die Salesforce-Plattform müssen diese gegebenenfalls bearbeitet werden, um sie erfolgreich und brauchbar abzuspeichern. Dabei kann es sich um die Formatierung eines Datums oder die Änderung des Trennzeichens für Zahlenwerte handeln. Der Wert eines Datums ist beispielsweise nur brauchbar, wenn dieser auch in einem Feld vom Typ Datum abgespeichert wird. Nur dann lassen sich die Werte als Datum verwenden und die Datensätze können gruppiert, sortiert oder gefiltert werden. Für die Vorverarbeitung der Daten bietet Salesforce ebenfalls mehrere Optionen. Dort gilt es zu analysieren, welche davon für einzelne Szenarien am besten geeignet sind. Auch die Kombination von mehreren Optionen ist nicht auszuschließen und erfordert deshalb eine genaue Betrachtung für eine erfolgreiche Zusammenarbeit. Insbesondere der Einsatz von APEX REST würde dem entgegenkommen, da die Vorverarbeitung dort direkt stattfinden könnte, nachdem die Daten abgefragt wurden.

Die Darstellung der Daten soll hauptsächlich durch die bereits erwähnten Berichte und dazugehörigen Diagramme aufgebaut werden. Hier muss entschieden werden, welche Daten entscheidend sind und in welchem Format diese am besten und sinnvollsten präsentiert werden können.

Die Applikation teilt sich grundsätzlich in drei Bereiche. Es werden benutzerdefinierte Schnittstellen implementiert, um alle notwendigen Daten in das System integrieren zu können. Diese Schnittstellen können anschließend von den eingesetzten Systemen des Nutzers der Applikation verwendet werden. Außerdem werden Daten aus anderen Systemen explizit abgefragt, um beispielsweise die Aktien immer aktuell halten zu können. Ein weiterer wichtiger Punkt ist eine sinnvolle Darstellung der Daten auf der Benutzeroberfläche. Bei den Depots soll der Gesamtwert, die Aufteilung, getätigte Käufe und Verkäufe und der Verlauf des Depotwerts dargestellt werden. Zu den Aktien werden tagesaktuelle Preise und einige Informationen über das Unternehmen verfügbar gemacht. In den Bereichen, in denen es sinnvoll ist, sollen dynamische Diagramme zur Veranschaulichung eingesetzt werden. Natürlich können aufgrund der Vielzahl an vorhandenen Daten auch eigene gewünschte Diagramme erstellt werden. Es können Berechnung und Umwandlungen der Daten je nach Belieben des Nutzers durchgeführt werden. Die Applikation bietet die Grundlage für all diese aufgelisteten Punkte.

Auf der Salesforce-Plattform sind viele Funktionalitäten bereits standardmäßig vorhanden. Diese können somit ohne große Anpassungen eingesetzt werden. Benutzerdefinierte Themen, wie die hier besprochene Applikation, erfordern jedoch aufwendige Anpassungen. Dies hat zur Folge, dass solch eine Implementierung mit hohen Kosten in Verbindung gebracht werden kann. Es sind eine gewisse Anzahl an Arbeitsstunden und ebenfalls Kenntnisse der benötigten Technologien nötig, um solch eine Implementierung umsetzen zu können. Aus diesen Gründen sind vorgefertigte Applikationen oftmals eine passendere Möglichkeit für Salesforce-Nutzer.

2.3 Performance Kennzahlen

Die Idee zur Ergänzung der Applikation um einen weiteren Bereich kam während der Implementierung. Die sogenannten Performance Kennzahlen können wichtige Informationen über die Entwicklung der Kunden-Depots liefern. Diese Informationen sind zum einen wichtig für den Inhaber des Depots, um die Entwicklung besser verfolgen zu können. Denn das Ziel jeden Investors ist es, Aktien mit einer hohen absoluten Performance im Portfolio zu haben.[10] Ebenso werden die Verwalter davon profitieren, genauer gesagt die Managerebene des Unternehmens. Die Depots können anhand dieser Kennzahlen bewertet und verglichen werden. Außerdem können auch die einzelnen Unternehmen damit bewertet werden und Depotverwalter könnten aufgrund dieser Bewertungen entscheiden gewisse Aktien besonders stark zu empfehlen oder im Gegenteil sogar gegebenenfalls aus ihrem Angebot zu streichen. Die Kennzahlen können für mehrere Zeiträume angegeben werden, was besonders auf die darauf basierenden Diagramme sehr nützlich sein kann. Die Schwierigkeit ergibt sich durch die aufwendige Berechnung, da die Performance Kennzahlen nicht bei der Abfrage der Aktien-Preise enthalten sind.

Die Werte müssen sowohl für die einzelnen Aktien als auch auf den Depots berechnet werden. Für die Berechnung auf den Aktien müssen, je nach gewünschtem Zeitraum, zwei Tagespreise verwendet werden. Folgende Formel wird zu Berechnung angewendet:

Zu berechnen ist die Performance der letzten sieben Tage. X ist der aktuelle Preis und Y der Preis vor sieben Tagen. Zuerst muss der ursprüngliche Wert Y, vom aktuellen Wert X, subtrahiert werden. Die Differenz wird nun durch den ursprünglichen Wert Y dividiert. Dieser Quotient muss anschließend mit 100 multipliziert werden. Der erhaltene Produktwert stellt die gewünschte Performance dar. Wäre der aktuelle Preis einer Aktie 132,78 € und der Preis vor sieben Tagen 128,46 €, würde sich folgende Performance daraus ergeben:

132,78 € - 128,46 € = 4,32 €
4,32 € / 128,46 € = 0,033629
*0,033629 * 100 =* ***3,36*** *%*

Diese Berechnung muss täglich für jede Aktie und für jeden betrachteten Zeitraum durchgeführt werden. Die daraus resultierenden Werte ergeben die Grundlage für die Berechnung der Depot-Performance, welche anhand der Performance der Aktien berechnet wird, die

sich im Depot befinden. Natürlich muss dabei die Gewichtung der Aktien im Depot beachtet werden, wofür der prozentuale Wert auf dem Anteil ergänzt werden muss. Folgende Beispielrechnung verdeutlicht das:

Bestehe Depot Z, aus den Aktien X und Y, wobei X 46% und Y 54% des Depots ausmacht. Die Performance der letzten sieben Tage von X sei 3,4%, während der Wert für Y -1,02% beträgt.

$$\begin{aligned} \textit{Depot Performance} &= \textit{(Performance X * Anteil X) + (Performance Y * Anteil Y)} \\ &= \textit{(3,4 * 0,46) + (-1,02 * 0,54)} \\ &= \underline{\textbf{\textit{1,01}}}\ \textit{\%} \end{aligned}$$

Es ist geplant die Performance für die Zeiträume der letzten sieben Tage, der letzten 30 Tage und der letzten 365 Tage anzugeben.

2.4 Datenmodell

Nachdem die Idee der Applikation nun ausformuliert ist und alle grundlegenden Anforderungen feststehen, steht als nächster Schritt der Entwurf des Datenmodels an. Gerade bei Salesforce Projekten sollte dies vor Beginn der wirklichen Implementation geschehen. Eine oberflächliche Darstellung des Datenmodels wird in Abbildung 2.2 gezeigt. Es ist klar zu erkennen, dass die Beziehungen zwischen den sogenannten *Objekten* entscheidend sein werden. Auf der Saleforce-Plattform werden Daten in Objekten festgehalten. Ein Objekt spiegelt eine Tabelle in der Datenbank wider. Es wird zwischen Standard-Objekten und benutzerdefinierten Objekten unterschieden. Für die Applikation werden, außer das Objekt Kontakt, nur benutzerdefinierte Objekte eingesetzt. Diese müssen während der Implementierung angelegt werden. Die verschiedenen Werte auf den Objekten werden in den sogenannten *Feldern* festgehalten. Diese Felder sind mit den Spalten der Datenbanktabellen vergleichbar. Sie geben an, welche Informationen in den Objekten gespeichert werden können.[11] Für jedes Feld muss immer ein Datentyp angegeben werden. Möglich sind beispielsweise Text, Zahl, Prozent oder Auswahlkästchen. Gerade für diese Applikation wird der Datentyp Währung besonders wichtig sein. Bei diesem Typ wird der Wert auf der Benutzeroberfläche immer mit einem Währungssymbol angezeigt. Für die Anzeige der Aktienpreise oder des Depotwerts ist das essenziell.

Die Personen, also Inhaber der Depots, werden durch das Standard-Objekt Kontakt dargestellt. Bei diesem Thema musste vorher zwischen den bereits erwähnten Kontakten und den sogenannten Personaccounts entschieden werden.

„In Personenaccounts werden Informationen über Einzelpersonen gespeichert. Dabei werden bestimmte Account- und Kontaktfelder in einem einzigen Datensatz miteinander kombiniert.“ [12]

Personaccounts kombinieren denmach Accounts, welche Unternehmen darstellen, und Kontakte, die für Einzelpersonen erstellt werden. Die Entscheidung für die Nutzung der Kontakte fiel aus folgenden Gründen:

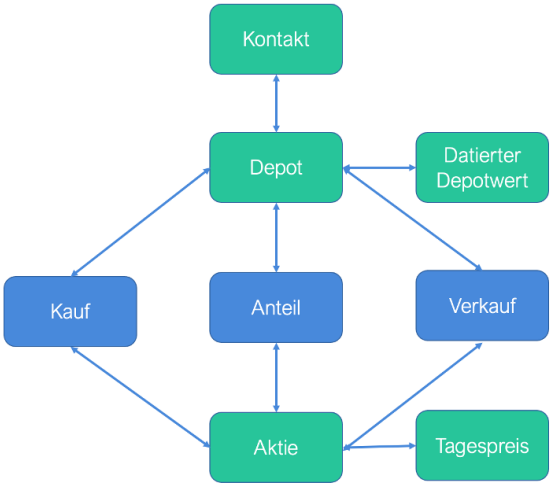

Abbildung 2.2: Datenmodell

1. Personaccounts sind ein besonderes Feature bei Salesforce. Sie sind nicht standardmäßig vorhanden und müssen vom Salesforce-Support aktiviert werden.[13]

2. Das fertige Projekt soll später gegebenenfalls als ein sogenanntes *unmanaged Package* in verschiedenen Salesforce Umgebungen installiert werden können. Diese werden in der Saleforce Welt genutzt, um eine zusammenhängende Gruppe von Funktionalitäten zu teilen oder installieren.[14] Aus diesem Grund ist es sinnvoll, auf die immer existierenden Kontakte zurückzugreifen, da Personaccounts nicht immer aktiviert sein müssen. Eine Notwendigkeit, diese für die Installation des Produkts aktivieren zu müssen, könnte manche potenziellen Kunden unter Umständen abschrecken.

Das Depot Objekt soll als eine Art „Rahmen-Objekt“ dienen. Es wird einige Informationen, wie Anlagedatum, eine Wertpapierdepot-Nummer und den zuständigen Verwalter, enthalten. Das wirkliche Herz des Depots, also die Wertpapiere, werden in einem anderen Objekt gespeichert. Nur so kann eine ressourcenschonende und skalierbare Nutzung garantiert werden. Jedes Wertpapier wird im System nur einmal angelegt. Es gibt beispielsweise genau einmal die Aktien Google, Facebook und Siemens. Ein falscher Weg wäre es, für jedes Depot einen neuen Datensatz dieser Aktie anzulegen, sobald sie in das jeweilige Depot aufgenommen wird. So hätte man nach kurzer Zeit unzählige Datensätze, welche die gleichen Informationen enthalten und bei möglichen Änderungen alle angepasst werden müssten.

Für die Erstellung der Beziehungen gibt es auf der Salesforce-Plattform mehrere Möglichkeiten. In den meisten Fällen muss die Wahl zwischen Nachschlage- und Master-Detail-Beziehungen getroffen werden. Nachschlage-Beziehungen sind lockerer, während Master-Detail-Beziehungen zwei Datensätze sehr eng aneinander binden.[15] Trotz ihrer unterschiedlichen Eigenschaften liegt unter beiden Arten das gleiche Prinzip. Es entsteht eine Referenz auf einen Fremdschlüssel, welches man als *Foreign Key Mapping* bezeichnet:

„A Foreign Key Mapping maps an object reference to a foreign key in the database.“[16]

In der Applikation werden ausschließlich Master-Detail-Beziehungen eingesetzt. Dies hat den Grund, dass alle Beziehungen aus fachlicher Sicht eng sind. So gehört ein Preis immer nur zu einer Aktie und wird im Nachhinein auch nicht zu einer anderen Aktie zugeordnet. Dieses fachliche Verhalten soll in das Datenmodell übernommen werden. So gibt es beispielsweise bei der Erstellung einer Master-Detail-Beziehung die Auswahl, ob nach Anlage des untergeordneten Datensatzes, der übergeordnete Datensatz geändert werden darf. Dies wäre der Fall, wenn ein Tagespreis nach Anlage zu einer anderen Aktie zugeordnet werden würde. Da dies wie bereits erwähnt fachlich keinen Sinn ergibt, wird diese Option nicht ausgewählt. Eine andere entscheidende Eigenschaft einer Master-Detail-Beziehungen ist, dass bei Anlage eines untergeordneten Datensatzes immer ein übergeordneter Datensatz ausgewählt werden muss. Andernfalls kann kein Datensatz angelegt werden. Auch das macht für die fachliche Seite Sinn. Es sollte beispielsweise niemals ein Depot angelegt werden können, dass keinen Inhaber hat. Ebenfalls vorteilhaft ist das Verhalten beim Löschen eines übergeordneten Datensatzes. Dabei werden automatisch alle untergeordneten Datensätze gelöscht. Vorteilhaft wäre das beim Löschen eines Depots. Dabei sollten ebenfalls alle Anteile des Depots gelöscht werden, da diese ohne das Depots nicht mehr notwendig sind.

Ein besonderes Konstrukt sind die Objekte Anteil, Kauf und Verkauf. Zwischen Depot und Aktie werden drei fachlich unterschiedliche *N zu N*-Beziehungen benötigt. Auf der Salesforce-Plattform nutzt man ein sogenanntes *Junction*-Objekt zur Erstellung einer *N zu N*-Beziehung. Das Objekt liegt zwischen den Objekten Depot und Aktie und für jede neue Beziehung zwischen diesen wird ein neuer Datensatz des Junction-Objektes angelegt. Der Datensatz kombiniert zwei *Eins zu N*-Beziehungen und gibt damit die Möglichkiet zur Erstellung einer *N zu N*-Beziehung, die durch die zwei einzelnen Beziehungen repräsentiert wird.[17] Für eine der Beziehungen macht für das Junction-Objekt der Name „Anteil“ Sinn, da es gibt Informationen darüber gibt, welche Aktien sich im Depot befinden. Auf dem Anteil Objekt können weitere wichtige Informationen gespeichert werden. Dabei kann es sich um die Anzahl der Aktien handeln und der Gesamtwert, den diese Aktien zusammen widerspiegeln. Nehme man das Beispiel der Adidas Aktie. Angenommen der Preis pro Stück sei bei 200 Euro und im Depot befänden sich davon drei Aktien. Hierfür würde eine Datensatz des Objekts Anteil angelegt werden, der auf den Datensatz der Aktie Adidas und das Depot des Kunden zeigen würde. Auf dem Anteil wird dazu die Anzahl drei gespeichert und der Preis kann berechnet werden. In diesem Fall wäre das dreimal 200 Euro, in Summe demnach 600 Euro. Auf der Datenbankebene sind Junction-Objekte mit Fremdenschlüsseltabellen zu vergleichen. Es gibt zwei Spalten, die jeweils den Primärschlüssel der zwei zu verbindenden Objekte enthalten und weitere Spalten, in denen zusätzliche Informationen über die Beziehung gespeichert werden können.

Das Aktien Objekt wird ähnlich aufgebaut sein wie das Depot Objekt. Es wird einige Stammdaten des jeweiligen Wertpapiers halten. Dabei handelt es sich um den Namen des Unternehmens, das Aktiensymbol und andere Stammdaten. Das Aktiensymbol ist eine eindeutige Identifizierung eines börsennotierten Unternehmens. Es wird oftmals auch als Börsenticker oder Tickersymbol bezeichnet.[18] Doch die wirklich entscheidende Information, der aktuelle Preis, wird wieder in einem anderen Objekt gespeichert und über eine *Eins zu N*-Beziehung mit der Aktie verbunden. Gäbe es immer nur den aktuellen Preis zu

betrachten, könnte man diesen auch immer direkt auf dem Aktien Datensatz speichern. Um aber eine Entwicklung der Depots und der einzelnen Aktien darstellen zu können, müssen historische Preisdaten vorhanden sein. Außerdem existiert an der Börse ja nicht nur genau ein Preis. Schaut man sich den Wert einer Aktie für ein bestimmtes Datum an, kann man das Tageshoch, Tagestief oder den Preis bei der Öffnung beziehungsweise Schließung der Börse betrachten. Aus diesen Gründen ist es mehr als sinnvoll, die Preise in einem zusätzlichen Objekt zu speichern. Für jede Aktie im System wird somit an jedem Tag ein neuer Datensatz dieses Objekts angelegt. Je nach Aktualisierungsintervall muss dieser Datensatz mehrmals am Tag aktualisiert werden, um den aktuellen Preis anzeigen zu können. Am nächsten Tag wird ein neuer Datensatz angelegt und der alte bleibt für historische Zwecke vorhanden.

2.5 Geplante Prozesse

Nach Formulierung der grundlegenden Idee und Erstellung eines passenden Datenmodells, muss nun diskutiert werden, welche Prozesse und Automatisierungen implementiert werden müssen, um eine praktische und effiziente Nutzung der Applikation gewährleisten zu können.

Anlage und Pflege der Depots in Salesforce werden komplett automatisiert ablaufen. Die wichtigsten Funktionalitäten der Applikation werden als REST-Webservicevorgang[2] zur Verfügung gestellt. Somit kann eine Integration der Depotverwaltung mit anderen Systemen des Kunden aufgebaut werden. Salesforce empfängt die Daten und pflegt sie dann in das System ein. Hier ist eine Definition von Fremdschlüsseln entscheidend, um eine eindeutige Zuordnung der Daten gewährleisten zu können. Für die Depots kann dafür die Depotnummer verwendet werden, da diese eindeutig ist. Bei den Aktien bietet sich das Aktiensymbol an.

Für die automatische Anlage der Depots werden zwei Informationen benötigt, welche der REST-Webservicevorgang entgegennehmen muss: Eine Kennung des Inhabers und die eindeutige Depotnummer. Alle anderen Informationen über das Depot werden automatisch durch Käufe und Verkäufe erstellt. Auch die Aktien werden automatisch angelegt. Dafür wird ebenfalls ein REST-Webservicevorgang implementiert. Die einzige benötigte Information ist das Aktiensymbol der Aktie. Um Beispielsweise die Apple Aktie in Salesforce aufzunehmen, reicht ein Aufruf mit folgenden Informationen:

{

“Symbol___c”: “AAPL”

}

Die weiteren wichtigen Informationen über die Aktien werden automatisch in Salesforce ergänzt. Dafür ist kein Handeln eines Mitarbeiters nötig. Zum einen werden einige Stammdaten geladen, wie beispielsweise die Assetklasse oder Dividendenrendite. Außerdem werden nach Anlage die Preisdaten der letzten 100 Tage geladen, um die Historie der Aktie

[2] Wird in Abschnitt 3.5 erläutert

darstellen zu können. Darüber hinaus wird in Zukunft täglich der aktuelle Preis der Aktie über einen Webservice eingelesen und gespeichert. Die Preise der Aktien sind somit immer tagesaktuell.

Ein weiterer benötigter Prozess wird für die Erhaltung der historischen Depotwerte implementiert werden müssen. Durch getätigte Käufe, Verkäufe und auch die Änderungen der Aktienpreise wird sich der Wert des Depots täglich ändern, wodurch es nicht möglich sein wird, den Verlauf dieser Änderungen rückwirkend dynamisch darzustellen. Aus diesem Grund muss der Wert des Depots regelmäßig festgehalten werden. Das Ganze sollte automatisch und ressourcenschonend geschehen, da durch eine gegebenenfalls große Anzahl an Depots mit einer gewissen Datenmenge gearbeitet werden muss und dadurch die Speicherkapazitäten und die Verarbeitungsdauer berücksichtigt werden müssen. Es sollte demnach eine minimale Lösung implementiert werden. Da fachlich nur der Wert, das Datum und die Zuordnung zu einem Depot benötigt werden, reichen diese drei Informationen bereits als Datenmodell aus. Das Objekt ist im Datenmodell bereits unter „datierter Depotwert“ aufgeführt. Der Prozess muss täglich einen Datensatz pro Depot angelegen, um die Historie der Entwicklung festhalten zu können.

Bei der Formulierung der Idee wurde ebenfalls die Zuordnung der Depots zu einem bestimmten Mitarbeiter besprochen. Dieses Thema wird sehr kompliziert umzusetzen sein. Das Problem dabei ist, dass die Zuordnungen von Depots zu einem spezifischen Mitarbeiter meistens aufgrund intern festgelegter Bedingungen geschehen. Als Entwickler einer Applikation ist es sehr schwierig dafür im Voraus einen festen Prozess zu entwickeln. Um dennoch eine Logik anbieten zu können, ist folgendes Vorgehen geplant: Für Kontakte und Accounts gibt es standardmäßig einen Inhaber, also einen Mitarbeiter, der diese Einzelperson oder das Unternehmen betreut. Der zu erstellende Prozess wird den Inhaber des Kontaktes, welcher bei Erstellung des Depots angegeben wird, übernehmen und als Depotverwalter eintragen. So ist immer der Mitarbeiter, der für den Inhaber zuständig ist, auch gleichzeitig für das Depot der entsprechenden Personen zuständig. Gleichzeitig ist wichtig, dass dieses Verhalten überschrieben werden kann. Das muss ohne Anpassung von Code möglich sein. Auf diesen Aspekt wird während der Implementierung genauer eingegangen.

3 Salesforce Konzepte

3.1 Multi-Tenancy-Architektur

Aufgrund der sogenannten *Multi-Tenancy-Architektur* unterscheiden sich Implementierungen auf der Salesforce-Plattform in vielen Bereichen von anderen Implementierungen. Die Multi-Tenancy-Architektur kann mit dem Wohnen innerhalb eines Apartment Komplexes verglichen werden. Jede Partei im Komplex hat einen privaten Bereich, die eigene Wohnung. Diese kann nach Belieben eingerichtet und verwendet werden, während niemand Außenstehendes Zugang dazu hat. Im Komplex gibt es zusätzlich noch einige gemeinsam genutzte Bereiche, wie die Lobby oder die Tiefgarage. Der Code der Standard-Applikationen und die darunterliegenden Ressourcen und Datenbanken werden also von mehreren Kunden gleichzeitig verwendet.[19] Auf dem Salesforce-Server sind das beispielsweise die CPUs. Diese Architektur hat ebenfalls den Vorteil der Nachhaltigkeit, da der CO2-Fußabdruck durch die Nutzung des Cloud Services reduziert werden kann.[20]

Durch diese Gegebenheiten sind die sogenannten *Governor Limits* entstanden. Dabei handelt es sich um vorgegebene Grenzen für bestimmte Ressourcen, an welche sich benutzerdefinierte Implementierungen halten müssen.[4] Dieses Konzept sollte jeder Salesforce-Entwickler frühzeitig verstehen und somit beachten können, da es sich hier um Grundlagen und Voraussetzungen für erfolgreiche Entwicklungen handelt. Die Limits beziehen sich hauptsächlich auf APEX Code mit Hinblick auf den Einsatz von SOQL-Abfragen und -Statements. APEX ist eine objektorientierte und stark typisierte Programmiersprache, welche JAVA ähnelt und zur Implementierung benutzerdefinierter Geschäftslogik eingesetzt wird.[4] Bei SOQL, kurz für *Salesforce Object Query Language*, handelt es sich um die Datenbanksprache der Salesforce-Plattform.[21] Die gegebenen Limits sind beispielsweise die maximale CPU-Zeit, die Anzahl an SOQL-Abfragen oder die maximale Heap-Size.[4] Verschiedene Kunden teilen sich die Ressourcen einzelner Salesforce-Server. Damit nicht eine einzelne oder wenige Parteien den Großteil der Ressourcen für sich beanspruchen, wurden die Governor Limits eingeführt. Generell gelten diese Limits pro Transaktion. Sobald eine Transaktion abgeschlossen ist, werden die Limits zurückgesetzt. Das ist vor allem für sogenanntes *Batch APEX* essenziell, worüber im nächsten Abschnitt gesprochen wird. Für manche Limits wird zwischen einer synchronen und asynchronen Ausführung unterschieden, sodass im asynchronen Kontext erhöhte Limits gelten. Ein Beispiel dafür ist die maximale Anzahl an ausgeführten SOQL-Abfragen. Synchron sind 100 erlaubt, während asynchron bis zu 200 Stück gestattet werden.[22]

3.2 Batch APEX

Im Projekt dieser Arbeit wird an mehreren Stellen auf das sogenannte *Batch APEX* gesetzt. Der Grund dafür sind die eben besprochenen Governor Limits. Eines dieser Limits besagt, dass pro SOQL-Abfrage maximal 50.000 Datensätze abgerufen werden dürfen.[22] Sobald eine Abfrage 50.001 Datensätze abruft, wird eine *Limit Exception* geworfen. Diese kann nicht behandelt werden und führt zwangsläufig zum Abbruch der Transaktion. Um dieses Limit umgehen zu können, wurde Batch APEX eingeführt.

Batch APEX ist ein Salesforce-eigenes Interface, mit dem Namen *Batchable*, für die Verarbeitung einer großen Menge an Datensätzen. Das Konzept ist, dass zu Beginn der Ausführung die Datenmenge definiert und diese später aufgeteilt wird. Im Batch Kontext ist eine Verarbeitung von bis zu 50.000.000 Datensätzen möglich.[23] Diese Datensätze werden in gleich große Stücke, sogenannte *Batches* oder *Chunks*, geteilt und dann einzeln verarbeitet. Batches können eine Größe von einem bis maximal 2000 Datensätzen aufweisen.[23] Die Größe sollte je nach Komplexität der durchzuführenden Logik gewählt werden. Die Definition der Daten findet in der *start()*-Methode statt.[23] Diese muss eine SOQL-Abfrage zurückgeben, kann zuvor aber noch andere Logik ausführen. So kann beispielsweise die SOQL-Abfrage aufgrund bestimmter Parameter angepasst werden.

Der Hauptteil jeder Batch Klasse ist die *execute()*-Methode. Sie beinhaltet die Logik, die auf den definierten Datensätzen ausgeführt werden soll. Die Ausführung einer Batch Klasse kann als eine Art Job betrachtet werden. Während in der *start()*-Methode der Umfang des Jobs definiert wird, wird in der *execute()*-Methode die schwere Arbeit erledigt. Die Methode wird, je nach Anzahl der insgesamt zu verarbeitenden Datensätze, mehrfach aufgerufen. Sie nimmt einen Parameter namens „scope“ entgegen, welcher immer die Liste der aktuell zu verarbeitenden Datensätze beinhaltet. Nach jeder Ausführung wird die Klasse automatisch neu initialisiert und erhält einen neuen Satz an Governor Limits. Der *execute()*-Methode wird der nächste Batch an Datensätzen durch den „scope“ Parameter übergeben.

Den Schluss jeder Batch Klasse bildet die *finish()*-Methode. Sie wird einmal ausgeführt, nachdem alle Datensätze durch die *execute()*-Methode verarbeitet wurden.[23] Sie kann genutzt werden, um eine bestimmte Aufräum- oder Berichtslogik auszuführen. Ein weit verbreiteter Verwendungszweck ist das Senden einer E-Mail an einen Entwickler oder Projektmanager, mit Informationen über den ausgeführten Batch-Job.

Batch Klassen müssen auf eine bestimmte Weise aufgerufen werden. Salesforce bietet dafür die Methode *Database.executeBatch()*. Diese nimmt zwei Parameter entgegen. Zuerst eine Instanz der Batch Klasse und anschließend noch optional die Batch Größe.[23] Wird keine Batch Größe angegeben, wird die standardmäßige Größe von 200 verwendet.[23] Der Aufruf für die fiktive Klasse „MyExampleBatch“ mit einer Batch Größe von 50 würde folgendermaßen aussehen: *Database.executeBatch(new MyExampleBatch(), 50);*. Ein beispielhafter Ablauf eines Batch-Jobs, also eine sich in der Ausführung befindende Batch-Klasse, kann Abbildung 3.1 entnommen werden. Die in der *start()*-Methode definierte SOQL-Abfrage definiert eine Liste von 120.000 zutreffenden Datensätzen. Von dieser Liste werden nun fortlaufend 50 Datensätze entnommen und an die *execute()*-Methode übergeben. Nach jeder

Ausführung der *execute()*-Methode werden die nächsten 50 Datensätze übergeben. Nach Verarbeitung der vollständigen Liste wird die *finish()*-Methode einmalig aufgerufen.

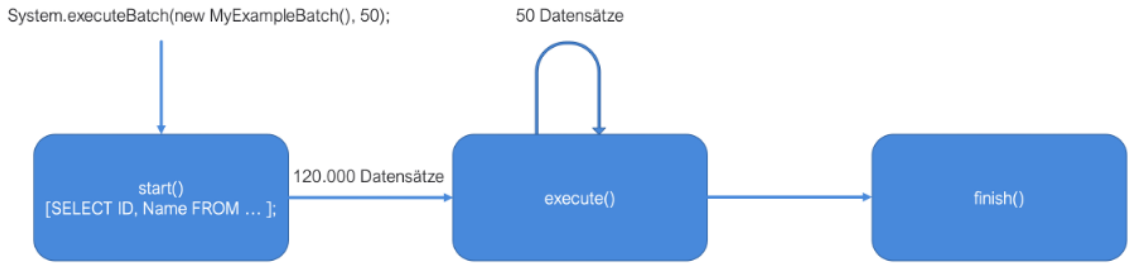

Abbildung 3.1: Ausführung einer Batch-Klasse

Das Konzept des Batchable Interfaces wird in der Applikation dieser Arbeit mehrfach eingesetzt werden müssen. Denn sobald eine gewisse Datenmenge erreicht wird, in Form von Aktien oder Depots, müssen Berechnungen, Einspielungen und ähnliches damit durchgeführt werden. Nur so können die bereits besprochenen Governor Limits eingehalten werden und eine skalierbare Lösung entstehen.

3.3 APEX Scheduler

Das sogenannte *APEX Scheduler* Interface macht es möglich, APEX Code zu bestimmten Zeiten automatisch auszuführen.[23] Das ist beispielsweise für nächtliche Datenverarbeitungen nützlich. Eigene Erfahrungen zeigen, dass das Interface oftmals in Kombination mit dem bereits besprochenen Batch APEX eingesetzt wird. Es wird eine Batch Klasse implementiert und mithilfe eines APEX Schedulers die wiederholte Ausführung der Klasse für bestimmte Zeitpunkte geplant.

Um dies zu erreichen muss eine Klasse vorhanden sein, die das Schedulable Interface implementiert. Das Interface besitzt eine *execute()*-Methode, die implementiert werden muss und bei Ausführung der Klasse automatisch aufgerufen wird.[23] Es ist ein bewährtes Verfahren, dass darin keine Logik ausgeführt wird. Sie soll lediglich die gewünschte Klasse bzw. Methode aufrufen. Ein Beispiel wäre die Klasse „MyScheduler“, welche die bereits bekannte Klasse „MyExampleBatch“ aufruft. Diese wird in Listing 3.1 aufgeführt.

```
global class MyScheduler implements Schedulable {
    global void execute(SchedulableContext sc) {
       MyExampleBatch exampleBatch = new MyExampleBatch();
       Database.executeBatch(exampleBatch, 50);
    }
}
```

Listing 3.1: Nutzung des Schedulable Interfaces

Das eigentliche Planen kann auf zwei Wegen geschehen. Der einfachere, aber dafür auch eingeschränktere Weg ist die Planung in den Einstellungen auf der Benutzeroberfläche. Dort können Klasse und Zeitpunkt mit einfachen Klicks gewählt werden. Da das Ziel der Applikation aber gegebenenfalls eine Veröffentlichung im AppExchange ist, ergibt diese Art keinen Sinn. Nach jeder Installation müsste das Verfahren zur Planung manuell wiederholt werden. Der zweite Weg dagegen ist der Aufruf innerhalb von APEX Code. Das geschieht durch den Aufruf der Klasse, welche das Schedulable Interface implementiert. Dafür kann die Methode *System.schedule();* verwendet werden.[23] Diese nimmt drei Parameter entgegen:

1. Den Namen des Jobs. Das ist ein beliebiger String, unter dem der Job auf der Übersichtsseite in den Einstellungen zugeordnet werden kann.
2. Einen Ausdruck, der den Zeitpunkt der geplanten Ausführung festlegt.
3. Eine Instanz der auszuführenden Klasse.

Der Ausdruck aus Punkt zwei setzt sich aus Sekunden, Minuten, Stunden, Tag des Monats, Monat, Wochentag und Jahr zusammen. In Kombination mit bestimmten Sonderzeichen können hier unzählige gewünschte Zeitpunkte und Intervalle erreicht werden.[23] Folgende zwei Beispiele stellen das Prinzip dar:

0 0 22 * * ? Diese Klasse wird täglich um 22 Uhr ausgeführt.
0 30 10 1 1 ? Diese Klasse läuft jedes Jahr am 1. Januar um 10:30 Uhr.

Soll nun die oben implementierte „MyScheduler“ Klasse von Montag bis Freitag um 20 Uhr laufen, würde der dafür benötigte Code wie in Listing 3.2 dargestellt aussehen.

```
MyScheduler myScheduler = new MyScheduler();
String expr = '0 0 20 ? * MON-FRI';
System.schedule('Example Scheduled Job', expr, myScheduler);
```

Listing 3.2: Planen eines wiederholten Auftrags

Ein zweites Verfahren zur Planung einer Batch Klasse, das aufgrund fachlicher Anforderungen oder der Einfachheit öfters eingesetzt wird, ist das wiederholte einmalige Planen des Batch Jobs. Dafür wird die Methode *System.scheduleBatch()* verwendet.[24] Eine passende fachliche Anforderung wäre der Satz: *Der Batch Job soll nach Abschluss der Verarbeitung nach vier Stunden erneut starten.* Um dies zu erreichen kann die *finish()*-Methode der Batch Klasse eingesetzt werden. Dort wird die eben genannte *System.scheduleBatch()*-Methode aufgerufen, mit welcher eine einmalige zukünftige Ausführung des Batches geplant werden kann. Die Methode nimmt drei verpflichtende und einen optionalen Parameter entgegen[25]:

1. Eine Instanz der Batch Klasse.
2. Der Name des Jobs, wie er auch bei der *System.schedule()*-Methode angegeben werden kann.

3. Ein Zeitintervall, nachdem der Batch Job starten soll. Dieses Intervall wird in Minuten angegeben.
4. Optional kann die Batch Größe angegeben werden. Falls diese nicht angegeben wird, wird sie standardmäßig 200 betragen.

In Listing 3.3 wird die *finish()*-Methode der bereits bekannten „MyExampleBatch“ Klasse dargestellt, wenn diese Klasse sich nach Ausführung automatisch neu planen soll. Als Intervall werden 240 Minuten, vier Stunden, angegeben und die Batch Größe wird auf 50 festgelegt.

```
public void finish(Database.BatchableContext batchContext){
    MyExampleBatch exampleBatch = new MyExampleBatch();
    System.scheduleBatch(exampleBatch, ‘Example Batch‘, 240, 50);
}
```

Listing 3.3: Einmaliges Planen einer Batch Klasse

Für die geplante Applikation kann das Schedulable Interface an mehreren Stellen eingesetzt werden. Dort wäre das täglich Einlesen der Preise oder das tägliche Abspeichern des Depotwerts zu nennen.

3.4 Queueable APEX

Mit dem sogenannten *Queueable Interface* bietet Salesforce die Möglichkeit asynchrone Aufträge zu starten, zu überwachen und zu verketten.[23] Es eignet sich, um intensive Operationen auszulagern und im Hintergrund laufen zu lassen. Dazu zählen beispielsweise umfangreiche Datenbank Operationen, lang andauernde Aufrufe von externen Systemen oder im Allgemeinen rechenintensive Ausführungen von Code. Der Vorteil von Queueable ist, dass die Ausführung unabhängig von der aktuellen Transaktion ist. Das bedeutet, dass die aktuelle Transaktion nicht behindert, beziehungsweise verlängert wird. Nach Aufruf der Klasse, die das Queueable Interface implementiert, wird die Ausführung dieses Auftrages an die sogenannte *Job Queue* gehängt. Dadurch kann die laufende Transaktion kann fortgeführt werden. Die Job Queue ist eine Art Warteschlange und beinhaltet alle asynchronen Jobs, die zur Ausführung aufgerufen, aber noch nicht gestartet wurden. Die Jobs aus der Queue werden gestartet, sobald ausreichend System-Ressourcen verfügbar sind.[26] Dabei ist wichtig zu erwähnen, dass das Queueable Interface nur eingesetzt werden sollte, wenn die aktuelle Transaktion nicht auf das Ergebnis des Auftrags angewiesen ist. Da der Queueable Auftrag als eine neue Transaktion gestartet wird, gibt es keine Möglichkeit zur vorherigen Transaktion zurückzukehren. So kann beispielsweise kein Queueable Auftrag gestartet werden und daraufhin ein Rückgabewert entgegen genommen werden.

Im Vergleich zum bereits besprochenen Batchable Interface bietet das Queueable Interface den Vorteil, dass Planung und Ausführung verhältnismäßig sehr kostengünstig sind. Bei der Ausführung einer Batch Klasse werden immer mindestens zwei Prozesse gestartet. Zuerst

muss die *start()*-Methode ausgeführt werden. Im Anschluss folgt ein neuer Prozess für die *execute()*-Methode.[23] Bei der Verarbeitung einer großen Menge an Datensätzen ist das vertretbar. Jedoch sollte bei wenigen Datensätzen das Queueable Interface gewählt werden, da dies einen Prozess einsparen kann und somit die Job Queue verkürzt.[23] Die automatisierten Prozesse, welche nach Anlage einer neuen Aktie gestartet werden müssen, bieten sich hier an, da dort in den meisten Fällen nur ein Datensatz verarbeitet werden muss.

Das Queueable Interface beinhaltet eine Methode namens *execute()*, die implementiert werden muss.[23] Sie wird beim Start der Ausführung aufgerufen und muss die gewünschte Logik enthalten. Aufrufen lassen sich die Queueable Aufträge mit der Methode *System.enqueueJob()*, welche eine Instanz der Klasse entgegennimmt, die das Queueable Interface implementiert.[23] Da die Klasse instanziiert wird, können Klassenvariablen definiert werden, welche dann über den Konstruktor befüllt werden können. Das ermöglicht die Übergabe von komplexen Datentypen an einen asynchronen Auftrag, was ohne dieses Interface noch nicht möglich war. Aus diesem Grund wurde sich für den Einsatz des Queueable Interfaces und gegen die sogenannten *@future*-Methoden entschieden. Dabei handelt es sich ebenfalls um asynchron aufrufbare Methoden, die jedoch nur primitive Datentypen entgegen nehmen können und sich zudem nicht verketten lassen. Listing 3.4 zeigt eine Klasse, die das Queueable Interface implementiert.

```
public class QueueableExample implements Queueable {
    List<Account> accountsToProcess;

    public QueueableExample(List<Account> accountsToProcess) {
        this.accountsToProcess = accountsToProcess;
    }

    public void execute(QueueableContext context) {
        // ...
    }
}

System.enqueueJob(new QueueableExample(accountsToProcess));
```

Listing 3.4: Queueable Interface

Der Aufruf der Methode *System.enqueueJob()* gibt die ID des Auftrags zurück. Damit ist es möglich sich Informationen, wie den Status oder die Fehleranzahl, über den Auftrag einzuholen. Erfolgen kann das über eine einfache SOQL-Abfrage des Objekts *AsyncApexJob*. Listing 3.5 zeigt eine mögliche SOQL-Abfrage. In der Variable "myJobID" wurde die ID gespeichert, welche beim Aufruf von *System.enqueueJob()* zurückgegeben wurde. Die Funktion kann genutzt werden, um beispielsweise den Auftrag nach einem Fehler erneut zu starten. Ein möglicher Fehler wäre der Fehlschlag eines Callouts an ein externes System. Sollte dieses System aus einem bestimmten Grund vorübergehend nicht erreichbar sein, kann der Auftrag nach einer gewissen Zeit neu gestartet werden und den Prozess wiederholen.[23]

```
AsyncApexJob info = [SELECT Status, NumberOfErrors
                     FROM AsyncApexJob
                     WHERE Id = :myJobID];
```

Listing 3.5: Abfrage eines asynchronen Auftrags

Ein weiteres nützliches Merkmal des Queueable Interfaces ist die Möglichkeit, mehrere Aufträge miteinander zu verketten. So kann ein Auftrag nach Fertigstellung eines anderen Auftrags aufgerufen werden, da dieser zum Beispiel auf das Ergebnis des ersten Auftrags angewiesen ist. Außerdem könnte komplexe Logik auf mehrere Klassen mit dem Queueable Interface verteilt werden, sodass jeder Auftrag ausschließlich für genau eine Sache zuständig ist. Dieses Vorgehen würde die Testbarkeit, Weiterentwicklung und Instandhaltung vereinfachen.

3.5 APEX REST

Salesforce bietet standardmäßig eine vollständige REST-API an, welche CRUD-Operationen für alle Objekte und weitere Funktionen bietet.[23] Sollten diese Funktionen allerdings nicht den Anforderungen entsprechen, kann auf das sogenannte *APEX REST* zurückgegriffen werden. Damit kann die REST-API ganz nach Belieben erweitert werden. Zudem ist es möglich, eine benutzerdefinierte Logik für externe Systeme zur Verfügung zu stellen und somit neue API-Endpunkte zu generieren. Einzelne Methoden einer Klasse können als diese neuen API-Endpunkte dienen und bei Aufruf die gewünschte Logik direkt ausführen.[17] So können beispielsweise Datensätze mehrerer unabhängiger Objekte mit einem Aufruf erstellt werden, Daten können vorverarbeitet werden oder komplexe Validierungen können durchgeführt werden, bevor ein Datensatz erstellt oder abgefragt wird.

Um das zu erreichen wird eine APEX Klasse als *@RestRessource* gekennzeichnet. Gleichzeitig muss ein sogenanntes URL-Mapping definiert werden. Dabei handelt es sich um einen relativen Pfad, der angibt, unter welcher URL die Methoden der Klasse aufgerufen werden können. Der Basis Pfad lautet: *https://instance.salesforce.com/services/apexrest/*. Daran wird der im URL-Mapping definierte Pfad angehängt. Die Methoden der Klasse müssen mit den HTTP-Verben gekennzeichnet werden.[27] Salesforce erlaubt diese HTTP-Verben:

- GET
- POST
- PUT
- PATCH
- DELETE

Jedes Verb darf pro Klasse nur einmal eingesetzt werden. Dies dient dazu, dass die Anfragen automatisch an die korrekte Methode geleitet werden können. So wird beispielsweise jede GET-Anfrage an den Endpunkt automatisch an die mit *@HttpGet* deklarierte Methode

geleitet. Somit muss diese nicht explizit angegeben werden. Die Syntax einer APEX REST Klasse wird in Listing 3.6 anhand eines Beispiels gezeigt.

```
@RestResource(urlMapping='/Example/*')
global with sharing class ExampleWebservice {

    @HttpPost
    global static void examplePost(String param1, String param2) {

    }

}
```

Listing 3.6: Beispielhafte Darstellung einer APEX REST Klasse

Am gezeigten Beispiel ist zu erkennen, dass die Klasse und die Methode als *global* kennzeichnet sind. Das ist Voraussetzung, um eine Klasse als REST-Endpunkt verfügbar zu machen. Eine Kennzeichnung mit *public* reicht dafür nicht aus. Außerdem ist im URL-Mapping ein Stern zu erkennen. Dabei handelt sich um einen Platzhalter. Dadurch werden alle Anfragen, die an „/Example/. . . “ gehen, an diese Klasse geleitet. Wichtig ist auch zu erwähnen, dass das URL-Mapping „case-sensitive“ ist, also wäre „/example“ nicht zulässig.[27]

Ebenfalls ist zu erkennen, dass die Methode zwei Parameter entgegen nimmt. Es gibt zwei Möglichkeiten, um auf die Daten des Aufrufs zugreifen zu können. Werden so wie hier Parameter definiert, wird Salesforce versuchen den Inhalt der HTTP-Anfrage in diese Parameter zu transferieren. Oftmals werden die Daten im sogenannten *JSON*-Format übertragen. JSON steht für *Java Script Object Notation* und ist ein Datenaustauschformat, auf dessen Verwendung sich viele Systeme für die Kommunikation von Daten geeinigt haben.[28] Dieses wird automatisch in die entsprechenden Datentypen umgewandelt und kann ohne einen zusätzlichen Schritt weiterverarbeitet werden. Die zweite Möglichkeit besteht darin keine Parameter zu definieren. In diesem Fall werden die Daten in *RestRequest.requestBody* gespeichert und können dort entnommen werden. Da für die Implementierung jedoch die erste Methode bevorzugt wird, wird an diesem Punkt nicht weiter auf die zweite Methode eingegangen.

APEX REST eignet sich perfekt für die Implementierung der Schnittstellen zur Erstellung von Depots, Käufen und Verkäufen.

3.6 APEX Trigger

Das sogenannte *APEX Trigger Framework* ist eine mächtige Komponente der Salesforce-Plattform. Übersetzt steht Trigger für Auslöser. Mit dem Framework ist es möglich auf die Events, die durch Änderungen an Datensätzen entstehen, mit APEX Code zu reagieren.[29] Diese Änderungen sind:

- Erstellung eines Datensatzes.

- Aktualisierung eines Datensatzes.
- Löschung eines Datensatzes.
- Wiederherstellung eines Datensatzes.

Somit ist es möglich bei all diesen Änderungen komplexe Logiken ausführen zu können. Dabei kann es sich um Berechnungen von Werten für bestimmte Felder, das Senden einer E-Mail beim Eintreten bestimmter Konditionen oder auch das Verhindern der Erstellung oder Anpassung eines Datensatzes handeln, sollten gewisse Bedingungen nicht erfüllt sein.

Ein Trigger kann zwischen diesen Events unterscheiden. Aus diesem Grund kann eine unterschiedliche Logik für jedes Event ausgeführt werden. Zusätzlich kann festgelegt werden, ob der Code vor oder nach Speicherung des Datensatzes ausgeführt werden soll. Diese Entscheidung sollte von der gewünschten Logik abhängig gemacht werden. Es ist ebenfalls möglich eine unterschiedliche Logik vor und nach dem Speichern des Datensatzes auszuführen.

Ein Trigger ist eine einfache APEX Datei, die mit den Stichwort *trigger* definiert wird. Es ist demnach keine Klasse. Die Definition des Triggers nimmt als Parameter die Events entgegen, auf die reagiert werden soll. Folgende Events sind möglich[29]:

- *before insert*
- *after insert*
- *before update*
- *after update*
- *before delete*
- *after delete*
- *after undelete*

Die genaue Bedeutung der Unterscheidung zwischen before und after wird im Abschnitt 3.11 über die *Order of Execution* besprochen. Ein Trigger wird immer für genau ein Objekt definiert. Er wird somit nur auf Events dieses Objekts reagieren. Theoretisch ist es möglich für ein Objekt mehrere Trigger zu definieren. Es ist aber ein bewährtes Verfahren pro Objekt nur einen Trigger zu besitzen.[23] Als Beispiel zeigt Listing 3.7 einen Trigger auf dem Account Objekt, der auf die Events *after insert* und *before update* reagiert.

```
trigger AccountTrigger on Account (after insert, before update) {

}
```

Listing 3.7: Trigger auf dem Objekt Account

Um mit einem Trigger auf mehrere Events reagieren und für die Events dennoch unterschiedliche Logik ausführen zu können, können die sogenannten *Trigger Kontext Variablen* eingesetzt werden. Diese sind nur innerhalb eines Triggers verfügbar und werden zur Laufzeit ausgewertet. Zugreifen kann man auf sie über die *System.Trigger*-Klasse. Zum einen gibt es Variablen zum Feststellen des Events, das den Trigger ausgelöst hat. Diese sind vom Typen Boolean und heißen *isInsert*, *isUpdate*, *isDelete* und *isUndelete*. Für das Leiten der Logik sind die zwei Variablen *isBefore* und *isAfter*, welche angeben, ob der Datensatz bereits gespeichert wurde oder nicht, ebenfalls wichtig. Die Datensätze selbst sind über mehrere Variablen verteilt, die nicht alle in allen Kontexten verfügbar sind. Generell gibt es *new*, *newMap*, *old* und *oldMap*. Die ersten zwei enthalten immer die neuen Datensätze, während die letzteren beiden den Zustand der Datensätze vor den Änderungen enthalten. Wird ein Datensatz aktualisiert, enthält *new* bereits die neuen Werte, während *old* noch den unveränderten Datensatz enthält. Die Variablen *newMap* und *oldMap* enthalten die identischen Daten wie *new* und *old*. Einzig ihre Struktur unterscheidet sich. Sie sind eine sogenannte *Map*, also eine Liste von Schlüssel-Werte-Paaren, wobei die ID des Datensatzes den Schlüssel darstellt.[30] Die Verwendung der Variablen ist in Listing 3.8 zu erkennen.

Ein bewährtes Verfahren besteht darin innerhalb eines Triggers auf jegliche Logik zu verzichten.[23] Im Trigger soll lediglich der Kontext geprüft werden, um anschließend die korrekten Variablen mit Datensätzen an die entsprechenden Methoden in einer Handler-Klasse zu übergeben. In der Handler-Klasse wird für jeden Kontext genau eine Methode erstellt, die auch nach dem Kontext benannt wird. Diese heißen dann beispielsweise *beforeInsert()* oder *afterUpdate()*. Innerhalb der Methoden kann im nächsten Schritt die gewünschte Logik ausgeführt werden, die bei Bedarf auch in eine Helper-Klasse ausgelagert werden kann.

```
trigger AccountTrigger on Account (
    before insert,
    after insert,
    before update,
    after update,
    before delete,
    after delete,
    after undelete
) {
    if (Trigger.isBefore && Trigger.isInsert) {
        // ...
    }

    if (Trigger.isBefore && Trigger.isUpdate) {
        // ...
    }

    if (Trigger.isAfter && Trigger.isInsert) {
        // ...
    }
```

```
    // ...
}
```

Listing 3.8: Struktur eines APEX Triggers

Zusätzlich ist weiterhin wichtig zu erwähnen, dass Trigger immer mit großen Mengen an Datensätzen umgehen können müssen. Sollten durch die Ausführung von Batch Klassen oder Daten Importen viele Datensätze verarbeitet werden, wird der Trigger mit einer Batch Größe von bis zu 200 Datensätzen gleichzeitig ausgeführt werden. Deshalb sollte immer mit Listen und anderen Kollektionen gearbeitet werden, damit die bereits besprochenen Governor Limits nicht überschritten werden. Ebenfalls sollten alle Datensätze, die sich in den Kontext Variablen befinden, beachtet werden. Nur so ist garantiert, dass alle betroffenen Datensätze durch die Logik verarbeitet werden.

In der Applikation dieser Arbeit kann ein Trigger eingesetzt werden, um beispielsweise den Prozess zur Abfrage der Aktienpreise und Unternehmensinformationen zu starten, nachdem eine neue Aktie angelegt wurde.

3.7 Flows

Die sogenannten *Flows* sind eine Möglichkeit auf der Salesforce-Plattform komplexe Automatisierungen, ohne die Verwendung von Code, zu implementieren. Das Ganze ist im sogenannten *Flow Builder* möglich. Das ist eine Art Entwicklungsumgebung, in der aus vorgefertigten Elementen verschiedene Pfade der Automatisierung zusammengestellt werden können. Es können Datensatz abgefragt, erstellt oder aktualisiert werden, Schleifen gebildet werden oder Entscheidungen zwischen verschiedenen Pfaden, welche mit einem *if-else-statement* vergleichbar sind, definiert werden.[31] Die Vorteile des Einsatzes von Flows sind eine oftmals einfachere Implementierung, die auch ohne einen erfahrenen Entwickler möglich ist und eine übersichtlichere Instandhaltung. Meistens ist auch die Migration auf andere Umgebungen problemloser. Es gibt verschiedene Arten von Flows. Alle zur Auswahl stehenden Arten sind in Abbildung 3.2 aufgeführt. Eine dieser sind sogenannte Bildschirm-Flows. Diese werden in der Regel durch den Klick auf einen Button ausgelöst. Durch diesen öffnet sich eine Art Dialog und der Benutzer kann damit interagieren. Eine zweite Art von Flows wird durch Änderungen an Datensätzen ausgelöst. Diese laufen im Hintergrund, wodurch sie Parallelen den bereits besprochenen APEX Triggern aufweisen. Diese werden in dem hier implementieren Projekt eingesetzt. Da ausgelöste Flows und APEX Trigger sich sehr ähneln, beziehungsweise mit beiden Möglichkeiten viele Anforderungen umgesetzt werden können, gibt es eine ausgeprägte Diskussion über die Entscheidung zwischen diesen beiden. Die komplette Besprechung über die Entscheidung zwischen dem Einsatz von Flow oder APEX übersteigt den Umfang dieser Arbeit. Aber aus den eben genannten Gründen sollte der Einsatz von Flows immer in Erwägung gezogen werden. Für komplexere Anwendungen können sogar APEX Methoden so deklariert werden, dass sie innerhalb eines Flows eingesetzt werden können.

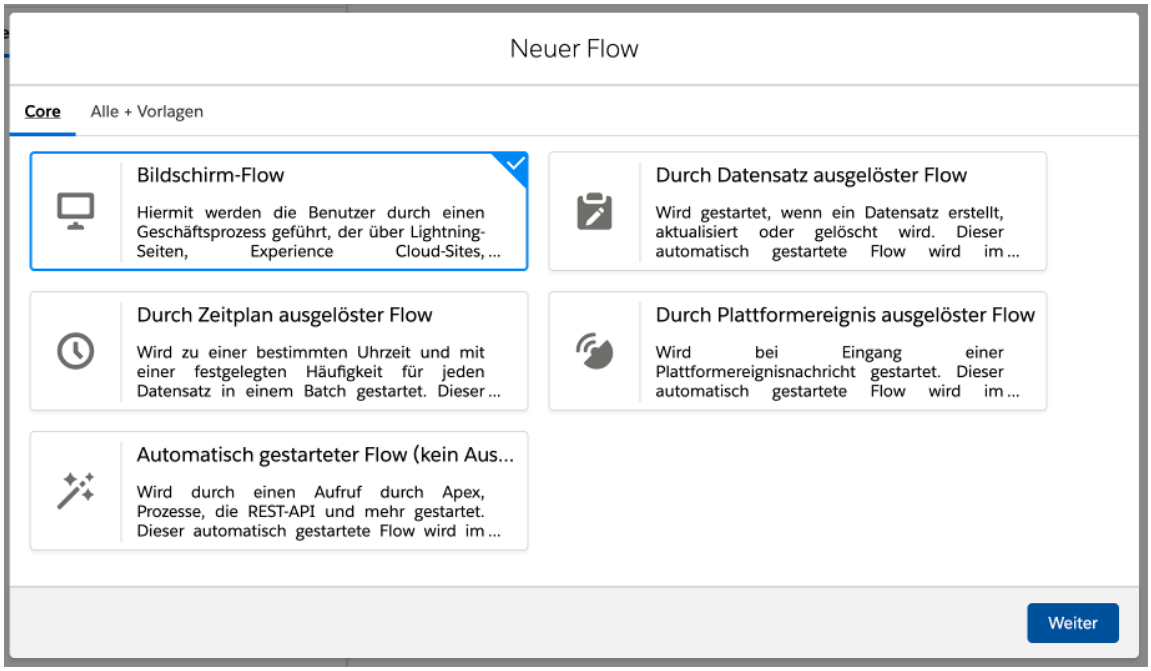

Abbildung 3.2: Verschiedene Arten von Flows

Ein einfaches Anwendungsbeispiel für einen Flow, der durch Änderungen an einem Datensatz ausgelöst wird, wäre beispielsweise:

Die Kunden des Online Shops „Example Shop“, dargestellt durch das Account Objekt, werden in die Kategorien Standard, Silber und Gold eingeteilt. Diese Einteilung geschieht aufgrund der Anzahl an Käufen, die sie bereits im Shop getätigt haben. Ab zehn Käufen ist der Silber Status erreicht. Tätigt ein Kunde mindestens 25 Käufe, erreicht er den Gold Status. Alle geringeren Kaufanzahlen gehören zum Standard Status. Die Käufe werden im Feld „Anzahl Käufe“ und der Status im gleichnamigen Feld festgehalten. Die Zuordnung des Status soll automatisch erfolgen.

Zur Umsetzung dieser Anforderung kann ein Flow erstellt werden, der durch die Aktualisierung eines Accounts ausgelöst wird. Er prüft den Wert im Feld „Anzahl Käufe“ und setzt dementsprechend das Feld „Status“. Hier macht der Einsatz eines Flow, welcher vor dem Speichern des Datensatzes ausgeführt wird, Sinn, da nur Felder auf dem Datensatz selbst abgeändert werden. Vor dem Speichern ist der Datensatz noch im lokalen Speicher geladen. Deshalb muss er nicht erneut aus der Datenbank abgefragt und danach wieder eingespielt werden. Auf die einzelnen Schritte der Erstellung eines Flows wird genauer in Abschnitt 4.7 eingegangen, wenn ein echter Flow für die Applikation entwickelt wird.

3.8 Connected Apps und OAuth

Die sogenannten *Connected Apps*, auf deutsch auch verbundene Anwendungen genannt, ermöglichen die Integration von externen Anwendungen mit Salesforce.[32] Connected Apps nutzen Standard Protokolle, wie beispielsweise OAuth, um Zugriff auf die Salesforce APIs zu gewährleisten.[32] Bevor also Abfragen über Daten, Anfragen zur Erstellung von Daten oder ähnliche Aufrufe getätigt werden können, muss sich die externe Anwendung mithilfe einer Connected App dazu autorisieren.

Durch die Aktivierung der OAuth-Einstellungen werden in der Connected App eine Client-ID und ein Client-Secret erzeugt. Diese sind notwendig, wenn das externe System sich das Zugangs-Token holen möchte. Um ein Zugangs-Token zu erhalten, muss eine Anfrage an den sogenannten *token*-Endpunkt gemacht werden. Dieser ist unter `https://login.salesforce.com/services/oauth2/token` erreichbar. Der Austausch des Tokens ist in Form von unterschiedlichen sogenannten *OAuth Authorization Flows* möglich. Darunter verstehen sich Abläufe, die für verschiedene Integrationen implementiert werden können. Während der Implementierung des Projekts dieser Arbeit und jetzt zur Veranschaulichung der Konzepte, wird der *Username-Password*-Flow eingesetzt.[33] Die Architektur der Applikation wird es dem Benutzer jedoch offenlassen, welche Art von OAuth Flow er einsetzen möchte.

Für den *Username-Password*-Flow werden folgende Daten benötigt:

- Client-Secret
- Client-ID
- Benutzername
- Passwort
- Sicherheitstoken

Für einen solchen Ablauf sollte immer ein entsprechender API-Benutzer angelegt werden, der genau die passenden Berechtigungen besitzt. Es ist nicht ratsam dafür einen Systemadministrator zu verwenden. Dieser hat vollen Zugriff auf alle Objekte und auch alle sonstigen Berechtigungen. Dies birgt die Gefahr, dass über die Integration ungewollte Änderungen durchgeführt werden.

Das Sicherheitstoken kann in den privaten Einstellungen des Benutzers ausgemacht werden. Es wird immer benötigt, wenn eine Anmeldung über die API durchgeführt werden soll. Das Token wird hinten an das Passwort gehängt. Bei einem Passwort „abc“ und einem Sicherheitstoken „123“, müsste bei der Anfrage „abc123“ als Passwort angegeben werden. Die Anfrage ist also eine POST-Anfrage an den *token*-Endpunkt, wobei alle eben aufgezählten Daten angegeben werden müssen. Zusätzlich muss *grant_type* auf „password“ gesetzt werden. Nur so weiß Salesforce, wie mit der Anfrage umzugehen ist. Abbildung 3.3 zeigt eine Abfrage zum Erhalt eines Zugangs-Tokens.

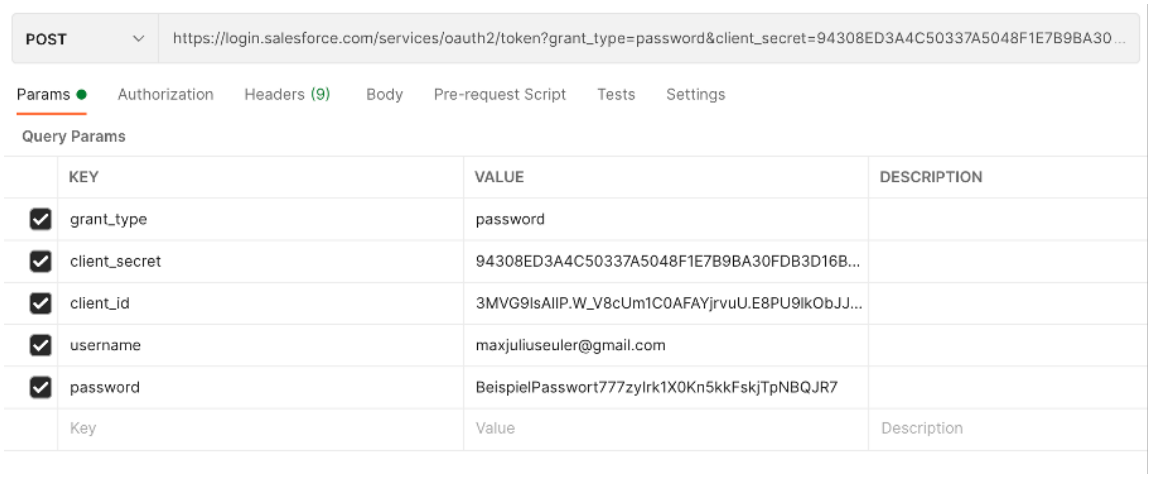

Abbildung 3.3: Benötigte Parameter für eine Token-Anfrage

Die Antwort beinhaltet zwei wichtige Parameter: Erstens das Zugangs-Token und zweitens die sogenannte Instanz-URL. Diese können Abbildung 3.4 entnommen werden. Der Zweck des Tokens wurde in diesem Kapitel bereits besprochen. Die Instanz-URL ist die URL, an welche alle weiteren Anfragen mit diesem Zugangs-Token gestellt werden müssen. Während die Anmeldung an den generischen `login.salesforce.com` Endpunkt gemacht werden konnte, ist das für alle weiteren Anfragen nicht mehr möglich. Die Instanz-URL ist die sogenannte *My Domain*. Diese ermöglicht es, die Salesforce URL nach Belieben anzupassen, damit sie beispielsweise den eigenen Firmennamen enthält. Jeder Saleforce Kunde kann sich dadurch seine eigene Subdomain erstellen.[34] Die URL der fiktiven Firma „Meier Logistik“ könnte `https://meierlogistik.my.salesforce.com` lauten. Zusätzlich kann durch den Einsatz von *My Domain* die Anmeldeseite angepasst und die Authentifizierung der Benutzer besser kontrolliert werden. Diese Themen sind für diese Arbeit allerdings nicht von Bedeutung, weshalb darauf nicht genauer eingegangen wird. Die übrigen Parameter werden im Zuge dieser Arbeit nicht benötigt.

3.9 SOQL Sicherheit

Bei den in diesem Projekt zu implementierenden APEX REST Klassen zur Anlage von Depots und Aktien werden SOQL-Abfragen durchgeführt werden müssen. Diese werden die übermittelten Parameter der Anfragen als Filter verwenden. An diesen Stellen ist besondere Aufmerksamkeit gefordert, da hier Schwachstellen des Systems für beispielsweise sogenannte *SOQL Injections* entstehen könnten. Diese Attacken sind eher als *SQL Injections* bekannt. Die eigene Datenbank Sprache der Plattform, SOQL, ist zwar im Gegensatz zu SQL eingeschränkter, aber auch hier sollten Implementierungen gegen solche Attacken geschützt werden. SOQL Injections können entstehen, wenn Eingaben von Nutzern direkt in eine dynamische Abfrage gegeben werden.[35] Dynamische Abfragen werden als String

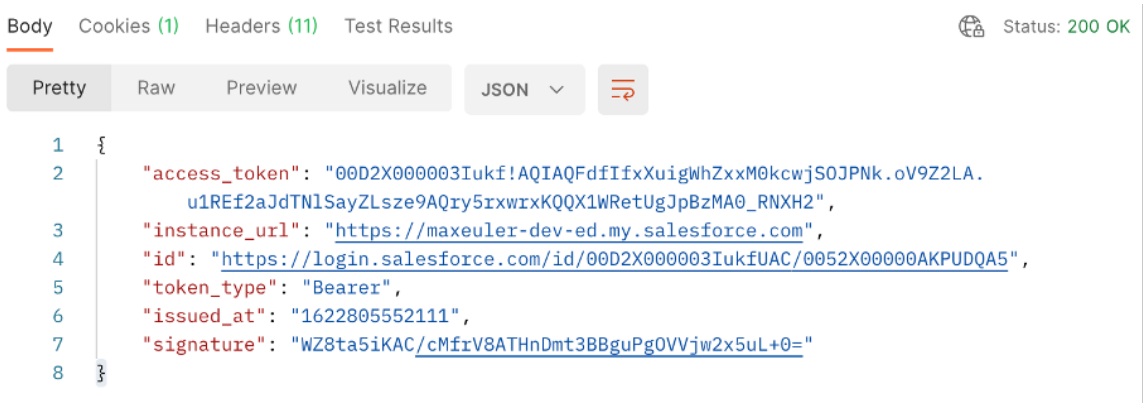

Abbildung 3.4: Übermittlung eines Zugang-Tokens

angelegt. Erst zur Laufzeit wird daraus eine Abfrage erstellt. Ein Beispiel dafür zeigt der Code in Listing 3.9. Die Methode nimmt einen String entgegen und setzt ihn direkt in eine SOQL-Abfrage ein.

```
public static void queryContacts(String lastName) {
    List<Contact> contacts = Database.query(
        'SELECT ID FROM Contact WHERE LastName = :lastName'
    );
}
```

Listing 3.9: Einfache dynamische SOQL-Abfrage

Oftmals wird diese Technik eingesetzt, um flexible Abfragen anhand von Nutzereingaben zu tätigen. Dabei besteht die Gefahr, dass durch bestimmte Eingaben zu viele oder ungewollte Datensätze abgefragt werden. Ist eine solche Abfrage beispielsweise Teil einer öffentlichen Webseite oder einer APEX REST Klasse, wäre die Preisgabe zu vieler Daten verheerend. Man nehme folgenden Code aus Listing 3.10 als Beispiel. Diese Abfrage gibt alle Kontakte zurück, welche als öffentlich markiert sind und einen passenden Nachnamen besitzen. Der Nachname wird von einem Benutzer eingegeben. Dies könnte beispielsweise über eine Form auf einer sogenannten *Site* geschehen. Dabei handelt es sich um öffentliche Seiten, die auf der Salesforce-Plattform entwickelt werden können und dann für Endkunden zugänglich gemacht werden.[36]

```
String queryString = 'SELECT ID FROM Contact WHERE ' +
    '(IsPublic__c = true AND LastName LIKE '%' + lastName + '%')';
return Database.query(queryString);
```

Listing 3.10: Anfällige dynamische SOQL-Abfrage

Würde nun der Wert „Mustermann%‘) OR (Name LIKE ‘“ (Hier sind die mit übergebenen Anführungszeichen zu beachten, welche Teil der Eingabe sind) als Eingabe für *lastName* verwendet werden, würde eine gefährliche Abfrage entstehen, die in Listing Listing 3.11

gezeigt wird.[35]

```
SELECT ID FROM Contact WHERE (IsPublic__c = true AND LastName LIKE '%Mustermann%')
OR (NAME LIKE '%')
```

Listing 3.11: Resultat einer SOQL-Injection

Alle Kontakte des Systems würden abgefragt und dadurch öffentlich verfügbar gemacht werden. Der Grund dafür ist das %-Symbol. Dieses ist ein Platzhalter, weshalb alle möglichen Namen eines Kontakts auf diese Abfrage zutreffen. Um dies zu verhindern wird bei der Implementierung der Applikation, mit Blick auf die APEX REST Klassen, auf statische SOQL-Abfragen in Kombination mit sogenannten Bindungsvariablen gesetzt. Mit Bindungsvariablen können flexible Werte in einer Abfrage ergänzt werden, ohne dass diese als ein ausführbares Element der Abfrage behandelt werden. Sie werden nur als eine Variable behandelt und der Angreifer kann somit nicht die Kontrolle über die Abfrage erlangen.[37] Die entsprechende Abwandlung der SOQL-Abfrage kann in Listing Listing 3.12 erkannt werden.

```
return [SELECT ID FROM Contact WHERE (IsPublic__c = true
        AND LastName LIKE :lastname)];
```

Listing 3.12: Verwendung einer Bindungsvariable

Die Eingabe, welche bei der dynamischen Abfrage alle Kontakte offenlegte, hätte bei dieser Syntax keine Wirkung. Die komplette Eingabe würde als ein einziger String behandelt werden. Somit würde nach Kontakten, deren Nachname „Mustermann%‘) OR (Name LIKE ‘“ lautet, gesucht werden.

Weitere Möglichkeiten zur Erhöhung der Sicherheit im Zusammenhang mit SOQL-Abfragen ist die Verwendung von speziellen Klauseln und Funktionen. Eine Möglichkeit wäre die Klausel *WITH SECURITY_ENFORCED*. Diese kann an jegliche SOQL-Abfragen innerhalb von APEX Code gehängt werden. Durch die Verwendung werden die Berechtigungen für Felder und Objekte des aktuellen Benutzers respektiert. Das bedeutet, dass nur die Daten abgefragt werden, auf die der Benutzer aufgrund seines Profils zugreifen darf. Eine detaillierte Besprechung von Profilen würde das Ausmaß dieser Arbeit übersteigen. Vereinfacht bedeutet dies, dass im Profil eines Benutzers festgelegt ist, welche Berechtigungen er auf der Salesforce-Plattform hat. Das betrifft die CRUD-Operationen für Objekte, den Zugriff auf bestimmte Seiten und Applikationen und Sicherheitsaspekte, wie die benötigte Mindestlänge eines Passworts. Ohne die entsprechenden Berechtigungen für ein gewisses Objekt, kann der Benutzer in der Applikation beispielsweise keine Datensätze für diese Objekt sehen.[38] Ein Profil kann zu mehreren Benutzern zugeordnet werden, die die gleichen Berechtigungen benötigen. Im Normalfall läuft APEX Code im sogenannten *system context*. Das bedeutet, dass während der Ausführung auf alle Daten im System zugegriffen werden darf und diese auch ohne Einschränkungen manipuliert werden dürfen. Durch den Einsatz von *WITH SECURITY_ENFORCED* ist garantiert, dass nur Objekte und Felder abgefragt werden können, die für den Benutzer zulässig sind. Listing Listing 3.13 zeigt den Einsatz der Klausel.

```
List<Contact> contacts = [SELECT ID FROM Contact WHERE Name LIKE 'Meier%' WITH
                          SECURITY_ENFORCED];
```

Listing 3.13: Objekt- und Feldberechtigungen bei SOQL

Neben *WITH SECURITY_ENFORCED* kann auch die Methode *Security.stripInaccessible()* eingesetzt werden. Diese Methode wird nicht direkt auf der SOQL-Abfrage angewandt, sondern arbeitet mit dem Ergebnis der Abfrage. Sie nimmt zwei Parameter entgegen. Zum einen die Art des Zugriffs, der geprüft werden soll. Dies muss in Form eines Werts des *AccessType* Enums sein. Möglich sind *CREATABLE*, *READABLE*, *UPDATABLE* und *UPSERTABLE*. Der zweite Parameter ist die Liste der Datensätze. Als Resultat erhält man ein Objekt vom Typen *SObjectAccessDecision*, von welchem man mit der Methode *getRecords()* die gefilterten Datensätze erhält. Aus dieser Liste wurden alle Felder entfernt, für die der Benutzer nicht die entsprechende Berechtigung hat. Die Methode eignet sich auch besonders gut, um alle unerwünschten Felder zu entfernen, bevor Datensätze erstellt oder aktualisiert werden sollen.[23]

Während der Implementierung der Applikation wird auf den Einsatz von *WITH SECURITY_ENFORCED* gesetzt. Dies hat den Vorteil, dass die Methode direkt in der Abfrage ergänzt werden kann und somit keine nachträgliche Filterung der abgefragten Datensätze notwendig ist.

3.10 Berichte

Mithilfe von Berichten können die Daten auf der Salesforce-Plattform abgefragt und ansprechend zur Verfügung gestellt werden. Im Grunde handelt es sich um Datenbank Abfragen, die auf der Benutzeroberfläche per *point-and-click* zusammengestellt werden und die daraus resultierenden Daten visuell darstellen können. Sie können in vielen verschiedenen Formaten erstellt werden, nur die gewünschten beziehungsweise benötigten Felder anzeigen und nach Belieben gefiltert werden.[39] Nach Erstellung kann ein solcher Bericht an verschiedenen Orten auf der Benutzeroberfläche platziert werden. Fügt man beispielsweise einen Bericht zu einer Datensatzseite hinzu, wird dieser automatisch nach dem betrachteten Datensatz gefiltert. Das ist für die Applikation dieser Arbeit von großer Bedeutung. Ein Bericht, der die Aufteilung eines Depots darstellt, soll selbstverständlich nur die Anteile des momentan betrachteten Depots zeigen.

Berichte basieren immer auf Berichtstypen. Diese definieren welche Objekte und Felder im Bericht verfügbar sind. Objekte in einem Berichtstyp müssen immer eine Beziehung zueinander besitzen. Viele Berichtstypen sind standardmäßig vorhanden, oftmals werden sie auch bei der Erstellung neuer Objekte automatisch angelegt. Für besondere Kombinationen muss allerdings manuell ein neuer Berichtstyp erstellt werden.[40] Dies kann einfach in den Einstellungen geschehen und ist somit ohne allzu großen Aufwand möglich.

Die einfachste Form an Berichten sind tabellarische Berichte, welche keinerlei Gruppierungen enthalten.[40] Hier geht es einzig darum die richtigen Datensätze mit den richtigen

Feldern darzustellen. Das ist vor allem hilfreich, wenn ein Objekt mit vielen Datensätzen und vielen Feldern vorliegt. Durch die Erstellung eines Berichts kann eine nützliche Übersicht erstellt werden. Zwei oft eingesetzte Formate sind „Summary“ und „Matrix“ Berichte. Im ersten Format werden die Zeilen gruppiert, während im letzteren die Daten nach Zeilen und Spalten gruppiert werden.[40] Beispiele wären die Anzeige aller Kunden gruppiert nach Umsatz oder die Anzeige aller Kunden gruppiert nach Umsatz und nach dem zuständigen Abteilungsleiter.

Zu einem Bericht kann auch immer, insofern es sich um einen Bericht im Format „Summary“ oder „Matrix“ handelt, ein Diagramm ergänzt werden. Ein solches Diagramm kann die Ergebnisse des Berichts in verschiedenen Formaten visualisieren. Liegt beispielsweise ein Bericht vor, welcher den Umsatz eines Unternehmens nach Monat gruppiert darstellt, würde sich ein Balken- oder ein Liniendiagramm eignen. In Hinblick auf die in dieser Arbeit implementierte Applikation würde ein Liniendiagramm zur Darstellung des Aktienpreises eine sinnvolle Visualisierung sein.

Ein weiterer wichtiger Aspekt sind die Sicherheitseinstellungen von Berichten. Berichte geben nur die Daten preis, auf die der Benutzer Zugriff hat.[41] Nehme man einen Bericht der Informationen über die Performance aller Depots darstellt. Depotverwalter dürfen aufgrund der Einstellungen im Unternehmen nur die Depots sehen, welche sie verwalten. Der Vorstand hingegen soll alle Depots sehen dürfen. Der erstellte Bericht wird diese Einstellungen berücksichtigen. So sieht der Vorstand im Bericht die Informationen zu allen Depots, während der Verwalter nur die Informationen bezüglich seiner Depots sieht.

3.11 Order of Execution

Die sogenannte *Order of Execution* ist ein Prinzip, das bei fast allen Implementierungen auf der Salesforce-Plattform beachtet werden muss. Da dieses Prinzip aber sehr umfangreich und komplex ist, werden in diesem Abschnitt nur die Punkte besprochen, die für diese Arbeit entscheidend sind. Mit der Order of Execution wird festgelegt, welche Events in welcher Reihenfolge ausgelöst werden, sobald ein Datensatz erstellt, aktualisiert oder gelöscht wird.[23] Da Salesforce zahlreiche Möglichkeiten für Automatisierungen, Validierungen und andere Operationen bietet, ist dieser Vorgang sehr umfangreich. Der entscheidende Punkt für diese Arbeit ist die Unterscheidung zwischen *before* und *after* Automationen in Form von APEX Triggern oder Flows. Während der Ausführung der Order of Execution gibt es einen Zeitpunkt an dem die Änderungen in der Datenbank gespeichert, aber noch nicht *commited* (auf deutsch übertragen), werden. In der kompletten Reihenfolge ist dies Punkt sieben von 18.[23] Alle vorherigen Events befinden sich im *before* Kontext und alle nachfolgenden demnach im *after* Kontext. Der Unterschied kann anhand eines Beispiels am verständlichsten erklärt werden:

Bei der Erstellung eines Datensatzes soll ein nummerisches Feld automatisch um eins erhöht werden. Geschieht dies im *before* Kontext, sind die Daten zum einen noch im Speicher geladen und zweitens noch nicht gespeichert. Deshalb kann die Änderung ausgeführt werden und wird ohne zusätzliche Schritte direkt mit übernommen. Die Änderungen werden direkt

am Datensatz im Speicher vorgenommen. Im *after* Kontext müsste der Datensatz noch einmal bewusst aktualisiert werden, indem *Database.update()* aufgerufen wird. Nur so würde die Änderung übernommen werden. Dies verursacht allerdings eine erneute Ausführung aller Events der Order of Execution. Dadurch werden die benötigte Zeit und Rechenkapazität deutlich erhöht. Der Mehraufwand einer Aktualisierung auf dem Datensatz, der die Automatisierung ausgelöst hat, wurde von Salesforce selbst untersucht. Das Ergebnis ist, dass der *before* Kontext dafür am besten geeignet ist. Die genauen Zahlen sind in Abbildung 3.5 zu erkennen. Die Grafik verdeutlicht, dass sich APEX Trigger und Flows im *before* Kontext am besten eignen. Die Entscheidung zwischen diesen beiden Möglichkeiten sollte anhand der Komplexität der Anforderungen getroffen werden. Außerdem sollte immer nur eines davon pro Objekt eingesetzt werden.

Weitere Szenarien, welche ebenfalls im *before* Kontext ausgeführt werden sollten, sind komplexe Validierungen oder die Befüllung von verpflichteten Feldern. Bei ersterem kann beispielsweise das Speichern eines Datensatzes verhindert werden, falls dieser nicht der gewünschten Datenqualität des Anwenders entspricht.

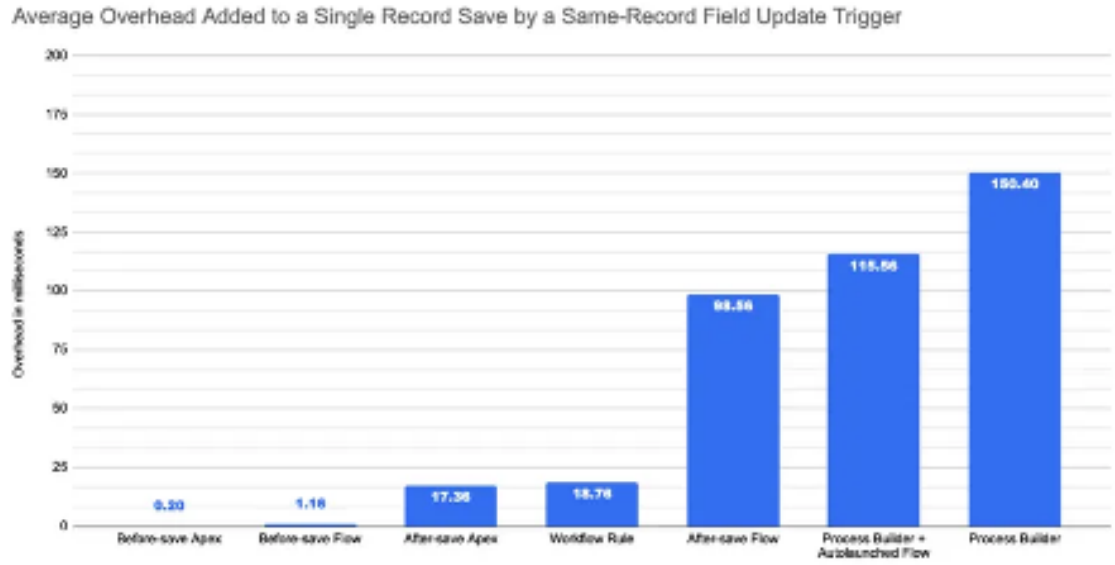

Abbildung 3.5: Mehraufwand bei verschiedenen Auslösern[42]

Anhand des oben genannten Beispiels lässt sich ableiten, in welchen Situationen die Entscheidung für welchen der beiden Kontexte fallen sollte. Generell kann festgehalten werden, dass alle Änderungen an dem Datensatz, der die Order of Execution ausgelöst hat, im *before* Kontext geschehen sollten. Ist nun aber die Anforderung verwandte Datensätze zu aktualisieren, ist die Logik im *after* Kontext besser aufgehoben. Bei der Applikation dieser Arbeit ist zum jetzigen Stand ein Flow geplant, der den Gesamtwert des Depots berechnen soll. Da dieser Flow auf Änderungen der Anteile reagieren und damit einen verwandten Datensatz (das Depot) aktualisieren wird, ist hier der Einsatz des *after* Kontextes passend. Der für die Applikation geplante APEX Trigger wird den asynchronen Prozess starten, wel-

cher nach Anlage einer Aktie ausgeführt werden muss. Da dieser Prozess für die aktuelle Transaktion des Triggers keine Rolle spielt, ist hier die Wahl des verwendeten Kontextes nicht entscheidend.

4 Implementierung

4.1 Entwicklungsumgebung

Zentrum der Entwicklung wird *Visual Studio Code*[1] in Kombination mit der *Salesforce CLI*[2] sein. Diese Tools werden von Salesforce empfohlen. Da die Ausführung von APEX Code und die Darstellung der verschiedenen Komponenten nur auf einem Salesforce Server möglich ist, wird eine sogenannte *Developer Edition Org* eingesetzt. Das sind Salesforce Umgebungen, welche schnell und kostenlos einsetzbar sind. Es bedarf lediglich einer Anmeldung bei Salesforce, nach dieser Salesforce die Umgebung innerhalb weniger Minuten zur Verfügung stellt. Zudem müssen alle Applikationen, die im AppExchange veröffentlicht werden sollen, auf solch einer Developer Edition Org entwickelt werden. Um eine Trennung zwischen der neuen Applikation und bereits vorhandener Logik in der Developer Edition Org zu ermöglichen, wird auf das Prinzip der sogenannten *Scratch Orgs* gesetzt. Das sind Umgebungen, die in wenigen Minuten über die Kommandozeile erstellt werden können. Sie existieren nur für einen begrenzten Zeitraum, welcher von Salesforce vorgegeben wird. Es werden keine Daten oder Metadaten aus der übergeordneten Developer Org übertragen. Sie sind aufgrund dessen komplett „leer“ und eignen sich deshalb perfekt für die Implementation einer neuen und alleinstehenden Applikation. Für die Versionskontrolle wird *GIT*[3] eingesetzt. Bei der Verwendung von Scratch Orgs ist es wichtig, dass diese nicht das führende System bilden. Das liegt zum einen daran, dass sie nur begrenzt verfügbar sind und nach Ablauf des Limits automatisch entfernt werden. Außerdem müssen alle Metadaten und Code-Dateien lokal vorliegen, um sie nach Fertigstellung der Applikation auf die Developer Org transferieren. Aus diesen Gründen wird das lokale GIT Repository immer die führende Stelle der Metadaten sein.

Der erste Schritt ist die Erstellung eines neuen Projekts über die Salesforce CLI. Damit wird die lokale Ordner-Struktur erstellt. Als nächstes muss eine Verbindung zur Developer Edition Org herstellt werden. Diese wird als sogenannter *Developer Hub* autorisiert. Der Developer Hub, kurz genannt auch Dev Hub, ermöglicht es, die benötigten Scratch Orgs für dieses Projekt zu erstellen und zu verwalten.[43] Scratch Orgs werden anhand einer JSON Datei erstellt, die automatisch während der Projekterstellung erzeugt wurde. Die standardmäßige Definition, bis auf den abgeänderten Namen, wird in Listing Listing 4.1 dargestellt. Die Datei kann erweitert oder angepasst werden, je nach Anforderungen an die zu erstellende Scratch Org.

[1] https://code.visualstudio.com/
[2] https://developer.salesforce.com/tools/sfdxcli
[3] https://git-scm.com/

```
{
  "orgName": "Depot Management",
  "edition": "Developer",
  "features": ["EnableSetPasswordInApi"],
  "settings": {
    "lightningExperienceSettings": {
      "enableS1DesktopEnabled": true
    },
    "mobileSettings": {
      "enableS1EncryptedStoragePref2": false
    }
  }
}
```

Listing 4.1: Definition einer Scratch Org

Durch die für Visual Studio Code verfügbaren Salesforce Ergänzungen lässt sich die Scratch Org ebenfalls schnell und einfach erstellen. Dafür muss ein Name und die gewünschte Haltbarkeit angegeben werden. Möglich ist ein Zeitraum zwischen einem und 30 Tagen. Nach nur wenigen Momenten ist sie einsatzbereit. Abbildung 4.1 zeigt die neu erstellte Umgebung.

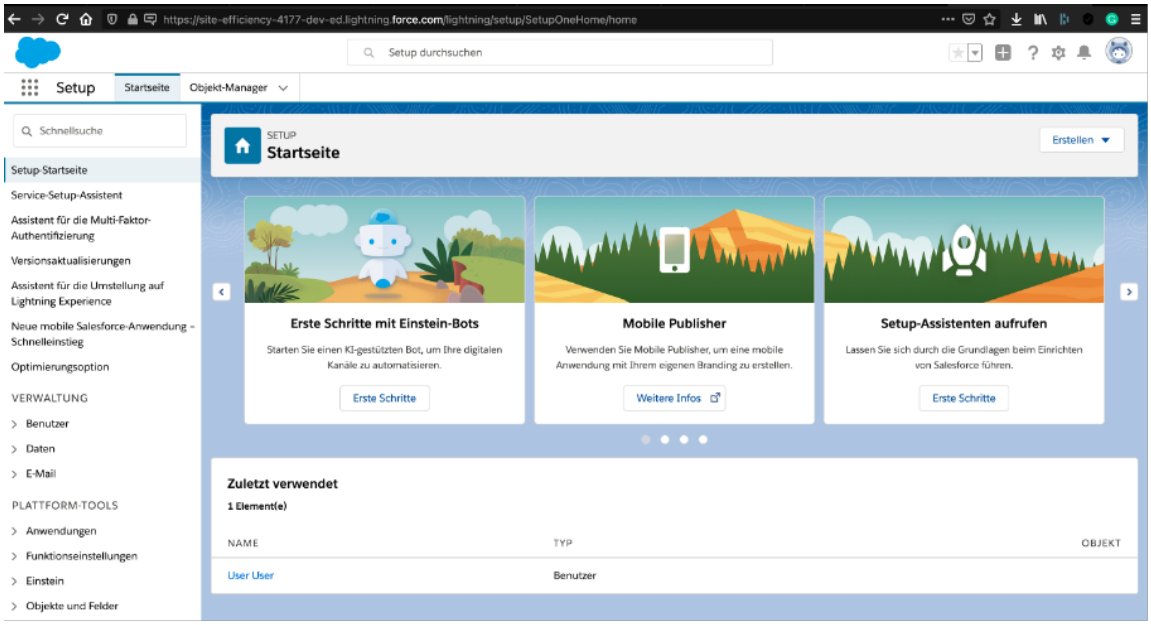

Abbildung 4.1: Setup-Menü der neuen Scratch Org

Der letzte Schritt des Setups ist die Initialisierung der Versionskontrolle. Dazu wird das in Visual Studio Code integrierte Terminal verwendet.

4.2 Aktien-Anlage

Der erste Teil der Implementierung wird sich mit den Aktien beschäftigen. Dafür muss das Datenmodell geplant und angelegt werden, außerdem die APEX REST Klasse zur Einspielung und Verarbeitung der Daten implementiert und eine ansprechende und nützliche Darstellung der Informationen gebaut werden.

Zu Beginn wird das Datenmodell benötigt. In Salesforce wird ein Datenmodell in Form von benutzerdefinierten Objekten erstellt. Auf diesen Objekten können einzelne Felder angelegt werden, welche später die Informationen speichern werden. Für die Darstellung aller Informationen werden zwei Objekte benötigt. Eines davon wird die Aktie selbst widerspiegeln. Das zweite Objekt beinhaltet die Preisinformationen. Da die Preise, wie in Abschnitt 2.2 besprochen, tagesaktuell sein und über den aktuellen Tag hinaus festgehalten werden sollen, muss dafür ein weiteres Objekt verwendet werden. Nur so kann die Wertentwicklung der Aktie dargestellt werden. Zwischen den Objekten wird eine Master-Detail-Beziehung erstellt. Diese verbindet N Tagespreise und eine Aktie eng miteinander. Diese Art von Beziehung eignet sich in Salesforce gut, wenn zwei Objekte eine enge und strikte Beziehung haben. Dieser Fall liegt hier vor. Jeder Preis muss immer zu genau einer Aktie gehören. Nach Anlage des Preises, wird sich die Aktie auch nicht ändern. Es würde fachlich gesehen keinen Sinn ergeben, einen vorhandenen Preis nachträglich zu einer anderen Aktie zuzuordnen.

Die Aktien werden anhand des Symbols eindeutig im System definiert. Die Aktie der Firma Apple hat beispielsweise das Symbol *AAPL*. Dieses Symbol darf nur einmal im System vorkommen, da jede Aktie nur einmal angelegt werden soll. Dafür wurde das Feld Symbol als eindeutig markiert. Salesforce wird die Erstellung von zwei Aktien mit identischem Symbol verhindern. Abbildung 4.2 zeigt einen Ausschnitt des Salesforce Schemagenerators. Dieser bietet eine grafische Darstellung des Datenmodells. In diesem Fall ist das Aktien-Objekt zu erkennen und dessen Beziehung zum Objekt Aktien Tagespreis.

Ein wichtiges Feld auf der Aktie ist „Aktueller Preis“. Dieses Feld wird immer den aktuellen Tagespreis speichern und muss demnach nach jeder Einspielung neuer Preis-Daten aktualisiert werden. Es ist wichtig, diesen Wert auf der Aktie selbst zu haben, um später das Depot richtig darstellen zu können.

Nachdem nun das Datenmodell vorhanden ist, muss die Automation zur Anlage der Aktien implementiert werden. Der Prozess wird folgendermaßen in drei Schritten ablaufen:

1. Anlegen eines Aktien Datensatzes mit dem Symbol der gewünschten Aktie.
2. Ausführen eines API-Aufrufs, um die Informationen über die Firma einzuholen.
3. Ausführen eines weiteren API-Aufrufs, um die Preis-Daten der Aktie für die letzten 100 Tage einzuholen. Somit kann direkt eine Historie dargestellt werden kann.

Da die Schritte zwei und drei hintereinander ausgeführt werden sollen, ohne dass der Benutzer handeln muss, wird hierfür das bereits besprochene *Queueable Interface* eingesetzt. Dieses Interface erlaubt es asynchrone Aufträge zu starten, welche dann in einer neuen und

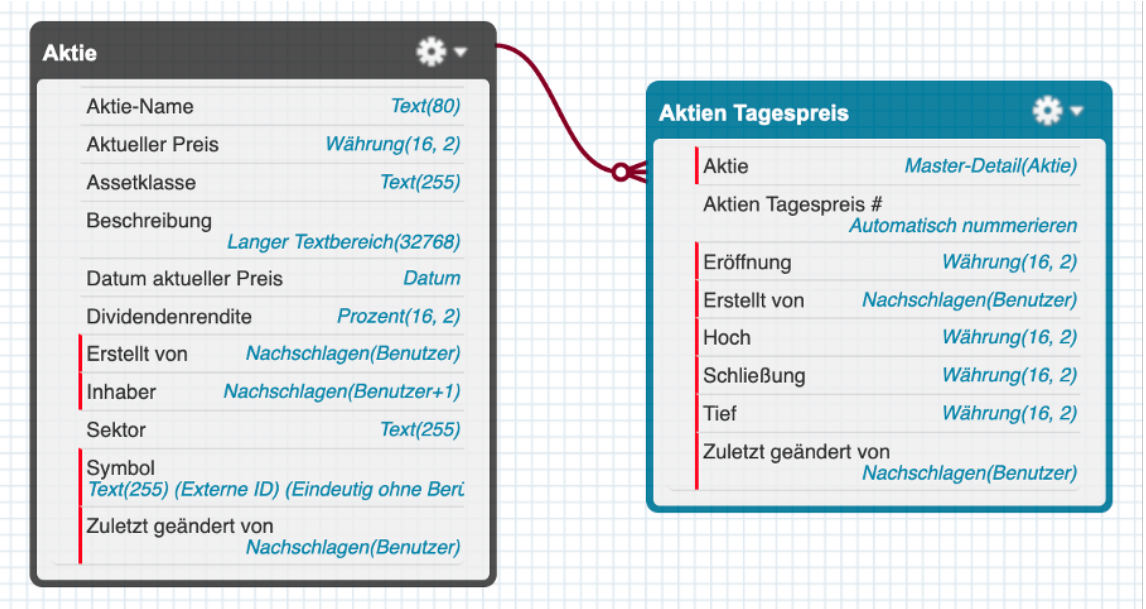

Abbildung 4.2: Das Aktien-Objekt mit den untergeordneten Aktien Tagespreisen.

unabhängigen Transaktion ausgeführt werden. Ebenfalls können mehrere Aufträge aneinander gekettet werden. Diese Aufträge laufen im Hintergrund, halten somit den Benutzer nicht auf und haben höhere Limits an System-Ressourcen. Da innerhalb der Klasse auch ein API-Aufruf durchgeführt werden soll, muss ebenfalls das Interface *Database.AllowsCallouts* implementiert werden. Nur mit diesem Interface gestattet Salesforce API-Aufrufe aus einer asynchronen Ausführung.[23] Für die Ausführung der Schritte zwei und drei wird nur eine einzige Klasse benötigt.

Queueables werden wie eine normale Klasse instanziiert, weshalb sie komplexe Datenstrukturen über den Konstruktor aufnehmen können. Diese Besonderheit stellt einen großen Vorteil zu anderen asynchronen Möglichkeiten auf der Salesforce-Plattform dar.[23] Für den Aktien Prozess werden drei Parameter an den Konstruktor übergeben, wie in Listing 4.2 zu erkennen ist.

```
public inherited sharing class ShareInitialProcess implements Queueable,
Database.AllowsCallouts {
    Boolean isFirstCallout;
    List<Share__c> shares;

    public ShareInitialProcess(List<Share__c> shares, Boolean isFirstCallout) {
        this.shares = shares;
        this.isFirstCallout = isFirstCallout;
    }
}
```

Listing 4.2: Queueable Konstruktor

Zuerst werden die neuen Aktien Datensätze übergeben, für welche die Informationen eingeholt werden sollen. Auch wenn es aufgrund der Architektur der Applikation in den aller meisten Fällen pro Transaktion immer nur ein Datensatz sein wird, sollte solch ein Code immer *bulkyfied* werden. Das bedeutet, dass der Code so implementiert wird, dass er auch mit einer großen Datenmenge umgehen kann, ohne die Limitationen der Ressourcen zu überschreiten.[23] Diese Klasse wird später aus dem Trigger-Kontext instanziiert. Was genau der Trigger-Kontext ist, wurde bereits in Abschnitt 3.6 besprochen. An dieser Stelle ist es aber noch einmal wichtig zu wissen, dass Trigger immer automatisch *bulkyfied* sind. Werden beispielsweise über die API oder den Data Loader, eine Desktopanwendung zum einspielen und auslesen von Datensätzen, 5000 Datensätze eingespielt, teilt Salesforce diese automatisch in kleine Pakete von 200 Datensätzen.[23] Für diese Pakete wird dann jeweils die Logik des jeweiligen Triggers ausgeführt. Der Code muss demnach immer mit 200 Datensätzen umgehen können. Ein Negativbeispiel ist eine SOQL-Abfrage innerhalb einer FOR-Schleife, da das Limit an SOQL-Abfragen pro Transaktion 100 beträgt.[22] Bei 200 Datensätzen würde dieses Limit überschritten werden und die Transaktion damit fehlschlagen.

```
for (Share__c share : Trigger.New) {
        List<Share__c> existngShare = [SELECT ID
                                       FROM Share__c
                                       WHERE Symbol__c = :share.Symbol__c
                                       WITH SECURITY_ENFORCED
                                       LIMIT 1];
        // Do something
}
```

Listing 4.3: Negatives Trigger-Beispiel

Der Code in Listing 4.3 soll prüfen, ob eine neu angelegte Aktie bereits im System vorhanden ist. Die Variable *Trigger.New* enthält alle Aktien Datensätze, die in dieser Transaktion erstellt wurden. Sollten dies über 100 Stück sein, wird das SOQL Limit erreicht und eine Limit Exception geworfen. Diese können nicht abgefangen werden, was zu einem Abbruch der Code-Ausführung führt. Aus diesem Grund sollte Code immer bulkyfied werden.

Der Parameter *isFirstCallout* im Konstruktor der Queueable Klasse gibt an, ob die Klasse zum ersten oder zum zweiten Mal ausgeführt wird. Die Klasse wird sich selbst aufrufen und damit den zweiten Auftrag starten. Beim zweiten Aufruf wird dieser Parameter auf *false* gesetzt, um den zweiten Teil der Logik ausführen zu können.

Der erste zu erledigende Schritt ist der HTTP-Callout zum Einlesen der Daten. Der Code dafür wird in einer extra Klasse definiert, da er später noch von anderen Klassen aufgerufen werden muss. Die Methode nimmt zwei Parameter entgegen:

- Function: Dieser Paramter gibt an, welche Informationen abgefragt werden sollen. Es können zwei Werte angenommen werden: *TIME_SERIES_DAILY_ADJUSTED* und *OVERVIEW*. Durch den ersten Wert werden die Preis-Daten der Aktie abgefragt,

während der zweite Wert für die allgemeinen Informationen über das Unternehmen steht.

- Symbol: Das Aktiensymbol für die gewünschte Aktie.

```
public static HttpResponse makeAlphaVantageCallout(String function, String symbol) {

    String calloutParams = '?function=' + function
                           + '&symbol=' + symbol
                           + '&apikey=' + System.Label.AlphaAPIKey;

    HttpRequest req = new HttpRequest();
    req.setEndpoint(System.Label.AlphaVantageAPIEndpoint + calloutParams);
    req.setMethod('GET');

    Http http = new Http();
    HttpResponse res = http.send(req);

    return res;
}
```

Listing 4.4: Methode mit HTTP-Callout

Die Methode, dargestellt in Listing 4.4, greift auf zwei sogenannte benutzerdefinierte Bezeichnungen zu. Dies geschieht durch *System.Label.AlphaVantageAPIEndpoint* und *System.Label.AlphaAPIKey*. Benutzerdefinierte Bezeichnungen lassen sich ganz einfach in den Einstellungen definieren. Sie erhalten einen Namen und einen bestimmten Wert. Dieser Wert kann danach in APEX Code referenziert werden.[44] Dies hat den Vorteil, dass keine hardkodierten Werte in der Klasse sichtbar sind. Es gibt drei Hauptgründe für diese Umsetzung:

1. Es handelt sich um sensible Daten, welche nicht für jeden mit Zugriff auf den Code sichtbar sein sollen. Ein API-Schlüssel wäre hierfür ein Beispiel. Während eines Code-Reviews kann das ganze Team den Code betrachten, ohne dass jeder den Schlüssel sehen kann.
2. Mehrere Sprachen sollen unterstützt werden.
3. Der Wert ändert sich in Zukunft gegebenenfalls. Sollte für diese Methode ein neuer API-Schlüssel eingesetzt werden müssen, da beispielsweise ein neues Paket dazu gebucht wird, müsste im Normalfall der Code angepasst und eine neue Version veröffentlicht werden. Mit benutzerdefinierten Bezeichnungen kann der neue Wert einfach in den Einstellungen aktualisiert werden. Das ist ebenfalls von Vorteil, wenn von mehreren Stellen auf den Wert zugegriffen wird, da er trotzdem nur an einer Stelle angepasst werden muss.

Die Methode kann genutzt werden, um die Historie der letzten 100 Tage an Preisdaten zu erhalten. Dafür wird in der Klasse eine weitere Methode ergänzt, welche in Listing 4.5

gezeigt wird. Diese nimmt das Symbol der Aktie entgegen und gibt die Preisdaten zurück.

```
public Map<String, Object> getDailySeries(String symbol) {
    HttpResponse res = ShareCalloutService.makeAlphaVantageCallout(
        'TIME_SERIES_DAILY_ADJUSTED',
        symbol
    );
    Map<String, Object> companyParams =
        (Map<String, Object>) JSON.deserializeUntyped(
            res.getBody()
    );
    Map<String, Object> seriesDaily = (Map<String, Object>) companyParams.get(
        `Time Series (Daily)`
    );

    return seriesDaily;
}
```

Listing 4.5: Abfragen der Preis-Daten

Diese Methode muss im nächsten Schritt für alle gewünschten Aktien aufgerufen werden. Der Aufruf wird in Listing 4.6 gezeigt. Da bei der API nur Informationen zu genau einer Aktie abgefragt werden können, muss die Methode für jede Aktie separat aufgerufen werden. Für jede Aktie gibt es eine Liste an Preisdaten. Hier ist das Limit an erlaubten Callouts pro Transaktion zu beachten. Dieses beträgt 100, weshalb die Klasse so eingesetzt werden sollte, dass pro Transaktion maximal 100 Aktien verarbeitet werden.[45] Da für jedes Element dieser Liste ein Datensatz erstellt werden muss, muss in der folgenden Methode auf eine verschachtelte FOR-Schleife zurückgegriffen werden. Dies ist hier akzeptabel, da die maximale Anzahl an Iterationen nach oben begrenzt ist und keine komplexen Anweisungen vorgenommen werden. Nachdem die Preisdaten abgefragt wurden, wird darüber iteriert und jeweils ein Datensatz vom Objekt Aktien Tagespreis erstellt. Währenddessen muss geprüft werden, ob es sich um den aktuellsten Preis handelt, da dieser auf der Aktie abgespeichert werden muss.

```
private void createShareDailyPrices() {
    ShareCalloutService helper = new ShareCalloutService();
    List<ShareDailyPrice__c> shareDailyPricesToInsert
        = new List<ShareDailyPrice__c>();
    Map<String,Share__c> sharesToUpdateBySymbol = new Map<String,Share__c>();

    for (Share__c share : this.shares) {
        Map<String, Object> dailySeries = helper.getDailySeries(share.Symbol__c);
        for (String dateValue : dailySeries.keySet()) {
            ShareDailyPrice__c ats = helper.getShareDailyPrice(
                (Map<String, Object>) dailySeries.get(dateValue),
                dateValue,
                share.Id
            );
```

```
            shareDailyPricesToInsert.add(ats);

            if (
                share.DateCurrentPrice__c == null ||
                share.DateCurrentPrice__c < ats.Date__c
            ) {
                Share__c updatedShare  = new Share__c(
                    Id = share.Id,
                    DateCurrentPrice__c = ats.Date__c,
                    CurrentPrice__c = ats.Close__c
                );
                sharesToUpdateBySymbol.put(share.Symbol__c, updatedShare);
            }
        }
    }

    if (sharesToUpdateBySymbol.values().size() > 0) {
        Database.update(sharesToUpdateBySymbol.values(), false);
    }
    Database.insert(shareDailyPricesToInsert, false);
}
```

Listing 4.6: Erstellung der Aktien Tagespreise

Damit wäre die Logik zum Abfragen und Einspielen der Preis-Historie abgeschlossen. Als nächstes geht es um die Abfrage weiterer Informationen über die Aktie und das Unternehmen. Dafür wird eine Methode implementiert, welche die Datensatz-ID der Aktie und ihr Symbol entgegennimmt. Mit Hilfe des Aktiensymbols wird der HTTP-Callout durchgeführt und die daraus gewonnenen Informationen werden über die Datensatz-ID auf die korrekte Aktie geschrieben. Listing 4.7 zeigt die dafür implementierte Methode. Hierbei war die Generierung des passenden Datentypen zu beachten, da durch die Deserialisierung von JSON eine Map mit Werten vom Typ *Object* entsteht. Um daraus beispielsweise den Dezimalwert zu bekommen, muss der Wert erst zu einem String umgewandelt werden, bevor er zu einer Dezimalzahl weiterverarbeitet werden kann.

```
public Share__c getCompanyInformation(ID shareId, String symbol) {
    HttpResponse res = ShareCalloutService.makeAlphaVantageCallout(
        'OVERVIEW',
        symbol
    );

    Map<String, Object> companyParams
        = (Map<String, Object>) JSON.deserializeUntyped(res.getBody());
    Share__c share = new Share__c();
    share.ID = shareId;
    share.Assettype__c    = String.valueOf(companyParams.get('AssetType'));
    share.Name  = String.valueOf(companyParams.get('Name'));
    share.Description__c  = String.valueOf(companyParams.get('Description'));
    share.Sector__c  = String.valueOf(companyParams.get('Sector'));
```

```
    share.Dividendyield__c  = Decimal.valueOf(
        String.valueOf(companyParams.get('DividendYield'))
    ) * 100;
    return share;
}
```

Listing 4.7: Abfragen und Verarbeiten der Unternehmens-Informationen

Diese Methode muss anschließend für alle Aktien innerhalb des zweiten Durchlaufs der Queueable Klasse aufgerufen werden. Da alle Aktien als Klassenvariable übergeben werden, muss über diese iteriert werden. Damit ist der Code, welcher zur Erstellung und initialen Befüllung der Aktien benötigt wird, vollständig. Der initiale Prozess wird nach Erstellung einer Aktie automatisch aufgerufen. Während des ersten Aufrufs, welcher anhand der Variable *isFirstCallout* erkannt werden kann, wird die bereits bekannte Methode zur Abfrage und Erstellung der Aktien Tagespreise aufgerufen. Anschließend ruft der Prozess sich selbst erneut auf, jedoch wird dieses Mal *isFirstCallout* negiert. Die entscheidende Methode *execute()* leitet den Kontrollfluss so, dass die Informationen über das Unternehmen abgefragt und ergänzt werden. Die finale Klasse, ohne die bereits aufgeführte *createShareDailyPrices()*-Methode, wird in Listing 4.8 dargestellt.

```
public inherited sharing class ShareInitialProcess implements Queueable,
Database.AllowsCallouts {
    Boolean isFirstCallout;
    List<Share__c> shares;

    public ShareInitialProcess(List<Share__c> shares, Boolean isFirstCallout) {
        this.shares = shares;
        this.isFirstCallout = isFirstCallout;
    }

    public void execute(QueueableContext context) {
        if (this.isFirstCallout) {
            this.createShareDailyPrices();
            System.enqueueJob(new ShareInitialProcess(this.shares, false));
        } else {
            this.fillShareRecord();
        }
    }

    private void fillShareRecord() {
        ShareCalloutService helper = new ShareCalloutService();
        List<Share__c> sharesToUpdate = new List<Share__c>();
        for (Share__c share : this.shares) {
            sharesToUpdate.add(
                helper.getCompanyInformation(share.Id, share.Symbol__c)
            );
        }
        Database.update(sharesToUpdate, false);
    }
```

```
    private void createShareDailyPrices() {
        // ...
    }
}
```

Listing 4.8: Asynchroner Prozess zur Aktien Anlage

Den letzten Schritt stellt das Aufrufen der Logik zum korrekten Zeitpunkt dar. Dafür wird ein APEX Trigger implementiert, wie in Listing 4.9 dargestellt. Die Funktionsweise eines Triggers wurde bereits im Abschnitt 3.6 besprochen. In diesem Fall ist der *after insert* Kontext am sinnvollsten.

- *insert*: Der initiale Prozess soll ausschließlich bei Anlage von neuen Aktien gestartet werden, deshalb ist *insert* hier die einzig richtige Möglichkeit.
- *after*: Da es sich hier ausschließlich um den Aufruf eines asynchronen Prozesses handelt, spielt die Entscheidung zwischen *before* und *after* nur eine untergeordnete Rolle. Die Wahl viel letztendlich auf *after*, da der Prozess für den aktuellen Speichervorgang nicht entscheidend ist. Generell wird der *before* Kontext nur genutzt, wenn der Datensatz manipuliert werden soll, während er sich noch im Speicher befindet.

Ein Flow ist nicht in der Lage einen asynchronen Prozess in Form einer Queueable Klasse zu starten, weshalb sich hier nicht für einen *before save* oder *after save* Flow entschieden wurde. Theoretisch wäre dies möglich, dafür müsste jedoch eine sogenannte *Invocable Method* definiert werden. Dabei handelt es sich um eine APEX Methode, welche dann von einem Flow aus aufgerufen werden kann.[46] Allerdings werden am Ende ein Flow und zusätzlich noch APEX Code benötigt, weshalb sich hier für eine reine APEX Lösung entschieden wurde.

```
trigger ShareTrigger on Share__c (after insert) {

    if (Trigger.isAfter && Trigger.isInsert) {
        System.enqueueJob(new ShareInitialProcess(Trigger.new, true));
    }

}
```

Listing 4.9: Trigger auf dem Objekt Aktie

Der benötigte Trigger besteht aus wenigen Zeilen Code. In Zeile eins wird definiert, auf welches Objekt und welches Event reagiert werden soll. In diesem Fall auf das Aktien Objekt, welches den API-Namen „Share___c“ besitzt. Der Code soll *after insert* ausgeführt werden. Im Trigger wird das Event noch einmal geprüft, falls der Code einmal erweitert werden sollte. Danach wird der erste Job gestartet. Als Parameter werden die neuen Aktien Datensätze, welche sich in der Kontext Variable *Trigger.new* befinden, und der Wert *true* übergeben. Durch das *true* wird der Queueable Klasse signalisiert, dass es sich um den ersten Durchlauf handelt.

Da bis zu diesem Punkt der Arbeit die gesamte Logik für die Aktien fertig ist, muss anschließend für eine ansprechende Darstellung gesorgt werden. Eine wichtige Komponente dabei ist die grafische Darstellung des Aktien-Kurses. Für diese wird ein Bericht erstellt. Berichte dienen in Salesforce dazu, Informationen zu kombinieren, gruppieren, analysieren und in einem gewünschten Format darzustellen. Zusätzlich können verschiedene Arten von Diagrammen hinzugefügt werden. Die Daten werden nach Datum gruppiert, um so den Kurs für jeden Börsen-Tag anzeigen zu können. Das Thema Berichte wurde im Detail bereits in Abschnitt 3.10 detailliert besprochen. Abbildung 4.3 zeigt die Erstellung des Berichts. Die grundlegenden Daten sind die Aktien Tagespreise. Auf der linken Seite kann unter „Zeilen gruppieren“ erkannt werden, dass die Datensätze nach dem Datum gruppiert werden. Für die anzuzeigenden Werte wird lediglich das Feld „Schließung“ gewählt, welches den Preis zum Börsenschluss enthält. Die daraus resultierenden Daten sind in der unteren Hälfte der Abbildung in Tabellenform zu erkennen. Für die grafische Darstellung wird ein Diagramm hinzugefügt, welches die Werte automatisch als Linie darstellt.

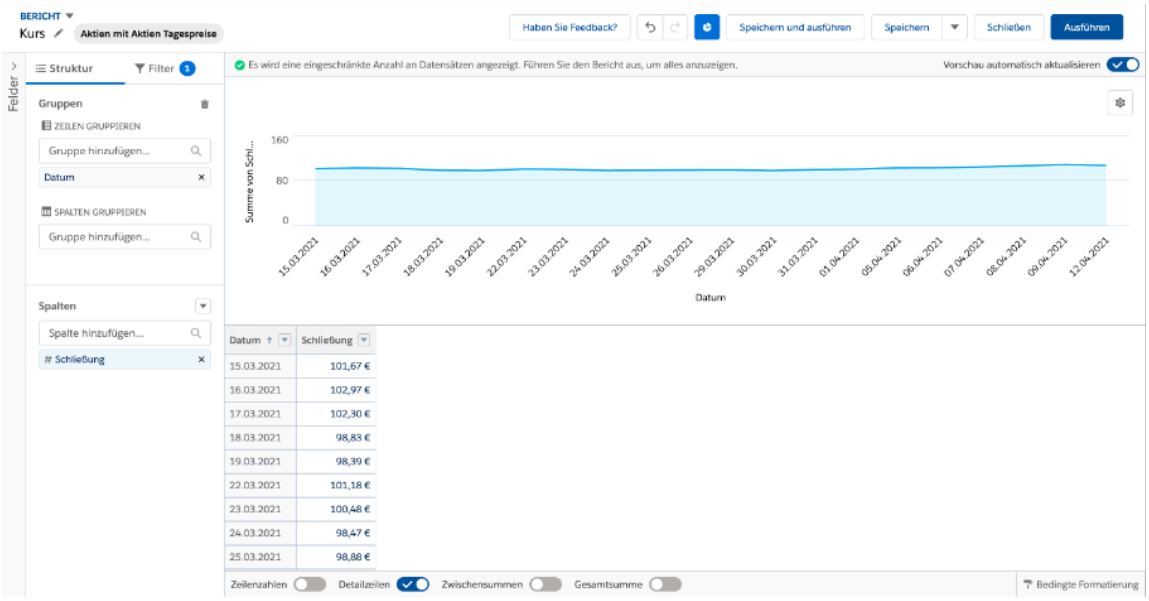

Abbildung 4.3: Erstellung des Aktien-Berichts

Damit ist das Kapitel der Aktien Anlange vollständig. Angelegt werden die Aktien über die Angabe des Aktiensymbols. Im Hintergrund wird der implementierte Trigger die Queueable Klasse aufrufen. Dort werden die zwei HTTP-Callouts getätigt und die Aktie befüllt. Nach Anlage der Aktie sind die Daten eingespielt und der Kurs kann von einem ansprechenden Linien-Diagramm abgelesen werden.

4.3 Aktien-Pflege

Da nun die Prozesse zur Anlage und Befüllung der Aktien und den initialen Preisdaten vollständig umgesetzt sind, muss als nächstes die fortlaufende Aktualisierung der Aktien implementiert werden. Das bedeutet, dass täglich die aktuellen Preise abgefragt und abgespeichert werden müssen. Dieser Prozess wird in Form einer Batch Klasse ausgeführt. Listing 4.10 zeigt die dafür zu implementierende, aber noch leere Klasse. Lediglich die drei Methoden des *Batchable* Interfaces sind bereits vorhanden. In der *start()*-Methode wurde die SOQL-Abfrage definiert. Es werden alle Aktien-Datensätze mit den Feldern *Symbol* und *Datum aktueller Preis* abgefragt. Letzteres wird benötigt, um den aktuellsten Tagespreis ausfindig machen zu können. Es sei an dieser Stelle zu erwähnen, dass in APEX Code immer die API-Namen der Felder verwendet werden müssen. *Datum aktueller Preis* ist beispielsweise nur die Bezeichnung des Feldes, die auch Leerzeichen oder Sonderzeichen enthalten darf. Für die eindeutige und strikte Kennzeichnung eines Feldes ist der API-Name zuständig. In diesem Fall ist das *DateCurrentPrice___c*.

```
public inherited sharing class ShareDailyPriceBatch implements
Database.Batchable<SObject>, Database.AllowsCallouts {

    @SuppressWarnings('PMD.EmptyStatementBlock')
    public ShareDailyPriceBatch() {}

    public Database.QueryLocator start(Database.BatchableContext batchContext){
        return Database.getQueryLocator([SELECT ID, DateCurrentPrice__c, Symbol__c
                                         FROM Share__c]);
    }

    public void execute(
        Database.BatchableContext batchContext,
        List<Share__c> scope
    ) {

    }

    @SuppressWarnings('PMD.EmptyStatementBlock')
    public void finish(Database.BatchableContext batchContext){}
}
```

Listing 4.10: Batch Klasse zum Einlesen der Aktien Tagespreise

Der sogenannte *scope* der Klasse besteht aus allen Aktien-Datensätzen im System. Für alle wird der neue Preis benötigt. Folgende Schritte müssen für jede Aktie durchgeführt werden:

1. Abfrage der Preis-Daten.
2. Herausfiltern des neusten Preises.

3. Prüfen, ob für dieses Datum bereits ein Preis abgespeichert ist.
4. Einen Aktien Tagespreis Datensatz erstellen und diesen mit der Aktie verbinden.
5. Den neuen Preis und das Datum auf der Aktie speichern.

Ein wichtiger Schritt, der zuerst erledigt werden muss, ist das Herausfinden des vorherigen Wochentages. Grundlegend sollen immer die Preise des Vortags abgefragt werden. An Montagen hingegen muss der Preis des vergangenen Freitags, nicht des Sonntags, abgefragt werden. Dafür wird eine Hilfsmethode implementiert, die mit den sogenannten Geschäftszeiten arbeitet und auf Rekursion setzt. Normalerweise werden die Geschäftszeiten für bestimmte Support-Prozesse in Salesforce verwendet, dennoch sind diese aber auch für diesen Zweck einsetzbar. Man definiert mit ihnen zu welchen Zeiten beispielsweise das Support Team verfügbar ist.[47] Im jetzigen Kontext sollen sie die Handelstage an der Börse, also Montag bis Freitag, repräsentieren. Abbildung 4.4 zeigt die dafür definierten Geschäftszeiten. Da nur der Tag und nicht die Uhrzeit entscheidend ist, wird jeweils ein 24 Stunden Zeitraum gewählt.

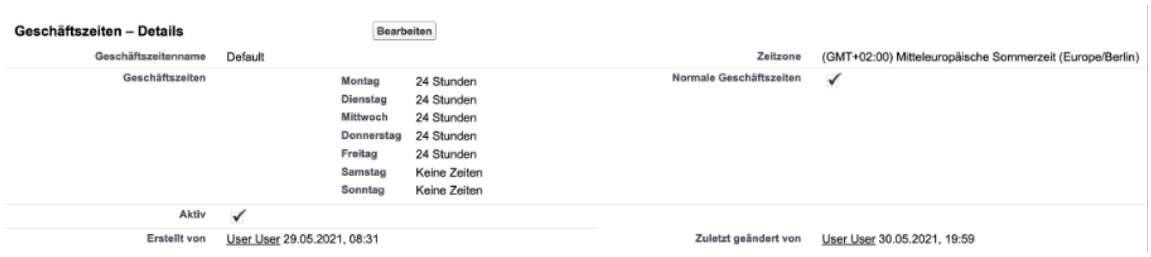

Abbildung 4.4: Geschäftszeiten zur Abfrage der Preis-Daten

Die Hilfsmethode in Listing 4.11 prüft, ob der vorherige Tag innerhalb der Geschäftszeiten liegt. Ist dies nicht der Fall, ruft sie sich selbst noch einmal mit dem vorherigen Tag auf.

```
private Datetime getPreviousWorkingDay(Datetime d){
    return !BusinessHours.isWithin(this.stockExchangeBusinessHours.Id, d)
            ? getPreviousWorkingDay(d.addDays(-1))
            : d;
}
```

Listing 4.11: Feststellen des letzten Wochentags

Die *execute()*-Methode der Klasse ist das Herzstück. Sie wird automatisch für jeden Block an Daten ausgeführt. Hier müssen die weiter oben aufgezählten fünf Schritte nun ausgeführt werden. Der HTTP-Callout ist identisch zu dem Callout, der nach Anlage der Aktien die Preise abfragt. Diesmal wird von der Antwort jedoch nur der aktuelle Preis verarbeitet. Zuallererst muss geprüft werden, ob die Batch Klasse zur richtigen Zeit aufgerufen wurde. Dafür wird während der Iteration über die Aktien geprüft, ob das aktuell auf der Aktie gespeicherte Datum mit dem Datum, für welches der Preis eingelesen werden soll,

übereinstimmt. Falls ja, kann diese Aktie übersprungen werden, da der Preis bereits vorhanden ist. Somit soll verhindert werden, dass der gleiche Tagespreis mehrfach im System vorkommt. Diese Situation könnte eintreten, falls die Batch Klasse mehrfach am gleichen Tag ausgeführt werden sollte.

Nach der Prüfung können die Preisdaten abgefragt und der passende Preis aus der Liste gefiltert werden. Diese werden unter dem Schlüssel „Time Series (Daily)“ geliefert und sind dort in einer Liste von Schlüssel-Werte-Paaren angeordnet. Dabei ist das Datum der Schlüssel. Dieses ist im Format „YYYY-MM-DD“, beispielsweise „2021-07-21“. Mit den korrekten Preis-Daten wird nun ein Aktien Tagespreis Datensatz erstellt und der Preis wird auf der Aktie aktualisiert.

Um die Preise nun auf einem aktuellen Stand halten zu können, muss die neue Batch Klasse mit Hilfe des *Schedulable* Interface automatisch gestartet werden. Dafür wird eine zweite Klasse, dargestellt in Listing 4.12, implementiert und diese wird nur eine Aufgabe haben: Den Aufruf der Batch Klasse. Durch die Implementierung des *Schedulable* Interfaces ist es möglich diese Klasse zu planen, damit sie automatisch zu bestimmten Zeitpunkten ausgeführt wird.

```
global inherited sharing class ShareDailyPriceBatchScheduler implements
Schedulable {
    global void execute(SchedulableContext sc) {
        ShareDailyPriceBatch dailyPriceBatch = new ShareDailyPriceBatch();
        Database.executeBatch(dailyPriceBatch, 200);
    }
}
```

Listing 4.12: Klasse mit Schedulable Interface

Damit sind die Logik und der Prozess zur Pflege der Aktien im System vollständig. Als Resultat werden jeden Morgen für alle Aktien die neuen Preise automatisch abgefragt und im System hinterlegt.

4.4 Depot-Anlage

Nachdem die Aktien nun angelegt sind und die Preise gepflegt werden, ist der nächste Schritt die Anlage der Depots. Das dazugehörige Datenmodell wurde bereits in Abschnitt 2.4 besprochen. Zur Anlage wird eine APEX REST Klasse implementiert, deren Methoden für externe Anwendungen verfügbar gemacht werden. Dafür wird eine APEX Klasse erstellt, welche als *@RestResource* deklariert wird, ein *URL-Mapping* erhält und *global* verfügbar gemacht wird. Darin wird die einzig benötigte Methode zur Anlage eines neuen Depots implementiert. Wie bereits besprochen, wird diese Methode eine Depotnummer und die E-Mail-Adresse des Inhabers entgegennehmen. Beide sind eindeutig im System und reichen deshalb als Informationen aus.

Bei jeder POST-Anfrage an den Endpunkt wird automatisch die Methode *createDepot* aufgerufen und muss dafür nicht spezifisch angegeben werden. Da *createDepot* immer einen neuen Datensatz erstellen wird, ist HTTP-POST das passende Verb. Innerhalb der Methode müssen folgende Schritte ausgeführt werden:

1. Prüfen, ob die Parameter mitgeliefert wurden.
2. Prüfen, ob ein Kunde mit der gelieferten E-Mail-Adresse im System existiert.
3. Sicherstellen, dass dieser Kunde noch kein Depot besitzt.
4. Erstellen des neuen Depots.

Während für Schritt eins eine einfache String-Methode ausreicht, müssen für Schritt zwei und drei SOQL-Abfragen erstellt werden. Da die Parameter der Methode als Filter verwendet werden, ist es wichtig auf die Sicherheit zu achten. Dafür wird eine statische Abfrage mit Bindungsvariablen eingesetzt. Die Gründe dafür wurden bereits in Abschnitt 3.9 besprochen. Wurden alle Prüfungen durchgeführt, kann das Depot unter Angabe der Nummer und der ID des Inhabers angelegt werden. Die vollständige Klasse wird in Listing 4.13 aufgeführt.

```
@RestResource(urlMapping='/Depot/*')
global with sharing class DepotWebservice {

    @HttpPost
    global static String createDepot(String ownerMail, String depotNumber) {

        if (String.isBlank(ownerMail)) {
            return 'No mail address provided!';
        }
        if (String.isBlank(depotNumber)) {
            return 'No depot number provided!';
        }

        List<Contact> depotOwner = [SELECT ID
                                    FROM Contact
                                    WHERE Email = :ownerMail
                                    WITH SECURITY_ENFORCED
                                    LIMIT 1];
        if (depotOwner.size() == 0) {
            return 'No customer with this mail address present!';
        }

        List<Depot__c> existingDepot = [SELECT ID
                                        FROM Depot__c
                                        WHERE Owner__r.Email = :ownerMail
                                        WITH SECURITY_ENFORCED
                                        LIMIT 1];
        if (existingDepot.size() > 0) {
            return 'A depot already exists for this customer!';
```

```
        }

        Depot__c newDepot = new Depot__c(
            DepotNr__c = depotNumber,
            Owner__c = depotOwner[0].Id
        );

        Database.SaveResult saveResult = Database.insert(newDepot);

        if (!saveResult.isSuccess()) {
            return 'The following error has occurred: ' + saveResult.getErrors();
        }

        return 'Successfully created depot.';
    }

}
```

Listing 4.13: APEX REST zur Depoterstellung

Um die APEX REST Klasse später im richtigen Kontext testen und nutzen zu können, wird ein aufwendigerer Test mit einem externen Programm zur Simualtion von API-Aufrufen durchgeführt werden müssen. Zum jetzigen Zeitpunkt reicht ein einfacher Test der Logik über die Entwicklerkonsole. Dabei wird lediglich die Methode aufgerufen und geprüft, ob der gewünschte Datensatz erstellt wird. Die Methode wird mit der E-Mail-Adresse eines vorhandenen Kontaktes aufgerufen und der Rückgabewert soll auf der Konsole ausgeben werden. Abbildung 4.5 und Abbildung 4.6 stellen diesen Vorgang dar.

Abbildung 4.5: Aufrufen der APEX REST Klasse

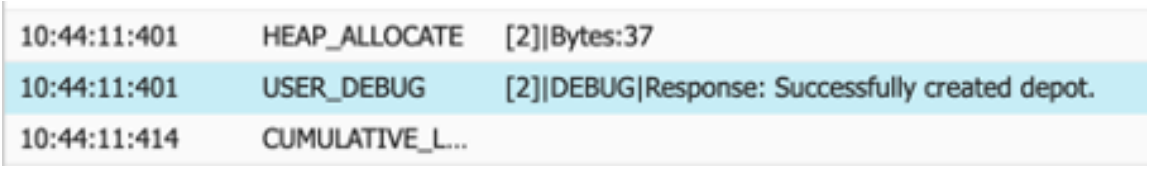

Abbildung 4.6: Rückgabewert der APEX REST Klasse

Der erwartete Rückgabewert kann in der Konsole abgelesen werden. Dementsprechend sollte ein neues Depot erstellt worden sein, welches in Abbildung 4.7 zu sehen ist.

Nach diesem Vorgehen müssen nun weitere Tests durchgeführt werden. Dabei sollten auch negative Tests beachtet werden. Dazu zählen fehlende Eingabeparameter, falsche E-Mail-

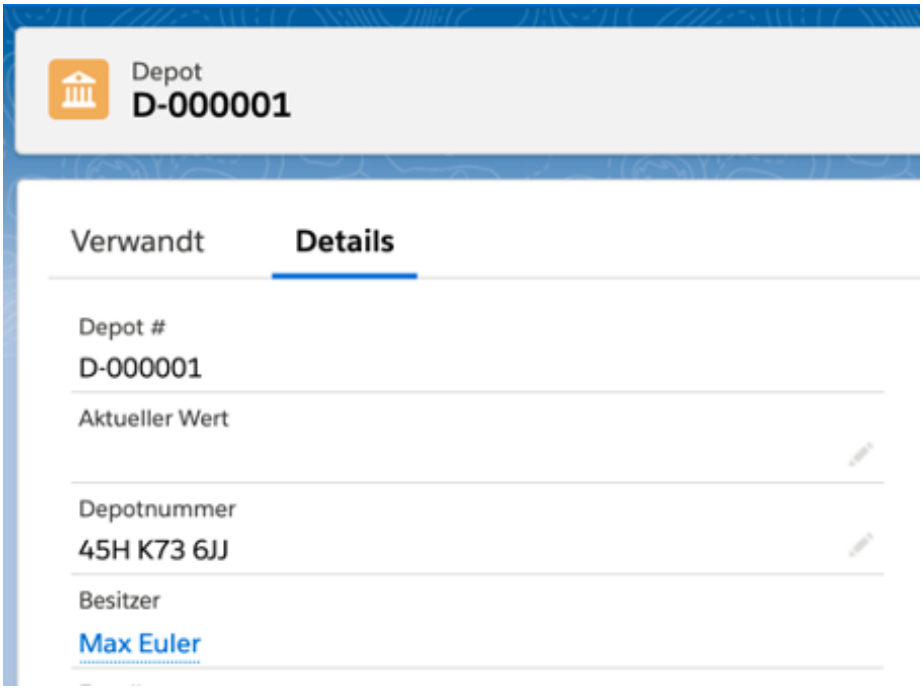

Abbildung 4.7: Erstelltes Depot

Adressen oder bereits vorhandene Depots. Da diese relativ identisch ablaufen, wird an dieser Stelle nicht weiter darauf eingegangen.

4.5 Depotverwalter

Für die Anlage der Depots ist ebenfalls die Festlegung eines zuständigen Depotverwalters notwendig. Auf Salesforce wird dies ein Benutzer, ein Datensatz vom Typ *User*, sein. Dafür wird es auf dem Depot Objekt ein neues Feld vom Typ Nachschlagebeziehung geben. Die Befüllung soll automatisch geschehen, deshalb wird dies in der bereits implementierten APEX REST Klasse ergänzt. Da der Verwalter nur bei der Erstellung des Depots gesetzt werden muss, ist keine weitere Anlage eines Auslösers nötig. Wie bereits in Abschnitt 2.5 besprochen, besteht das Standardverhalten darin, dass immer der Inhaber des Kontakts als Verwalter des Depots übernommen wird. In der APEX REST Klasse muss dafür lediglich die ID des Kontaktinhabers abgefragt und auf das Depot übernommen werden. Auf dem Depot wurde dafür speziell das Feld Verwalter angelegt, welches immer auf einen Benutzer zeigt. Wichtig ist aber, dass dieses Verhalten deaktiviert werden kann. Dafür wird eine benutzerdefinierte Bezeichnung erstellt. Ein Anwendungsfall für eine solche war bereits bei der Implementierung der Abfrage der Aktien Daten in Abschnitt 4.2 zu sehen. In diesem Fall wird die Bezeichnung nur den Wert *TRUE* oder *FALSE* enthalten. Wie in Abbildung 4.8 zu sehen, wird als Standard *TRUE* hinterlegt.

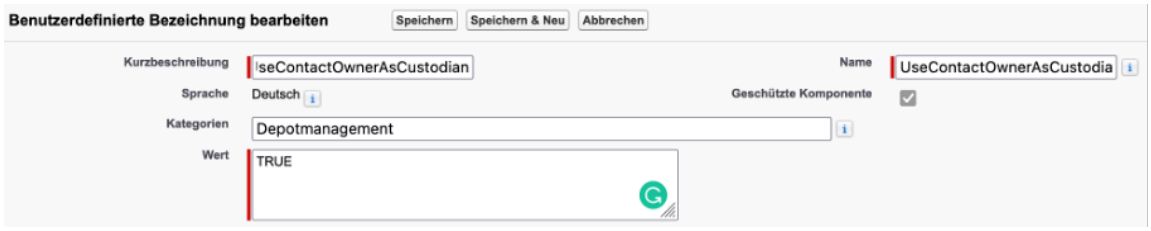

Abbildung 4.8: Erstellung einer benutzerdefinierten Bezeichnung

Im Code kann der Wert der Bezeichnung nun geprüft und dementsprechend gehandelt werden. Soll der Verwalter nicht automatisch vom Kontakt Datensatz übernommen werden, muss einzig nur der Wert der Bezeichnung angepasst werden. Dies ist in den Einstellungen direkt auf der Produktionsumgebung möglich. Es muss keine Code-Anpassung mit anschließender Bereitstellung durchgeführt werden. An dieser Stelle ist jedoch wichtig anzumerken, dass benutzerdefinierte Bezeichnungen immer als ein String behandelt werden. Obwohl als Wert *TRUE* eingetragen wurde, muss der Code auf einen String prüfen. Die entsprechende Prüfung wird in Listing 4.14 gezeigt.

```
if (System.Label.UseContactOwnerAsCustodian == 'TRUE') {
    newDepot.Custodian__c = depotOwner[0].OwnerId;
}
```

Listing 4.14: Zugriff auf eine benutzerdefinierte Bezeichnung

Für ausgewiesene Mitarbeiter muss eine Änderung des Depot-Verwalters problemlos und einfach möglich sein. Hierfür kann auf eine sogenannte *Object-Specific Action* zurückgegriffen werden. Darunter verstehen sich Aktionen, die spezifisch für ein Objekt angelegt werden können. Dabei kann der Typ „Datensatz aktualisieren“ ausgewählt werden. Diese Aktion wird auf der Datensatzseite platziert und kann somit immer den aktuell betrachteten Datensatz aktualisieren.[48] Das Layout kann frei gewählt werden. Das ist hier von Vorteil, da nur der Verwalter angepasst werden soll. Deshalb wird nur dieses Feld im Layout enthalten sein. Ebenfalls kann die Beschriftung des Buttons frei gewählt werden. Dieser wird später auf der Datensatzseite die Aktion öffnen wird. Alle Anpassungen können in den Einstellungen vorgenommen werden. Weder Code, noch eine andere Automation sind notwendig. Zusätzlich ist das Standardstyling von Salesforce und das Verhalten für das Speichern und ebenfalls das Abbrechen der Aktion automatisch inkludiert, wie in Abbildung 4.9 erkennbar. Die Aktion wird in den Hervorhebungsbereich der Datensatzseite hinzugefügt. Dabei können Filter zur Sichtbarkeit angelegt werden. So kann festgelegt werden, dass nur berechtigte Personen die Aktion sehen und damit die Änderung des Verwalters durchführen können. Dies kann somit vom Anwender der Applikation individuell an die Unternehmensstruktur angepasst werden.

Ein Problem, oder eine mögliche Situation, die eintreten könnte, ist die Änderung des Inhabers der Kontakte. Sollte sich dieser ändern, wäre der Depotverwalter nicht mehr

Abbildung 4.9: Ändern des Depotverwalters

identisch mit dem Inhaber des Depotinhabers. Um dies zu verhindern, müsste ein Auslöser auf dem Kontakt implementiert werden, der die Änderung des Inhabers immer auf das Depot überträgt. Da Kontakt aber ein etabliertes Standard-Objekt ist, das in jeder Saleforce-Umgebung in verschiedenen Kontexten eingesetzt wird, würde diese Automation den Zuständigkeitsbereich der Applikation überschreiten. Für die Umsetzung müsste ein APEX Trigger oder ein durch Datensatz ausgelöster Flow entwickelt werden, der sich nur um diese Problemstellung kümmert. Wie bereits in Abschnitt 3.6 besprochen, sollte immer nur einen Trigger/Flow pro Objekt eingesetzt werden. Besitzt der Anwender der Applikation jedoch eines davon, würde mit der Installation der Applikation ein zweiter Trigger/Flow im System ergänzt werden. Aus diesem Grund wird auf die Implementierung dieses Prozesses verzichtet.

4.6 Depot – Käufe und Verkäufe

Bis zu diesem Punkt können Depots angelegt werden. Als nächstes müssen demnach die Prozesse zur Befüllung der Depots implementiert werden. Wie bereits besprochen, werden die Depots durch die verschiedenen Anteile zusammengestellt. Die Anteile selbst werden durch Käufe und Verkäufe verwaltet. Zur Erreichung dieses Prozesses muss das Datenmodell erstellt und die Logik zur Verknüpfung der Käufe und Verkäufe mit den Anteilen implementiert werden. Das bedeutet, dass jeder Kauf und Verkauf automatisch in den Anteilen widergespiegelt werden muss. Da das Datenmodell bereits behandelt wurde, wird in diesem Abschnitt nur noch auf die zu implementierende Logik eingegangen.

Die Informationen zu Käufen und Verkäufen werden aus den externen Systemen an die Salesforce-Plattform übergeben werden. Aus diesem Grund muss hier erneut auf eine benutzerdefinierte APEX REST Klasse zurückgegriffen werden. Genauer gesagt handelt es sich um zwei Stück, da jedes HTTP-Verb pro Klasse nur einmal verwendet werden darf. Zu Beginn wird die Klasse zur Anlage und Verwaltung der Käufe besprochen. Folgende Schritte müssen implementiert werden:

1. Prüfung der Eingabeparameter.

2. Abfrage der verwandten Datensätze des Kaufs (Aktie und Depot).

3. Erstellen des Datensatzes für den Kauf.

4. Erstellen oder anpassen des Datensatzes, der den Anteil der Aktie im Depot widerspiegelt.

Der letzte Schritt ist sehr wichtig. Denn nicht für jeden Kauf muss ein neuer Anteil zum Depot hinzugefügt werden. Es wird häufig vorkommen, dass der Depotinhaber schon eine gewisse Anzahl einer Aktie besitzt und nur neue Aktien nachkauft. In solch einem Fall muss der vorhandene Anteil um die Anzahl der neu gekauften Aktien erhöht werden. Zuerst muss allerdings der Kauf in Salesforce angelegt werden. Hier muss noch keine Prüfung auf einen vorhandenen Datensatz erfolgen, da jeder Kauf einzigartig sein soll. Der Code dafür muss lediglich prüfen, ob zu den angegebenen Daten eine passende Aktie und ein passendes Depot vorhanden sind. Daraufhin wird ein Datensatz vom Objekt „Kauf“ erstellt. Sollte währenddessen etwas fehlschlagen, wird die Fehlermeldung als Antwort zurückgegeben. Der noch fehlende Code muss die Übernahme des Anteils in das Depot behandeln. Dieser ist etwas komplizierter, wie in Listing 4.15 erkennbar. Dafür wird aus der Aktie und dem Depot eine eindeutige Kennung generiert, die im System immer einzigartig ist. Als nächstes muss geprüft werden, ob unter dieser Kennung bereits ein Datensatz existiert. Dies würde bedeuten, dass die Aktie in diesem Depot bereits vorhanden ist. Falls das der Fall ist, wird die Anzahl des neuen Kaufs zur bestehenden Anzahl hinzugerechnet und der Datensatz aktualisiert. Falls nicht, wird ein neuer Anteil erstellt.

```
String uniqueIdentifier = purchasedShare[0].Id + '' + depot[0].Id;
List<Proportion__c> existingProportion = [SELECT ID, Count__c, UniqueIdentifier__c
                                    FROM Proportion__c
                                    WHERE UniqueIdentifier__c = :uniqueIdentifier
                                    WITH SECURITY_ENFORCED
                                    LIMIT 1];

Proportion__c newProportion = new Proportion__c();
if (existingProportion.size() == 1) {
    newProportion.Id = existingProportion[0].Id;
    newProportion.Count__c = existingProportion[0].Count__c + count;
} else {
    newProportion.Share__c = purchasedShare[0].Id;
    newProportion.Count__c = count;
    newProportion.Depot__c = depot[0].Id;
    newProportion.UniqueIdentifier__c = uniqueIdentifier;
}

Database.UpsertResult saveResultProportion = Database.upsert(newProportion);

if (!saveResultProportion.isSuccess()) {
    return 'Error while creating proportion record: '
        + saveResultProportion.getErrors();
```

```
}
```

Listing 4.15: Erstellung oder Aktualisierung eines Anteils

Damit wäre die Klasse zur Erstellung von Käufen vollständig. Der erste Test kann, genau wie der Test der Depot APEX REST Klasse, in der Entwicklerkonsole durchgeführt werden. Der Code dafür wird in Listing 4.16 aufgeführt. Dafür wird das Symbol einer Aktie, eine beliebige Anzahl und die Nummer eines existierenden Depots benötigt.

```
DepotPurchaseWebservice.createNewPurchase('JNJ', 3, '45H K73 6JJ');
```

Listing 4.16: Simulation eines Aktien-Kaufs

Nach Ausführung kann mithilfe zweier Datenbank Abfragen validiert werden, dass die korrekten Datensätze erstellt wurden. In Abbildung 4.10 und Abbildung 4.11 können die jeweiligen Resultate eingesehen werden.

Purchase__c@2:25 PM

SELECT ID, Share__r.Name, Depot__r.Name, Count__c, pricepershare__c from purchase__c

Query Results - Total Rows: 1

Id	Share__r.Name	Depot__r.Name	Count__c	PricePerShare__c
a031j00000OiAueAAF	Johnson & Johnson	D-000001	3	131.0606

Abbildung 4.10: Der erstelle Kauf Datensatz

Proportion__c@2:27 PM

SELECT ID, Share__r.Name, Depot__r.Name, Count__c, value__c from proportion__c

Query Results - Total Rows: 1

Id	Share__r.Name	Depot__r.Name	Count__c	Value__c
a041j00000Etw3yAAB	Johnson & Johnson	D-000001	3	393.18

Abbildung 4.11: Der erstelle Anteils Datensatz

Identisch zum Prozess eines Kaufs, muss nun die Logik für die Verkäufe implementiert werden. Ein Unterschied hier ist die Fehlerbehandlung. Erhält das System die Information zum Verkauf von beispielsweise drei Apple-Aktien, im Depot sind aber weniger als drei vorhanden, darf kein Verkauf durchgeführt werden. Ein solches Problem konnte bei den Käufen nicht auftreten. Daraus ergeben sich folgende zu erledigende Schritte:

1. Prüfen und Abfragen der Aktie und des Depots.
2. Bilden der eindeutigen Kennzeichnung für den Anteil.
3. Abfragen des Anteil Datensatzes.

4. Prüfen der vorhandenen Anzahl der Aktie im Depot.
5. Subtrahieren der verkauften Aktien auf dem Anteil Datensatz.
6. Erstellen eines neuen Verkauf Datensatzes.
7. Gegebenenfalls Löschung des Anteils.

Listing 4.17 zeigt die dafür entwickelte Methode. Zuerst werden SOQL-Abfragen durchgeführt, um die betroffene Aktie und das Depot zu erhalten. In diesem Zug wird zeitgleich geprüft, ob die übermittelten Parameter korrekt und dafür Datensätze vorhanden sind. Dieser Punkt ist dahingehend wichtig, damit der Aufrufer spezifische Fehlermeldungen erhält und auch innerhalb des Codes unbehandelte Fehler verhindert werden können. Aus den erhaltenen Datensätzen kann nun die eindeutige Kennzeichnung erstellt werden, welche den Anteil im System repräsentiert. Daraufhin erfolgt die Prüfung, ob der Verkäufer genug Aktien besitzt, um die gewünschte Anzahl verkaufen zu können. Auf diese Weise wird verhindert, dass bei der Anzahl eines Anteils eine negative Zahl zu sehen wäre. Denn es würde fachlich keinen Sinn ergeben, sollten in einem Depot eine negative Anzahl an Aktien vorhanden sein.

```
@HttpPost
global static String createNewSale(
    String shareSymbol,
    Integer count,
    String depotNumber
) {
    List<Share__c> shareToSale = [SELECT ID, CurrentPrice__c
                                  FROM Share__c
                                  WHERE Symbol__c = :shareSymbol
                                  WITH SECURITY_ENFORCED
                                  LIMIT 1];

    if (shareToSale.size() != 1) {
        return ‘No share found for this symbol.‘;
    }

    List<Depot__c> depot = [SELECT ID
                            FROM Depot__c
                            WHERE DepotNr__c = :depotNumber
                            WITH SECURITY_ENFORCED
                            LIMIT 1];

    if (depot.size() != 1) {
        return ‘No depot found for this number.‘;
    }

    String uniqueIdentifier = shareToSale[0].Id + ’’ + depot[0].Id;
    List<Proportion__c> existingProportion = [SELECT
                            ID, Count__c, UniqueIdentifier__c
                        FROM Proportion__c
                        WHERE UniqueIdentifier__c = :uniqueIdentifier
```

```
                        WITH SECURITY_ENFORCED
                        LIMIT 1];

    if (existingProportion.size() == 0
        || existingProportion[0].Count__c < count
    ) {
        return 'Not enough shares available for sale in this depot.';
    }

    Proportion__c prop = existingProportion[0];
    prop.Count__c = existingProportion[0].Count__c - count;

    Database.SaveResult saveResultProportion = Database.update(prop);

    if (!saveResultProportion.isSuccess()) {
        return 'Error while creating proportion record: '
            + saveResultProportion.getErrors();
    }

    Sale__c newSale = new Sale__c(
        Share__c = shareToSale[0].Id,
        Depot__c = depot[0].Id,
        PricePerShare__c = shareToSale[0].CurrentPrice__c,
        Count__c = count,
        SalesDate__c = Date.today()
    );

    Database.SaveResult saveResultSale = Database.insert(newSale);

    if (!saveResultSale.isSuccess()) {
        return 'Error while creating sale: ' + saveResultSale.getErrors();
    }

    return 'Successfully created new sale record.';
}
```

Listing 4.17: Ausführung eines Verkaufs

Da Verkäufe, genau wie Käufe, nur erstellt und nicht aktualisiert oder gelöscht werden können, ist die Implementierung dieses Teils der APEX REST Klasse zum Einspielen der Verkäufe damit abgeschlossen. Eine besondere Situation, die bis jetzt noch nicht behandelt wurde, ist das Löschen eines Anteils. Sollte ein Depotinhaber den vollständigen Anteil einer Aktie in seinem Depot verkaufen, muss der Datensatz, der den Anteil widerspiegelt, aus Salesforce entfernt werden. Es macht fachlich keinen Sinn, einen Anteil mit einer Anzahl von null Aktien im Depot zu haben. Aufgrund des bisherigen Aufbaus und aus Sicht der Übersichtlichkeit und zukünftigen Pflege der Anwendung ist es am sinnvollsten, die dafür benötigte Logik ebenfalls in der eben erstellten Klasse zu platzieren. So kann auf eine weitere zu implementierende Lösung, wie einen APEX Trigger oder einen Flow, verzichtet werden. Andernfalls würde der Datensatz aktualisiert werden und erst während

des Speichervorgangs würde die Prüfung stattfinden, ob er gegebenenfalls gelöscht werden muss. Auf der Salesforce-Plattform sollte immer versucht werden, jede mögliche Datenbank Operation zu umgehen, sofern es eine ressourcensparende Alternative gibt. Dazu zählen beispielsweise die Erstellung, Aktualisierung oder Löschung eines Datensatzes. Generell sollte vermieden werden verschiedene auslösende Automationen auf dem gleichen Objekt zu definieren. Andernfalls kann eine Architektur entstehen, in der wiederholte Datenbank Operationen ausgeführt werden. Zusätzlich kann dies dazu führen, dass die sogenannte *save order of execution* mehrfach ausgeführt wird.[23] Diese besteht aus allen definierten Logiken, die bei der Speicherung eines Datensatzes ausgeführt sind. Die genaue Bedeutung der *order of execution* wurde bereits im Kapitel Order of Execution besprochen.

Ergänzt werden muss die neue Logik, nachdem die neue Anzahl auf dem vorhandenen Datensatz berechnet wurde. Dort kann geprüft werden, ob die Anzahl noch größer null ist. Ist dies der Fall, kann der Datensatz auf gewöhnliche Weise aktualisiert werden. Sollte dies jedoch nicht der Fall sein, wird der Datensatz gelöscht. Das funktioniert mit einem Aufruf der Methode *Database.delete();*. Die verbleibende Logik der Methode kann weiterhin ohne Einschränkungen ausgeführt werden. Ein Datensatz für den Verkauf muss in beiden Fällen erstellen werden, unabhängig davon, ob vorher der Anteil aus dem Depot gelöscht wurde. Der Verkauf hat in beiden Fällen stattgefunden und sollte daher auch immer erstellt werden, um keine Lücken in der Berichtserstattung aufkommen zu lassen.

4.7 Depotwert

Eine wichtige Kennzahl des Depots ist der Gesamtwert. Da sich dieser aus den verschiedenen Anteilen zusammensetzt, muss er berechnet werden. Wichtig dabei ist, dass er immer aktuell ist, demnach bei jedem Kauf oder Verkauf angepasst wird. Für dieses Anforderung wird ein Flow implementiert. Dieser wird auf Änderungen an den Anteilen der Depots hören und falls diese auftreten, den Gesamtwert anpassen. Die Bedingungen werden im Start-Element des Flows definiert. Außerdem muss dort angegeben werden, ob der Flow vor oder nach dem Speichern des Datensatzes ausgeführt werden soll. Da der Flow einen anderen Datensatz und nicht den Anteil selbst aktualisieren wird, ist eine Ausführung nach Speichern des Datensatzes die sinnvolle Wahl. Dies wurde bereits im Abschnitt Order of Execution besprochen.

Der erste zu erledigende Schritt ist das Abfragen der Werte aller Anteile für das betroffene Depot. Dafür kann ein sogenanntes *Datensätze abrufen* Element verwendet werden. Das erlaubt die Definition einer SOQL-Abfrage mit beliebigen Filtern. Außerdem kann definiert werden, welche Felder gespeichert werden sollen. Der Flow wird durch Änderungen an einem Anteil Datensatz ausgelöst werden, wodurch die Informationen über diesen Datensatz in der globalen Variable *$Record* gespeichert sein werden.[49] Da ein Anteil immer eine Beziehung zu einem Depot hat, kann über diese Variable auf die ID des Depots zugegriffen werden. Diese ID wird benötigt, um die Abfrage der Anteile im Element *Datensätze abrufen* korrekt zu filtern. Abbildung 4.12 zeigt die einfache Definition der SOQL-Abfrage über die Benutzeroberfläche eines Flows.

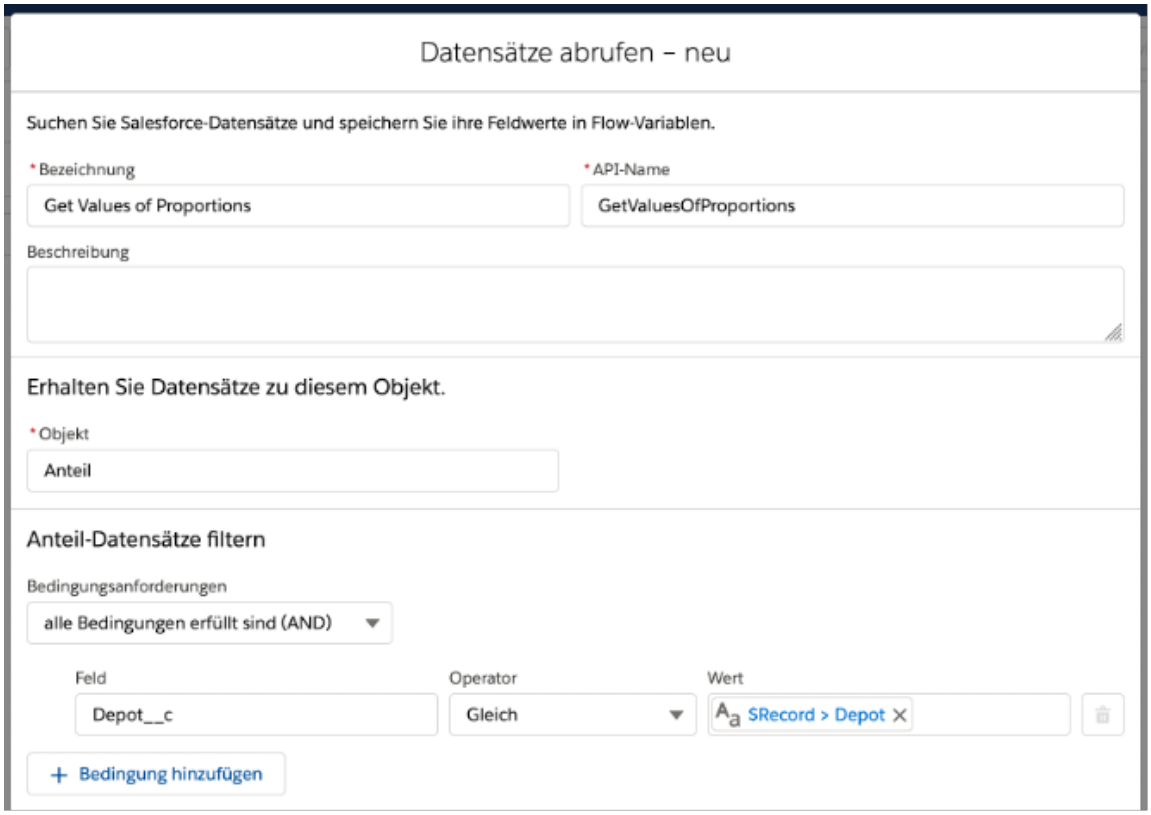

Abbildung 4.12: Definition einer SOQL-Abfrage in einem Flow

Nachdem nun alle benötigten Werte vorhanden sind, kann die Berechnung mithilfe eines *Schleifen* Elements durchgeführt werden. Dieses Element ermöglicht es über eine beliebige Kollektion zu iterieren. Es kann mit einer normalen FOR-Schleife einer Programmiersprache verglichen werden. Bei der Definition des Schleifen Elements muss lediglich das Ergebnis des zuvor erstellten Elements, in welchem die Werte der Anteile abgerufen wurden, als Sammlungsvariable ausgewählt werden. Durch die Schleife werden zwei neue Pfade zur Verfügung gestellt. Einen, der in jeder Iteration ausgeführt wird und einen für den weiteren Verlauf des Flows. Der erste Pfad repräsentiert dabei den Bereich innerhalb der geschweiften Klammern, um wieder den Bezug zu der kodierten FOR-Schleife herzustellen. Innerhalb der Schleife wird ein Zuweisungs-Element benötigt. Dieses wird den Wert jedes Anteils zu einer Variablen namens *DepotValue* hinzufügen, um so den Gesamtwert zusammen zurechnen.

Nach kompletter Ausführung der Schleife beinhaltet die Variable *DepotValue* als Ergebnis den gewünschten Gesamtwert des Depots. Der letzte Schritt besteht nun in der Aktualisierung dieses Wertes auf dem Depot Datensatz. Dafür wird ein *Datensätze aktualisieren* Element eingesetzt. In diesem Element wird über die bereits bekannte Depot ID das gewünschte Depot angegeben. Zu aktualisieren ist nur das Feld *Aktueller Wert* auf dem Depot. Als Wert wird die Variable *DepotValue* verwendet, welche in der Schleife befüllt

wurde. Wie in Abbildung 4.12 zu erkennen, ist auch die Definition einer Datenbank Operation einfach über die Benutzeroberfläche möglich.

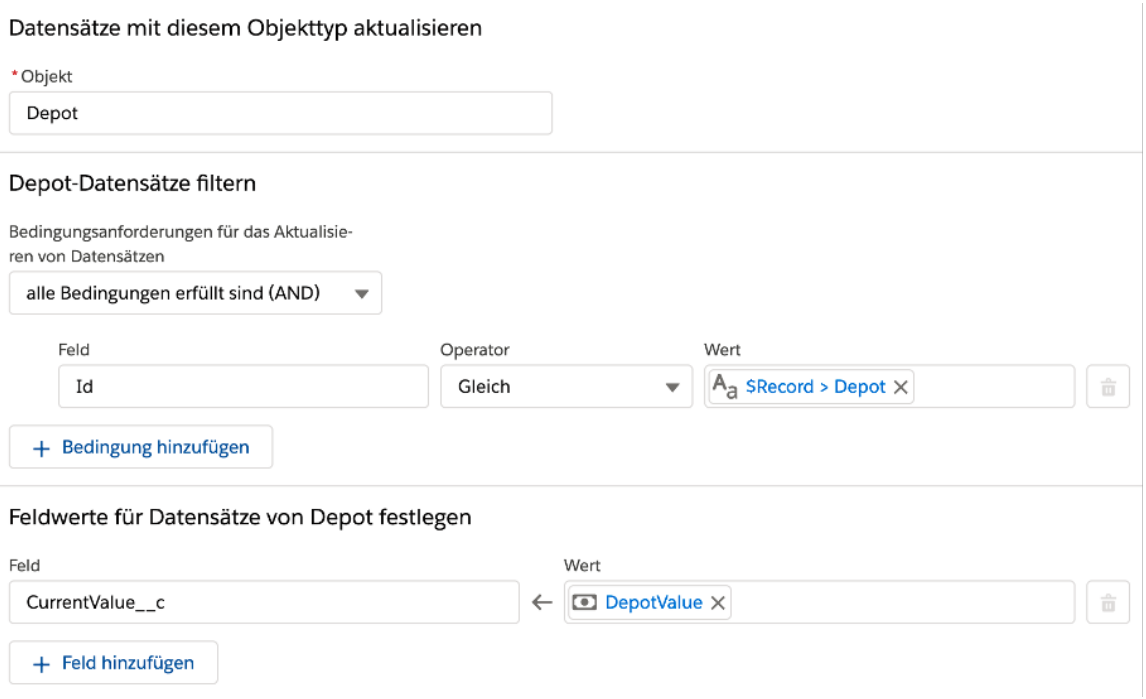

Abbildung 4.13: Definition einer Datenbank Operation in einem Flow

Damit ist die Logik des Flows abgeschlossen. Nach Aktivierung wird bei jeder Änderung eines Anteils in einem Depot der Gesamtwert neu berechnet und auf dem Depot Datensatz abgespeichert. So sind die Depots immer auf dem aktuellen Stand.

4.8 Depotentwicklung und Performance

Der nächste zu implementierende APEX Code der Applikation behandelt die Erfassung der Depotentwicklung und die Berechnung der Performance Kennzahlen. Dafür wird auf eine Batch Klasse zurückgegriffen, welche den Namen *DatedDepotValueBatch* erhält. Das geplante Vorgehen für die Depotentwicklung sieht wie folgt aus: Für die Depots muss jeden Tag der aktuelle Wert festgehalten werden. Nur so kann ein Diagramm angezeigt werden, welches die Entwicklung des Depots aufzeigt. Dafür wurde ein Objekt angelegt, welches den Wert des Depots für ein bestimmtes Datum festhalten soll. Da hier mit großen Datenmengen gearbeitet wird, ist eine Batch Klasse notwendig.

Die Grundlage der Batch Klasse sind die Depot Datensätze. Die Logik muss für alle Datensätze im System ausgeführt werden, weshalb auf die SOQL-Abfrage in der *start()*-Methode keine Filter angewendet werden. Je nach Größe des Kunden und Anzahl der verwalteten

Depots kann hier natürlich eine große Menge an Daten anfallen. Aus diesem Grund muss bei der Klasse auf bewährte Verfahren und eine saubere Implementierung gesetzt werden. Nur so kann die Verarbeitung beim zukünftigen Einsatz effizient durchgeführt werden. Die Logik zur Erfassung der Depotentwicklung wird in der *execute()*-Methode verhältnismäßig einfach ausfallen. Es reicht die Erstellung eines Datensatzes vom Objekt *Datierter Depotwert* für jedes Depot. Diese halten den aktuellen Depotwert, das heutige Datum und einen Verweis auf das Depot.

Der komplexere Part besteht aus der Berechnung der Performance Kennzahlen. Hier ist anzumerken, dass diese Berechnung aus zwei Ebenen besteht. Es muss sowohl die Performance der einzelnen Aktien, als auch die Performance der Depots berechnet werden. Letzteres ist abhängig von der berechneten Aktien Performance. Deshalb müssen diese Schritte sequentiell durchgeführt werden. Die Werte sollen als Prozentwert angegeben werden. Bevor mit der Implementierung begonnen werden kann, muss auf den Anteilen ein Feld ergänzt werden. Dieses Feld soll den prozentualen Wert halten, den die Aktie im Depot ausmacht. Der Wert wird für die Berechnung der Depot-Performance benötigt. Als Feld Typ wird eine Formel verwendet, welche in einem Prozentwert resultieren wird. Das Ergebnis ist der Quotient des Wertes des Anteils und dem Gesamtwert des Depots:

Value___c / Depot___r.CurrentValue___c.

Eine Besonderheit dabei ist der Zugriff auf den Gesamtwert des Depots. Dieses Feld befindet sich auf dem Depot Datensatz, welcher dem Anteil Datensatz übergeordnet ist. Durch die Verwendung von „___r“ kann innerhalb von Formelfeldern (identisch auch in APEX Code) auf die einzelnen Felder des übergeordneten Datensatzes zugegriffen werden. Das Ergebnis kann bei den Anteilen in Abbildung 4.14 in der rechten Spalte „Anteil %“ abgelesen werden.

Depots > D-000001

Anteile

5 Elemente · Sortiert nach Wert · vor einer Minute aktualisiert

	Aktie-Name	Symbol	Anzahl	Wert ↓	Anteil %
1	Alphabet Inc	GOOGL	2	3.933,44 €	67,73 %
2	Microsoft Corporation	MSFT	4	828,43 €	14,26 %
3	Walmart Inc	WMT	4	458,64 €	7,90 %
4	The Coca-Cola Company	KO	7	319,43 €	5,50 %
5	Johnson & Johnson	JNJ	2	267,96 €	4,61 %

Abbildung 4.14: Anteile eines Depots mit Angabe des Prozentwerts

Ein wichtiger Aspekt der Performance Werte ist, dass diese täglich neu berechnet werden müssen. Die Zeitperiode verschiebt sich täglich um einen Tag nach vorne und dementsprechend muss die Berechnung erneuert werden. Die Logik für die Berechnung der Aktien Performance wird in den *ShareDailyPriceBatch* eingebaut. Dieser legt jeden Morgen die neuen Preise an. Sobald der neuste Preis vorhanden ist, kann anschließend die Performance berechnet werden. Die Zeiträume werden in einer separaten Klasse definiert, damit

diese nur einmalig erzeugt werden müssen und danach an verschiedenen Stellen einsetzbar sind. Hier ist entscheidend, dass nur die Wochentage berücksichtigt werden. Aus diesem Grund muss auf die bereits definierte Funktion, mit welcher der letzte Wochentag ausfindig gemacht wird, zurückgegriffen werden. In der Klasse werden die Zeiträume für die letzte Woche, den letzten Monat und das letzte Jahr definiert. Der aktuelle Preis stammt immer vom Vortag und wird durch die Variable „yesterday“ repräsentiert. Listing 4.18 zeigt die dafür implementierte Klasse. Die Logik wird direkt im Konstruktor ausgeführt. So muss keine Methode aufgerufen werden und die Daten sind direkt nach der Erstellung einer Instanz der Klasse zugänglich. Um eine Manipulation der Zeiträume zu verhindern, sind die Werte als *private* markiert und können nur über die entsprechenden *getter*-Methoden erhalten werden.

```
public inherited sharing class PerformancePeriods {

    private BusinessHours stockExchangeBusinessHours;
    private Date yesterday;
    private Date lastWeek;
    private Date lastMoth;
    private Date lastYear;

    public PerformancePeriods() {
        this.stockExchangeBusinessHours = [SELECT Id FROM BusinessHours
                                           WHERE Name = 'Default'];
        this.fillTimeZones(Datetime.now());
    }

    private void fillTimeZones(Datetime now) {
        this.yesterday = this.getPreviousWorkingDay(now.addDays(-1)).date();
        this.lastWeek = this.getPreviousWorkingDay(now.addDays(-7)).date();
        this.lastMoth = this.getPreviousWorkingDay(now.addMonths(-1)).date();
        this.lastYear = this.getPreviousWorkingDay(now.addYears(-1)).date();
    }

    public Datetime getPreviousWorkingDay(Datetime d){
        return !BusinessHours.isWithin(this.stockExchangeBusinessHours.Id, d)
        ? getPreviousWorkingDay(d.addDays(-1))
        : d;
    }

    public Date getYesterday() {
        return this.yesterday;
    }

    public Date getLastWeek() {
        return this.lastWeek;
    }

    public Date getLastMoth() {
        return this.lastMoth;
    }
```

```
    public Date getLastYear() {
        return this.lastYear;
    }
}
```

Listing 4.18: Festlegung der Zeiträume zur Performance Berechnung

Mit diesen Zeiträumen müssen nun als erstes die Preise der Aktien an diesen Tagen abgefragt und gespeichert werden. Da die Klasse auf bis zu 200 Aktien Datensätzen gleichzeitig operiert, müssen die Daten nach Aktie gruppiert abgespeichert werden. Daraus entsteht eine sogenannte Map, welche die Aktie als Schlüssel besitzt, mit dem Namen *valueByPeriodByShare*. Maps, ähnlich wie JSON, sind Datenstrukturen, die aus einer Liste an Schlüssel-Werte-Paaren bestehen.[23] In diesem Fall befindet sich hinter jedem Schlüssel eine weitere Map. Diese zweite Map ordnet jedem Datum einen Preis zu. Daraus entsteht der Datentyp *Map<Id,Map<Date,Decimal> >*. Als nächstes wird über die zu bearbeitenden Aktien iteriert und in jedem Durchlauf die zugehörige Map, mit den Zuordnungen von Datum und Preis, aus *valueByPeriodByShare* entnommen. Damit kann für jeden gewünschten Zeitraum die Berechnung der Performance aufgerufen werden, welche in die Methode *calculatePerformance()* verlagert wurde. Listing 4.19 zeigt diese Berechnung. Diese spiegelt die im Abschnitt 2.3 besprochene Berechnung wider. Die Prüfung zu Beginn ist notwendig, da bei nicht vorhandenem Preis eine sogenannte *NullPointerException* geworfen werden würde. Dieser Fehler entsteht, wenn auf eine Variable zugegriffen wird, die keinen Wert enthält. Da dies bei einem Ausfall der API, die zur Abfrage der Preise verwendet wird, vorkommen kann, muss die Prüfung stattfinden.

```
private Decimal calculatePerformance(Decimal currentPrice, Decimal oldPrice) {
    if (currentPrice == null || oldPrice == null) {
        return null;
    }
    Decimal diff = currentPrice - oldPrice;
    Decimal res = (diff / oldPrice) * 100;
    return res;
}
```

Listing 4.19: Berechnung der Performance einer Aktie

Nach einem Test dieser Methode an der Microsoft Aktie, können die berechneten Kennzahlen in der Entwicklerkonsole abgelesen werden, wie in Abbildung 4.15 dargestellt. Ein Abgleich mit der Webseite eines renommierten Anbieters solcher Daten bestätigt die Korrektheit der Berechnung.

Der nächste Schritt ist die Weiterverarbeitung dieser Informationen, damit daraus die Performance der Depots berechnet und dargestellt werden kann. Die Performance des Depots ergibt sich aus der Kombination der Performance der Aktien mit dem Anteil, den die einzelnen Aktien im Depot ausmachen. Da die Werte jeden Tag aufs Neue berechnet werden

Symbol__c	PerformanceOneWeek__c	PerformanceOneMonth__c	PerformanceOneYear__c
MSFT	3.207032680802572	2.5932518812201635	24.432754489236643

Abbildung 4.15: Berechnete Performance der Microsoft Aktie

müssen, kann die bereits vorhandene Batch Klasse *DatedDepotValueBatch* verwendet werden. Diese iteriert bereits täglich über alle Depots und kann somit eingesetzt werden, um eine weitere Iteration über alle Depots zu vermeiden. Die benötigten Informationen für die Berechnungen können alle aus dem *Junction*-Objekt Anteil gewonnen werden. Durch den Einsatz einer *Relationship*-Abfrage können alle verwandten Felder in einer SOQL-Abfrage inkludiert werden. Mithilfe von *Relationship*-Abfragen ist es möglich, Felder von mehr als einem Objekt in einer einzelnen SOQL-Abfrage zu erhalten. Dies ist allerdings nur möglich, wenn die Objekte über eine Beziehung verbunden sind. Dabei ist es nicht wichtig, ob es eine Master-Detail- oder eine Nachschlage-Beziehung ist. *Relationship*-Abfragen können sowohl vom übergeordneten zum untergeordneten Datensatz, als auch vom untergeordneten zum übergeordneten Datensatz traversieren. In diesem Fall wird letzteres verwendet. Dies ist nötig, da der prozentuale Anteilswert und die Performance Werte, die auf der Aktie gespeichert sind, benötigt werden. Listing 4.20 zeigt die SOQL-Abfrage.

```
private List<Proportion__c> getProportions(List<Depot__c> depots) {
    return [SELECT
                ID,
                Depot__c,
                ProportionPercent__c,
                Share__r.PerformanceOneWeek__c,
                Share__r.PerformanceOneMonth__c,
                Share__r.PerformanceOneYear__c
            FROM Proportion__c
            WHERE Depot__c IN :depots];
}
```

Listing 4.20: SOQL-Abfrage

Um mit den Daten arbeiten zu können, müssen diese zuerst nach dem Depot gruppiert werden. Dafür wird die bereits bekannte Datenstruktur Map eingesetzt, die jedem Depot eine Liste mit den zugehörigen Anteilen zuordnet. Bei der Verarbeitung der Daten wird die Performance das größte Problem darstellen. Aufgrund der Datenstruktur, die mehrere Depots mit jeweils mehreren Anteilen vorweist, ist eine verschachtelte FOR-Schleife unvermeidlich. Da darin aber keine rechenintensiven Aufgaben, sondern nur einfache Arithmetik geplant ist, kann diese Situation akzeptiert werden. Listing 4.21 zeigt die entscheidende Funktion *getDepotWithPerformance()*. In einer FOR-Schleife, welche über alle Depots iteriert, wird diese aufgerufen und nimmt die ID des Depots und die Liste mit allen zugehörigen Anteilen entgegen. Zuerst müssen die Variablen deklariert und initialisiert werden, welche jeweils die Performance für die Woche, den Monat und das Jahr halten werden. Bei der Iteration über die Anteile, welche den inneren Teil der verschachtelten FOR-Schleife bildet, werden diese

Werte nun zusammengerechnet. Dafür wird der Performance Wert der Aktie mit ihrem prozentualen Anteil im Depot multipliziert. Hier ist es wichtig den prozentualen Anteil vorher durch 100 zu dividieren, da Salesforce Prozente wie absolute Zahlen abspeichert. So werden 34,29% im Code als 34,29 und nicht als 0,3429 dargestellt. Nachdem die Werte berechnet sind, müssen sie auf dem Depot Datensatz gespeichert werden. Da die ID mit an die Funktion übergeben wird, kann hier folgender aufwandsparender Vorgang durchgeführt werden: Es wird ein neuer Depot Datensatz angelegt und bei diesem wird das ID Feld mit der vorhandenen ID befüllt. Somit wird beim Aufruf von *Database.update()* das bereits vorhandene Depot aktualisiert und die Felder für die Performance Werte werden übernommen.

```
private Depot__c getDepotWithPerformance(Id depotId, List<Proportion__c> props) {
    Decimal depotWeekPerformance = 0;
    Decimal depotMothPerformance = 0;
    Decimal depotYearPerformance = 0;

    for (Proportion__c prop : props) {
        Decimal absoluteValue = prop.ProportionPercent__c / 100;

        if (prop.Share__r.PerformanceOneWeek__c != null) {
            depotWeekPerformance
                += (prop.Share__r.PerformanceOneWeek__c * absoluteValue);
        }

        if (prop.Share__r.PerformanceOneMonth__c != null) {
            depotMothPerformance
                += (prop.Share__r.PerformanceOneMonth__c * absoluteValue);
        }

        if (prop.Share__r.PerformanceOneYear__c != null) {
            depotYearPerformance
                += (prop.Share__r.PerformanceOneYear__c * absoluteValue);
        }
    }

    Depot__c updatedDepot = new Depot__c(
        Id = depotId,
        PerformanceOneWeek__c = depotWeekPerformance,
        PerformanceOneMonth__c = depotMothPerformance,
        PerformanceOneYear__c = depotYearPerformance
    );

    return updatedDepot;
}
```

Listing 4.21: Berechnung der Depot Performance

Damit ist an diesem Punkt der Arbeit die Berechnung der gesamten Performance auf allen Ebenen abgeschlossen. In Zukunft werden jeden Tag zuerst die Werte für die einzelnen Aktien berechnet. Dies geschieht, nachdem die neuen Preise eingelesen wurden. Zu einem

späteren Zeitpunkt wird der Job starten, welcher den aktuellen Stand der Depots abspeichern wird. Dieser wird nun ebenfalls die Performance auf den Depots berechnen. Über die Darstellung der Daten wird in Abschnitt 4.10 gesprochen, sobald die Datensatzseiten implementiert werden.

4.9 Depot – Berichte

Da nun die gesamte Logik zur Einspielung und Verarbeitung aller Daten abgeschlossen ist, fällt als nächstes die Darstellung der Depots an. Da auf der Datensatzseite der Depots zwei Diagramme geplant sind, werden zwei verschiedene Berichte benötigt, welche diese Daten bereitstellen. Für den Bericht, zuständig für die Aufteilung des Depots, muss zuerst ein Berichtstyp erstellt werden. Dieser Berichtstyp definiert die Daten, welche später in dem Bericht verfügbar sind.[40] Die Erstellung ist in den Einstellungen möglich. Um einen neuen Berichtstypen zu erstellen, muss als erstes das Hauptobjekt gewählt werden. Für den Bericht zur Aufteilung des Depots ist dies das Depot Objekt, wie in Abbildung 4.16 zu erkennen ist. Im nächsten Schritt wird definiert, welche verwandten Datensätze von untergeordneten Objekten zudem verfügbar sein sollen. Dabei können nur Objekte ausgewählt werden, die eine Beziehung zum Hauptobjekt haben. In diesem Fall sind das die Anteile. Hier ist entscheidend, dass die Option „Jeder ‘A‘-Datensatz muss mindestens einen verwandten ‘B‘-Datensatz haben.“ ausgewählt wird. Für Depots, die noch keine Anteile besitzen, soll kein Diagramm angezeigt werden.

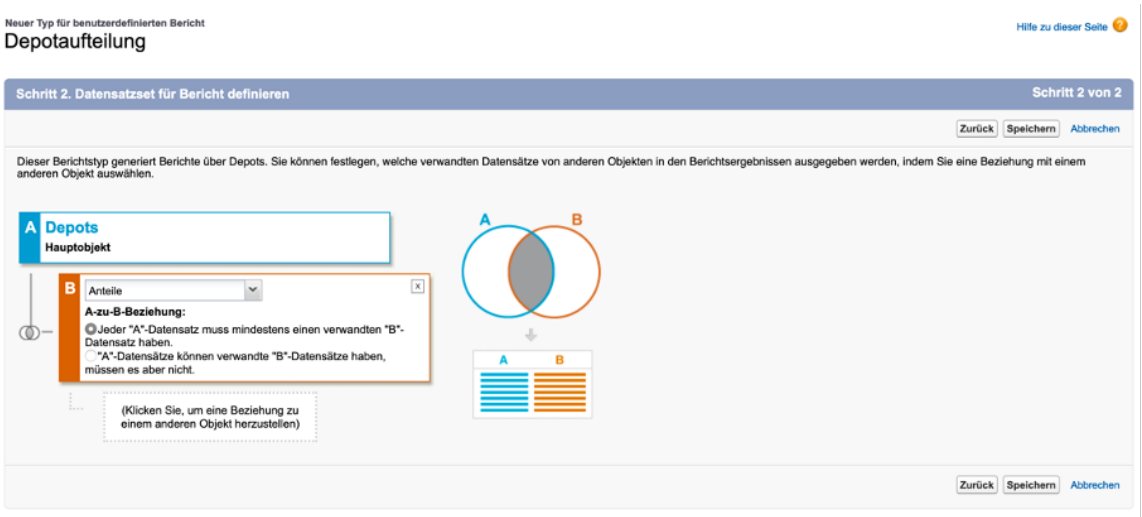

Abbildung 4.16: Erstellung eines Berichtstypen

Nach Erstellung des Berichtstypen kann das Feldlayout angepasst werden. Dieses bestimmt welche Felder der einzelnen Objekte zur Anzeige verfügbar sind.[50] Je weniger Felder ausgewählt werden, desto übersichtlicher wird die Erstellung des Berichts. Jedoch besteht die

Gefahr, dass bei zu wenigen Feldern bestimmte Dinge nicht mehr dargestellt werden können, da beispielsweise ein Feld für eine gewünschte Gruppierung fehlt. Aus diesem Grund muss hier genau abgewogen werden, welches Ziel mit Berichtstypen erreicht werden soll und in welchem Kontext dieser später verwendet wird. Für den geplanten Bericht gehören beispielsweise der Name der Aktie und der Wert des Anteils zu den erforderlichen Feldern. Nach diesen Einstellungen kann der neue Berichtstyp verwendet werden, um den gewünschten Bericht zu erstellen. Es handelt sich dabei um einen einfachen Bericht. Die Daten müssen anhand des Aktien Symbols gruppiert werden. Einzig der Wert des Anteils wird benötigt. Als Diagramm ist ein sogenanntes *Donut*-Diagramm am sinnvollsten. Es stellt die Aufteilung am besten dar. Zudem lässt sich der Gesamtbetrag im Mittelpunkt anzeigen. Wenn das Diagramm in Abschnitt 4.10 auf die Datensatzseite der Depots hinzugefügt wird, muss auf den Filter geachtet werden. Abbildung 4.17 zeigt den erstellten Bericht.

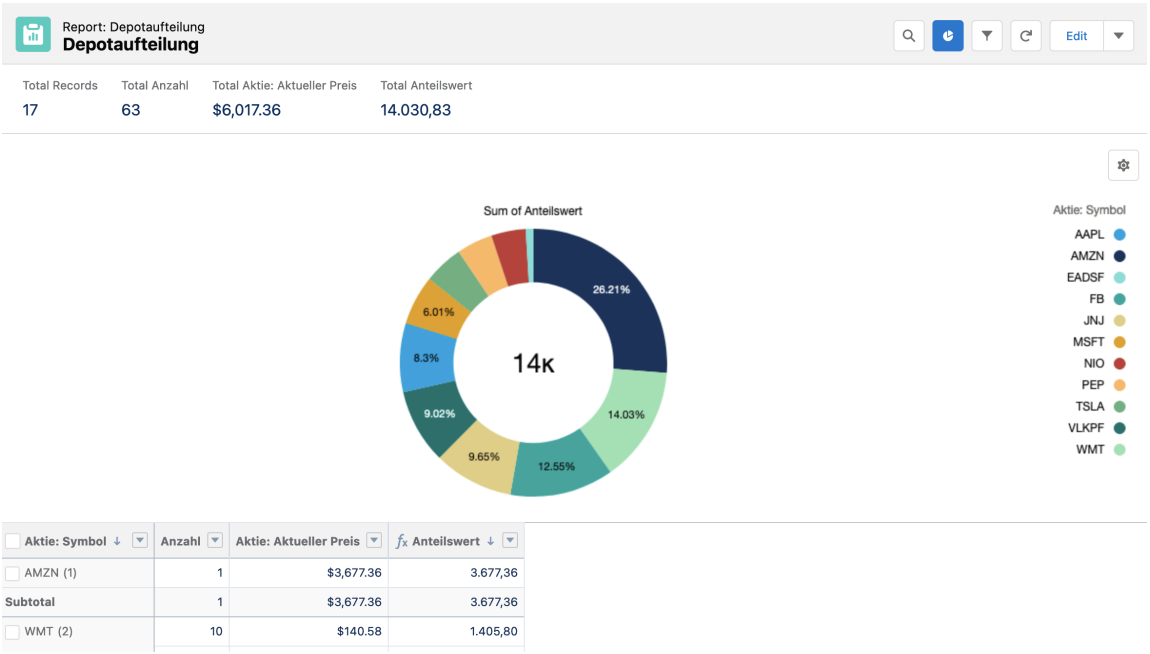

Abbildung 4.17: Bericht zur Depotaufteilung

Für den Bericht zur Darstellung des Depotverlaufs wird ebenfalls ein neuer Berichtstyp benötigt. Dieser hat ebenso das Depot als das Hauptobjekt. Mit inkludiert werden noch die Datierten Depotwerte. Diese für die Darstellung des Verlaufs notwendig. Auch hier sollte das Feldlayout angepasst werden, um nur die gewünschten Felder zu beinhalten. Als Diagrammtyp wird ein *Linien*-Diagramm verwendet. Dieses eignet sich zur Darstellung des Verlaufs am besten. Die Daten müssen anhand des Anlagedatums des Datensatzes gruppert werden. Mit diesem Schritt wäre jetzt auch der dritte und letzte Bericht des Projektes abgeschlossen. Alle Berichte werden im Abschnitt 4.10 dargestellt.

4.10 Benutzeroberfläche

Ein wichtiger Punkt der Applikation ist eine ansprechende, übersichtliche und nützliche Darstellung der Daten, damit Nutzer die Funktionen schnell und allumfassend verwenden können. Dazu müssen als erstes Datensatzseiten für die Aktien und Depots erstellt werden. In Salesforce ist dies im Lightning App Builder mit Klicks möglich und kein Code ist notwendig. Auf der Datensatzseite der Aktien ist der Aktienkurs der entscheidende Faktor. Deshalb sollte dieses Diagramm möglichst groß dargestellt werden. Des weiteren sollten die wichtigsten Kennzahlen schnell erkennbar sein. Dafür eignen sich die sogenannten *kompakten Layouts*. Sie erlauben, es eine kleine Anzahl an Feldern zu selektieren, welche dann im oberen Teil der Datensatzseite angezeigt werden.[51] Für die Aktien sind die entscheidenden Felder: Symbol, Assetklasse, Sektor, Aktueller Preis und Dividendenrendite. Da die Beschreibung der Aktie ein längerer Text ist, sollte dieses Feld nicht im kompakten Layout vorhanden sein. Dafür wird ein separater Bereich hinzugefügt, welcher die Beschreibung ohne Platzprobleme beinhalten wird. Ebenfalls essenziell sind die Performance Kennzahlen der Aktie. Aus diesem Grund werden diese auf der Datensatzseite hervorgehoben. Erkennbar ist dies in Abbildung 4.18.

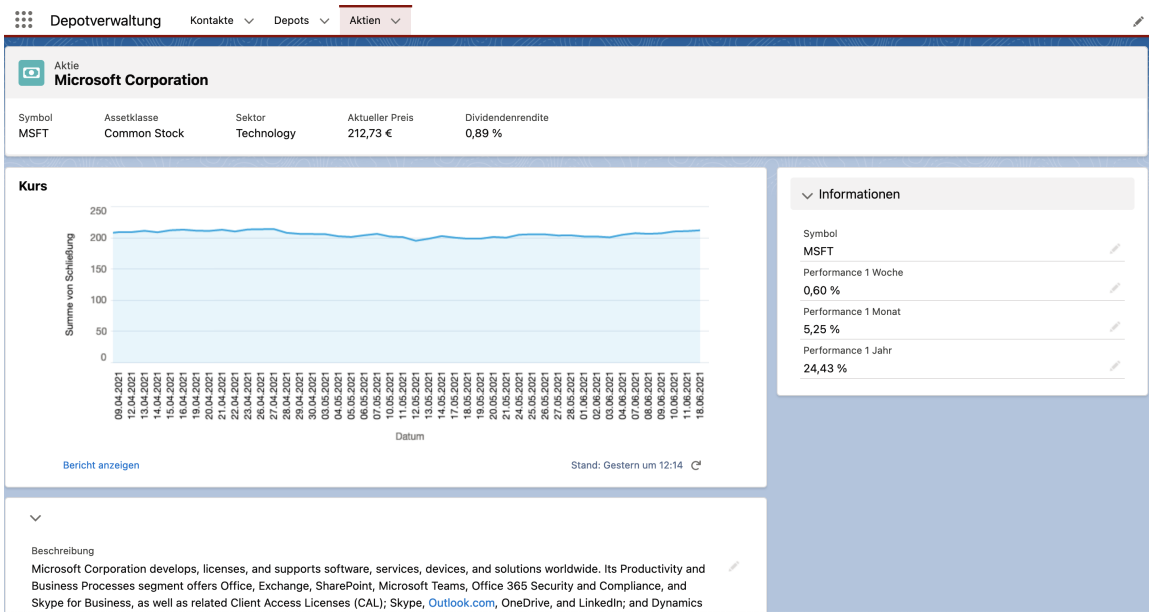

Abbildung 4.18: Datensatzseite der Aktien

Etwas komplexer wird die Datensatzseite der Depots, dargestellt in Abbildung 4.19. Hier müssen zwei Diagramme, sowie alle wichtigen verwandten Datensätze anzeigt werden. Dazu gehören die Anteile, Käufe und Verkäufe. Für die Anzeige der Aufteilung des Depots wird der bereits erstellte Bericht verwendet. Dieser wird nach der ID des aktuell betrachteten Depots gefiltert und zeigt somit jederzeit die richtige Aufteilung an. Die verwandten Datensätze werden durch sogenannte *Related Lists* angezeigt. Darin werden alle untergeordneten Datensätze angezeigt, die über eine Beziehung mit dem Depot verbunden sind.

Für jede Beziehung existiert eine gesonderte Liste. Diese können beliebig auf der Seite verteilt werden. Zudem kann für jede Liste gewählt werden, welche Spalten angezeigt werden sollen.[52] So sind die wichtigsten Informationen auf einen Blick verfügbar. Wie bereits bei der Aktien Datensatzseite, wird auch für die Depots ein kompaktes Layout angelegt. Hier sind die entscheidenden Felder der Inhaber, der aktuelle Wert und der Verwalter des Depots.

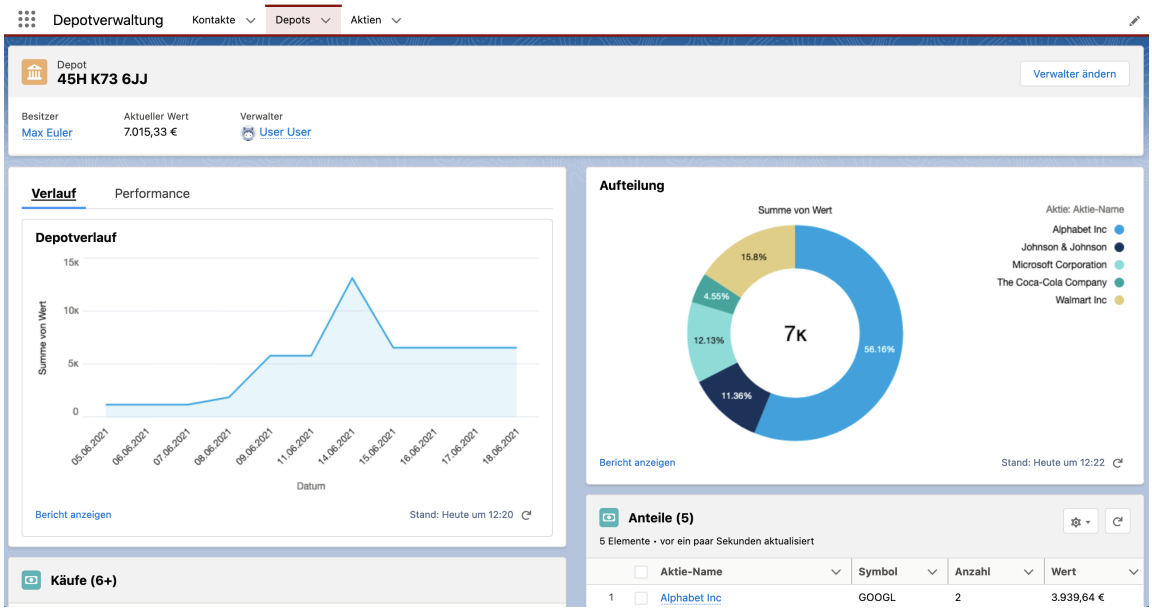

Abbildung 4.19: Datensatzseite der Depots

Um aus den implementierten Komponenten eine wirkliche Applikation zu erstellen, muss zum Schluss noch eine sogenannte *Lightning-Anwendung* erstellt werden. Dabei handelt es sich um eine Zusammenstellung verschiedener Komponenten, die fachlich zusammenpassen. Auf der Salesforce-Plattform definieren Lightning-Anwendungen welche Elemente in der Navigationsleiste angezeigt werden.[53]

Unter den Elementen der Navigationsleiste können sich verschiedene Komponenten befinden. Oftmals ist das erste Element die Startseite. Pro Anwendung kann genau eine Startseite definiert werden. Diese kann sich allerdings je nach Profil des Nutzers unterscheiden. Der Zweck der Startseite ist eine Zusammenstellung von beliebigen Komponenten, wie beispielsweise Diagrammen, Aufgaben- und Ereignislisten oder zuletzt verwendete Datensätze. Der Großteil der Elemente, die man in einer Navigationsleiste vorfindet, sind Registerkarten für Objekte. Diese werden in der Lightning-Anwendung zur Depotverwaltung eingesetzt. Unter diesen Registerkarten sind immer Listenansichten des jeweiligen Objekts zu finden. Sie bieten demnach den Zugang zu den gespeicherten Datensätzen über die Benutzeroberfläche.

Die entwickelte Lightning-Anwendung wird in Abbildung 4.20 gezeigt. Erstellt werden kann diese im Anwendungs-Manager in den Einstellungen. Als erstes müssen Name, Beschrei-

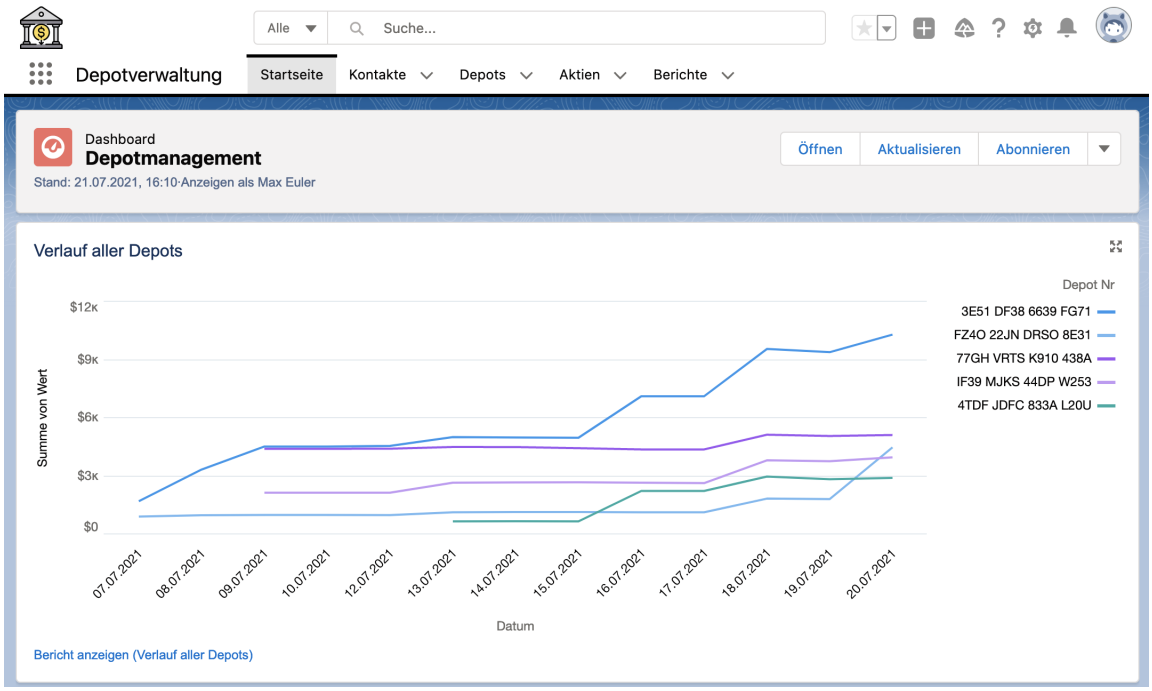

Abbildung 4.20: Depotverwaltung Lightning-Anwendung

bung und eine primäre Farbe angegeben werden. Die primäre Farbe wird von der Navigationsleiste übernommen werden. So kann schneller unterschieden werden, welche Anwendung aktuell geöffnet ist. Optional kann noch ein Bild hinzugefügt werden, welches dann in der oberen linken Ecke das Salesforce Logo ersetzen würde. Es ist aber anzumerken, dass all diese Dinge vom Anwender angepasst werden können. Die Anwendung soll nur den Startpunkt liefern. Wie der Anwender die Komponenten am Ende einsetzt, bleibt ihm überlassen. Den spannende Teil bei der Erstellung stellt die Auswahl der Navigationselemente dar. Hier wird der Aufbau der Navigationsleiste festgelegt. Für die Depotverwaltung sind fünf Elemente von Bedeutung: Eine Startseite, die Kontakte, genauer gesagt die Inhaber der Depots, die Depots selbst, Aktien und Berichte.

Der letzte Schritt der Anwendungserstellung betrifft die Berechtigung zur Nutzung der Anwendung. Hier müssen alle Profile ausgewählt werden, die Zugriff auf die Anwendung haben sollen. Es reicht allerdings die Auswahl des Systemadministrator Profils, da der Anwender wahrscheinlich seinen eigenen benutzerdefinierten Profilen die Berechtigung geben möchte. Damit ist die Erstellung der Anwendung zur Depotverwaltung abgeschlossen.

5 Fazit und Ausblick

Diese Arbeit wurde mit dem Ziel durchgeführt, die Möglichkeiten zur Entwicklung einer Applikation zur Depotverwaltung zu analysieren und diese daraufhin zu implementieren. Grund der Analyse war die Klärung, ob ein Markt dafür vorhanden ist, die Umsetzbarkeit aus technischer Sicht und eine mögliche Veröffentlichung der implementierten Applikation. Die Analyse ergab direkt zu Beginn das Resultat, dass die Salesforce-Plattform nicht zum Betrieb der wirklichen Endkunden Applikation geeignet ist, die beispielsweise die Transaktionen in Realität durchführen würde. Dafür sind andere Architekturen und Mechanismen nötig. Aus diesem Grund wurde der Fokus der Applikation auf die reine Verwaltung der Daten, die für die Depotverwaltung entscheidend sind, gerichtet. Andererseits wurde festgestellt, dass auch dieses Vorhaben umsetzbar ist, jedoch die Veröffentlichung ein Problem darstellt. Die Arbeit hat ebenfalls verdeutlicht, dass der Vormarsch des Cloud Computing mit Hinblick auf Salesforce und die wachsende Nachfrage an Aktien als Finanzprodukt, vielversprechende wirtschaftliche Gründe zum Vertrieb einer solchen Applikation darstellt.

Auf der technischen Seite hat die Arbeit gezeigt, dass das Einspielen, Pflegen, Berechnen und Darstellen aller benötigten Daten und Informationen auf der Salesforce-Plattform umsetzbar ist. Es wurden verschiedene Technologien und Konzepte der Plattform besprochen und ihre Vorteile für die konkreten Anforderungen verdeutlicht. Durch die Verwendung von APEX REST Klassen können Daten empfangen und in das erstellte Datenmodell eingespielt werden. Der Einsatz von Batch Klassen in Kombination mit dem Schedulable Interface ermöglicht eine automatische und regelmäßige Verarbeitung großer Datenmengen, um daraus Informationen, wie die Performance Kennzahlen oder den Depotverlauf, zu generieren. Auch die sinnvolle grafische Darstellung der Daten kann mithilfe von Berichten und Diagrammen erreicht werden. Dies wurde bei der Implementierung der Datensatzseiten gezeigt. Sowohl das Donut-Diagramm zur Darstellung der Aufteilung des Depots, als auch das Linien-Diagramm zur Darstellung des Depotverlaufs bieten wertvolle Informationen und eine benutzerfreundliche Darstellung für die zuständigen Mitarbeiter. Es hat sich also herausgestellt, dass die Salesforce-Plattform durchaus in der Lage ist, eine Depotverwaltung, unter gewissen Einschränkungen, zu betreiben. Es ist eine Applikation entstanden, welche entscheidende Informationen zu allen gewünschten Aktien ins System einspielen, dort pflegen und darstellen kann. Es werden Performance Kennzahlen berechnet, die ausschlaggebend für die Bewertung der Aktien sein können. Die Depots werden aus skalierbaren Datensätzen aufgebaut. Alle entscheidenden Transaktionen können festhalten werden. Zudem liefert das Datenmodell eine Grundlage für eine beliebige Weiterverarbeitung der gewonnen Informationen.

Die Schwierigkeit liegt eindeutig bei der Veröffentlichung der Applikation im AppExchange. Dafür gibt es mehrere Ursachen. Erstens wäre dort die Beschaffung der Daten zu nennen.

Aus rechtlicher oder vertraglicher Sicht ist es diffizil, die benötigten Daten von einem Anbieter zu erhalten und diese dann ohne Einschränkungen an jeden Verwender der Applikation weiterzugeben. Die Implementierung hat gezeigt, dass es für den Eigenbetrieb keine Herausforderung darstellt, einen Anbieter zu finden. Viele stellen ihre Daten dazu kostenlos zur Verfügung. Sobald die Applikation aber für einen kommerziellen Betrieb eingesetzt werden soll, müssen die vertraglichen, rechtlichen und auch wirtschaftlichen Faktoren berücksichtigt werden. Auch auf der technischen Seite ist dies schwer umzusetzen, da jeder Verwender direkt mit dem Anbieter kommunizieren muss. Daher braucht er einen eindeutigen API-Schlüssel, der nicht mit der Applikation ausgeliefert werden kann. Das ist auf der Salesforce-Plattform notwendig, da es keinen zentralen Server gibt, welcher die Daten für alle Verwender zur Verfügung stellen könnte. Hier wurde während der Ausarbeitung der grundlegenden Idee der Unterschied zwischen der Architektur einer mobilen Anwendung oder Web-Anwendung und Anwendungen aus dem AppExchange nicht ausreichend bedacht. Als zweiter Punkt wäre die Aktualität der Daten zu nennen. Die Implementierung hat gezeigt, dass selbst tagesaktuelle Daten einen großen Aufwand in der Pflege mit sich bringen.

Ein weiterer kritischer Punkt, der während der Arbeit aufkam, ist die unbekannte Struktur des Anwenders der Applikation. So stellt die implementierte Applikation den Besitzer des Depots als Kontakt dar. Es kann jedoch nicht garantiert werden, dass jeder potenzielle Anwender seine Kunden ebenfalls als Kontakte darstellt. Dies wäre ein Grund für diesen, die Applikation nicht zu installieren.

Als Resultat dieser Arbeit kann festgehalten werden, dass die Implementierung einer Depotverwaltung auf der Salesforce-Plattform technisch durchaus umsetzbar ist, jedoch eine generische Veröffentlichung im AppExchange auf zu viele schwer lösbare Probleme stößt. Aus technischer Sicht stellen sich hierbei natürlich die Aktualität der Daten und die große Datenmenge als die entscheidenden Faktoren dar. Die ursprüngliche Idee dieser Arbeit basierte auf einer ähnlichen Implementierung, welche allerdings mit Fonds anstelle von Aktien arbeitet. Bei Fonds ist ein tagesaktueller Preis gängig und im Regelfall dem Großteil der Anwender ausreichend. Jedoch ist bei Aktien oftmals ein Preis in Echtzeit wichtig. Die Wichtigkeit dieses Punktes wurde zu Beginn der Arbeit unterschätzt, jedoch während der detaillierten Ausarbeitung immer bewusster.

Als Möglichkeiten zur Veröffentlichung der Applikation eignet sich zum einen der direkte Vertrieb über einen Implementierungspartner. Dieser nutzt die erstelle Applikation und passt sie spezifisch an die Anforderungen des Betreibers an. Dabei werden die vertraglichen Faktoren direkt zwischen Betreiber und Datenanbieter geregelt. Die zweite Möglichkeit wäre die zur Verfügungsstellung einzelner Komponenten. So könnten diese als eine Art Bausatz dienen, um sich daraus eine beliebige Benutzeroberfläche erstellen zu können. Falls jedoch weiterhin eine generische Veröffentlichung angestrebt werden sollte, müsste die dafür durchzuführende Analyse erweitert werden. Dies könnte als Fortsetzung dieser Arbeit gesehen werden. Es müsste dafür zuerst ein Anbieter gefunden werden, der es erlaubt, die eingekauften Daten in der Applikation zu verwenden und somit für jeden Verwender zugänglich zu machen. Sollte solch ein Vertrag möglich sein, stellt der zweite Schritt die Speicherung der Daten auf einem externen Server dar. Von dort könnte sich die Applikation

die benötigten Daten holen und damit arbeiten. Um die Umsetzbarkeit dieses Vorhabens zu ermitteln, müsste die in dieser Arbeit begonnene Analyse fortgesetzt werden. Das Szenario soll nur als ein Ausblick dienen.

Literaturverzeichnis

[1] Reinheimer, Stefan: *Cloud Computing: Die Infrastruktur der Digitalisierung*, Springer Fachmedien Wiesbaden GmbH, Wiesbaden, 2018, https://doi.org/10.1007/978-3-658-20967-4

[2] Nicoletti, Bernardo, *Cloud Computing in Financial Services*, Palgrave Macmillan UK, 2013, https://doi.org/10.1057/9781137273642

[3] Rimal, B.P., Jukan, A., Katsaros, D. et al., *Architectural Requirements for Cloud Computing Systems: An Enterprise Cloud Approach*, J Grid Computing 9, 3–26, 2011, https://doi.org/10.1007/s10723-010-9171-y

[4] Weissman, Craig D., Bobrowski, Steve, *The Design of the Force.Com Multitenant Internet Application Development Platform*, Association for Computing Machinery, New York, NY, USA, 2009, https://doi.org/10.1145/1559845.1559942

[5] Manohar, Anuradha, Chouhan, Ankit, „Salesforce CRM: A new way of managing Customer Relationship in cloud environment“, *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, 1-4, 2017, https://doi.org/10.1109/ICECCT.2017.8117887

[6] Pohl, David, *Investieren in Zeiten von Corona: Untersuchung des Anlageverhaltens von Privatinvestoren in Zeiten der Corona-Pandemie*, o.A., 2020

[7] Leven, Franz-Josef, Schlienkamp, Christoph, *Erfolgreiches Depotmanagement: Wie Ihnen die moderne Portfoliotheorie hilft*, Gabler, Wiesbaden, 1998, https://doi.org/10.1007/978-3-322-89060-3

[8] Nullmeier F., Rüb F.W., *Erschöpfung des Sozialversicherungsprinzips? Gesetzliche Rentenversicherung und sozialstaatlicher Republikanismus*. In: Riedmüller B., Olk T. (eds), *Grenzen des Sozialversicherungsstaates*, Leviathan (Zeitschrift für Sozialwissenschaft), vol 14, VS Verlag für Sozialwissenschaften, 1994, https://doi.org/10.1007/978-3-322-93496-3

[9] trailhead.salesforce.com: *Visualisieren von Daten mit Dashboards und Diagrammen*, 2021, https://trailhead.salesforce.com/de/content/learn/modules/reports_dashboards/reports_dashboards_visualizing_data, letzter Zugriff: 12. Juli 2021.

[10] Hasler, Peter Thilo *Aktien richtig bewerten: Theoretische Grundlagen praktisch erklärt*, Gabler, Berlin Heidelberg, 2011, https://doi.org/10.1007/978-3-642-21170-6

[11] Shaalan, Sharif, *Salesforce for Beginners*, Packt, 2020.

[12] help.salesforce.com: *Sales Cloud Grundlagen: Personenaccounts*, 2020, https://help.salesforce.com/articleView?id=sf.account_person.htm&type=5, letzter Zugriff: 13. Juli 2021.

[13] help.salesforce.com: *Setting Up Person Accounts, Summer '21*, 2021, https://resources.docs.salesforce.com/232/latest/en-us/sfdc/pdf/salesforce_B2C_implementation_guide.pdf, letzter Zugriff: 15. Juli 2021.

[14] Davis, Andrew, *Mastering Salesforce DevOps: A Practical Guide to Building Trust While Delivering Innovation*, Apress, Berkeley, CA, 2019, https://doi.org/10.1007/978-1-4842-5473-8

[15] Fawcett, Andrew, *Salesforce Lightning Platform Enterprise Architecture*,Third edition: November 2019, Packt Publishing, Birmingham, 2019.

[16] Fowler, Martin, *Patterns of Enterprise Application Architecture*, 1. Edition, Addison-Wesley Professional, 2002.

[17] Masri, David, *Developing Data Migrations and Integrations with Salesforce: Patterns and Best Practices*, Apress, Berkeley, CA, 2019, https://doi.org/10.1007/978-1-4842-4209-4

[18] IG Group, *Aktiensymbol Definition: Was bedeutet Aktiensymbol?*, 2020, https://www.ig.com/de/trading-glossar/aktiensymbol-definition, letzter Zugriff: 16. Juni 2021.

[19] Antonopoulos, Nick, Gillam, Lee, *Cloud Computing: Principles, Systems and Applications*, Secod edition: 2017, Springer International Publishing, 2019, https://doi.org/10.1007/978-1-84996-241-4

[20] Muller, DBA, William, *An Analysis of Salesforce.Com, a Cloud Based Solutions Provider, Best Known for Its Customer Relationship Management (CRM) Products*, o.A., 2016, http://dx.doi.org/10.2139/ssrn.2850650

[21] developer.salesforce.com: *Salesforce Object Query Language (SOQL)*, 2020, https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql.htm, letzter Zugriff: 18. Juli 2021.

[22] developer.salesforce.com: *Execution Governors and Limits*, 2020, https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm, letzter Zugriff: 18. Juli 2021.

[23] Battisson, Paul, *Mastering Apex Programming: A developer's guide to learning advanced techniques and best practices for building robust Salesforce applications*, Packt, United Kingdom, 2020.

[24] developer.salesforce.com: *Apex Scheduler*, 2020, https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_scheduler.htm, letzter Zugriff: 18. Juli 2021.

[25] developer.salesforce.com: *Using Batch Apex*, 2020, https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_batch_interface.htm#apex_batch_scheduleBatch_section, letzter Zugriff: 19. Juli 2021.

[26] developer.salesforce.com: *Queueable Apex*, 2020, https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_queueing_jobs.htm, letzter Zugriff: 20. Juli 2021.

[27] trailhead.salesforce.com: *Apex-Webservices*, 2021, https://trailhead.salesforce.com/de/content/learn/modules/apex_integration_services/apex_integration_webservices, letzter Zugriff: 20. Juli 2021.

[28] Bassett, Lindsay, *Introduction to JavaScript Object Notation*, O'Reilly Media Inc., 2015.

[29] developer.salesforce.com: *Triggers*, 2020, https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_triggers.htm, letzter Zugriff: 21. Juli 2021.

[30] developer.salesforce.com: *Trigger Context Variables*, 2020, https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_triggers_context_variables.htm, letzter Zugriff: 21. Juli 2021.

[31] trailhead.salesforce.com: *Führen von Benutzern durch Ihre Geschäftsprozesse mit Flow Builder*, 2021, https://trailhead.salesforce.com/de/content/learn/modules/business_process_automation/flow, letzter Zugriff: 22. Juli 2021.

[32] help.salesforce.com: *Connected Apps*, 2021, https://help.salesforce.com/articleView?id=sf.connected_app_overview.htm&type=5, letzter Zugriff: 22. Juni 2021.

[33] help.salesforce.com: *OAuth Authorization Flows*, 2021, https://help.salesforce.com/articleView?id=sf.remoteaccess_oauth_flows.htm&type=5, letzter Zugriff: 22. Juli 2021.

[34] help.salesforce.com: *My Domain*, 2021, https://help.salesforce.com/articleView?id=sf.domain_name_overview.htm&type=5, letzter Zugriff: 22. Juli 2021.

[35] developer.salesforce.com: *SOQL Injection*, 2020, https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/pages_security_tips_soql_injection.htm, letzter Zugriff: 22. Juli 2021.

[36] help.salesforce.com: *Salesforce Sites*, 2020, https://help.salesforce.com/articleView?id=sf.sites_overview.htm&type=5, letzter Zugriff: 22. Juli 2021.

[37] trailhead.salesforce.com: *Mitigate SOQL Injection*, 2021, https://trailhead.salesforce.com/de/content/learn/modules/secure-serverside-development/mitigate-soql-injection, letzter Zugriff: 22. Juli 2021.

[38] Goodey, Paul, *Salesforce CRM – The Definitive Admin Handbook*, Fifth Edition, Packt, 2019.

[39] help.salesforce.com: *Berichte*, 2020, https://help.salesforce.com/articleView?id=sf.rd_reports_overview.htm&type=5, letzter Zugriff: 23. Juli 2021.

[40] help.salesforce.com: *Referenz für Berichtstypen*, 2020, https://help.salesforce.com/articleView?id=sf.reports_report_type_reference.htm&type=5, letzter Zugriff: 23. Juli 2021.

[41] help.salesforce.com: *Steuern der Standard-Berichtssichtbarkeit*, 2020, https://help.salesforce.com/articleView?id=sf.security_sharing_owd_external_user_srt.htm&type=5, letzter Zugriff: 23. Juli 2021.

[42] architect.salesforce.com: *Record-Triggered Automation*, 2021 https://architect.salesforce.com/design/decision-guides/trigger-automation/#Performance_Discussion__Same_Record_Field_Updates, letzter Zugriff: 23. Juli 2021.

[43] developer.salesforce.com: *Developer Hub*, 2020 https://developer.salesforce.com/docs/atlas.en-us.packagingGuide.meta/packagingGuide/dev_hub_intro.htm, letzter Zugriff: 23. Juli 2021.

[44] help.salesforce.com: *Benutzerdefinierte Bezeichnungen*, 2020, https://help.salesforce.com/articleView?id=sf.cl_about.htm&type=5, letzter Zugriff: 23. Juli 2021.

[45] developer.salesforce.com: *Callout Limits and Limitations*, 2020 https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_callouts_timeouts.htm, letzter Zugriff: 24. Juli 2021.

[46] help.salesforce.com: *Ausführen von Apex-Aktionen in Flows*, 2020 https://help.salesforce.com/articleView?id=flow_build_extend_apex.htm&type=0, letzter Zugriff: 24. Juli 2021.

[47] help.salesforce.com: *Festlegen der Geschäftszeiten*, 2020, https://help.salesforce.com/articleView?id=sf.customize_supporthours.htm&type=5, letzter Zugriff: 24. Juli 2021.

[48] help.salesforce.com: *Objektspezifische Aktionen*, 2020, https://help.salesforce.com/articleView?id=sf.actions_overview_object_specific.htm&type=5, letzter Zugriff: 24. Juli 2021.

[49] help.salesforce.com: *Flow-Ressource: Globale Variablen*, 2020, https://help.salesforce.com/articleView?id=sf.flow_ref_resources_global_variables.htm&type=5, letzter Zugriff: 24. Juli 2021.

[50] help.salesforce.com: *Einrichten des Feldlayouts für Berichte, die basierend auf Ihrem Typ für benutzerdefinierte Berichte erstellt werden*, 2020, https://help.salesforce.com/articleView?id=reports_report_type_layouts.htm&type=0, letzter Zugriff: 25. Juli 2021.

[51] trailhead.salesforce.com: *Anpassen von Datensatzhervorhebungen mithilfe kompakter Layouts*, 2021 https://trailhead.salesforce.com/de/content/learn/modules/lex_customization/lex_customization_compact_layouts, letzter Zugriff: 25. Juli 2021.

[52] help.salesforce.com: *Anpassen von Themenlisten*, 2020, https://help.salesforce.com/articleView?id=sf.customizing_related_lists.htm&type=5, letzter Zugriff: 25. Juli 2021.

[53] trailhead.salesforce.com: *Erstellen und Anpassen von Lightning-Anwendungen* , 2021 https://trailhead.salesforce.com/de/content/learn/modules/lex_customization/lex_customization_apps, letzter Zugriff: 25. Juli 2021.