Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
B2B	Business to Business
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
OAuth	Open Authorization
REST	Representational State Transfer
URL	Uniform Resource Locator
App	Application
XML	Extensible Markup Language
JSON	JavaScript Object Notation
MVC	Modell View Controller

1 Einführung

1.1 Ausgangsproblematik

In der heutigen digitalen Welt sind Online-Shops für Unternehmen jeder Größe und Branche nicht mehr wegzudenken. Um eine erfolgreiche Online-Präsenz aufzubauen, ist es wichtig, eine leistungsstarke und einfach zu bedienende Benutzeroberfläche bereitzustellen. Dabei kann es einem Entwickler schwerfallen aus einer der vielen Möglichkeiten die beste Lösung für sein Projekt auszuwählen. Auch ist durch die Vielzahl der Programmiersprachen oder Plattformen die Integration in ein anderes System oft nicht leicht. Daher ist die Benutzung einer Schnittstelle für die benötigten Daten oftmals eine der besseren Entscheidungen, da die Unabhängigkeit verschiedenster Plattformen gewährleistet ist. Eine Möglichkeit, um einen unabhängigen Online-Shop zu erstellen, ist die Verwendung eines decoupled Frontends.

Decoupled Frontends ermöglichen es, die Anzeige und Interaktion mit dem Benutzer von der Verarbeitung von Anfragen und der Datenverarbeitung zu trennen. Dadurch kann die Anwendung flexibler und wartbarer gestaltet werden. In dieser Arbeit wird der Entwurf und die prototypische Umsetzung eines decoupled Frontends für einen Online-Shop auf Basis von Salesforce B2B Commerce behandelt.

1.2 Ziele, Inhalt und Methodik der Arbeit

In dieser wissenschaftlichen Arbeit wird das Ziel verfolgt, ein decoupled Frontend für einen Online-Shop auf Basis von Salesforce B2B Commerce zu entwerfen und prototypisch umzusetzen. Der Schwerpunkt der Arbeit liegt dabei auf der Nutzung der Salesforce REST API für das Anfragen der benötigten Daten im Backend und der Erstellung einer Webanwendung mit Spring Boot als Schnittstelle für das Frontend. Damit soll die Funktionalität der Kompatibilität einer externen Anwendung auf Salesforce untersucht werden, um die Möglichkeit einer späteren Anwendung für eventuelle Kunden zu bieten. Der Entwurf des Frontends selbst wird nicht im Detail behandelt, sondern dient lediglich als Anwendungsoberfläche zur Funktionsüberprüfung des Backends. Während des Projekts werden die Entwurfsentscheidungen und Arbeitsergebnisse in der dotSource Knowledge Base dokumentiert, um eine Nachvollziehbarkeit der Arbeit zu gewährleisten und andere Entwickler zu unterstützen, die ähnliche Projekte durchführen möchten. Dabei soll sich grundlegend auf die Dokumentation von Salesforce und Spring Boot zum Erstellen dieser Anwendung konzentriert werden, da diese Informationen aus erster Hand bieten.

2 Grundlagen

Um ein Verständnis für das Projekt zu bekommen, soll in den folgenden Kapiteln ein Überblick über die genutzten Rahmenbedingungen aufzeigt und begründet werden. Dafür wird zuerst Salesforce B2B erklärt und welchen Nutzen es für Unternehmen hat. Darauffolgend wird der Begriff API und Salesforce REST-API definiert und deren Nutzen für einen Headless-Ansatz erklärt. Am Ende erfolgt eine Beschreibung des genutzten Frameworks.

2.1 Salesforce B2B Commerce

Salesforce ist ein Unternehmen, das eine Reihe von Cloud-basierten Geschäftssoftwarelösungen anbietet. Mit Salesforce können Unternehmen eine Plattform zur Verwaltung und Automatisierung von Prozessen im Zusammenhang mit Kundenbeziehungen (CRM), wie z.B. Marketing, Vertrieb und Kundenservice nutzen. So umfasst Salesforce neben diesen Lösungen auch Funktionalitäten für Salesforce B2B, Salesforce B2C oder Salesforce Digital Experience.[1]

Salesforce B2B (Business-to-Business) ist dabei eine Lösung, die es Unternehmen ermöglicht, ihre Geschäftsprozesse zu automatisieren und zu optimieren. Es ist grundsätzlich für Unternehmen konzipiert, die Geschäfte mit anderen Unternehmen tätigen und bietet damit die bereits oben erwähnen Lösungen für Kundenbeziehungen an. Ein Unternehmen kann Salesforce B2B zum Beispiel nutzen, um Informationen über seine Kunden, wie Kontaktdaten, Historie und Kommunikationsprotokolle, in einer zentralen Datenbank zu speichern. Dies ermöglicht es diesem Unternehmen, personalisierte Angebote und Marketingkampagnen zu erstellen sowie die Leistung des Vertriebs zu verfolgen.[2]

Ein weiteres wichtiges Merkmal von Salesforce B2B ist die Möglichkeit, Geschäftsprozesse zu automatisieren. Beispielsweise kann ein Unternehmen Regeln festlegen, die automatisch bestimmte Aktionen auslösen, wenn ein bestimmter Kundenstatus erreicht wird. Damit können Unternehmen zum Beispiel schneller und effizienter auf Kundenbedürfnisse reagieren. Die Möglichkeit der Anwendung hängt jedoch vom Unternehmen und deren Bedürfnisse ab. So verwenden einige es als alleinstehende Anwendung, während andere es mit anderen Tools und Plattformen integrieren, um eine umfassendere und personalisiertere Lösung zu erhalten.

Salesforce bietet Unternehmen jedoch nur begrenzte Möglichkeiten an um individuelle Lösungen wie zum Beispiel ein auf einem anderen Framework basierenden Online-Shop zu betreiben. Hierfür ist der Ansatz eines Headless-Konzeptes von großer Bedeutung, da hierbei, anders als bei Salesforce, keine Beschränkung bei der Definition von genutzten Frameworks vorliegt.

[1] Vgl. [SFH23]
[2] Vgl. Ebenda

2.2 Salesforce REST API

API (Application Programming Interface) ist eine Schnittstelle, die es Anwendungen ermöglicht über diese Schnittstellen miteinander zu kommunizieren und die Funktionalität dieser anderen Anwendung zu nutzen. Dabei sendet eine Anwendung eine HTTP-Anfrage (Hypertext Transfer Protocol) an einen API-Server, der die benötigten Ressourcen bereitstellt, auf die zugegriffen werden soll. Die Anfrage enthält dabei Informationen darüber, welche Aktion durchgeführt werden soll (z.B. Abrufen, Erstellen, Aktualisieren oder Löschen einer Ressource) sowie möglicherweise weitere Daten, die für die Ausführung der Anfrage erforderlich sind (z.B. Ressourcen-ID, Abfrageparameter oder eine Authentifizierung). Dies hängt jedoch vom Konfigurierten API-Endpunkt ab. Dagegen enthält die Antwort Informationen darüber, ob die Anfrage erfolgreich war und gegebenenfalls die angeforderten Daten oder weitere Informationen.[3]

Jede Anfrageart hat ihre eigenen Anforderungen und Einschränkungen, die in der Dokumentation der API beschrieben sein sollten. Einige APIs erfordern möglicherweise Authentifizierungs- und Autorisierungsmechanismen, um sicherzustellen, dass nur autorisierte Anwendungen auf die Ressourcen zugreifen können. Zum Beispiel benutzt die Salesforce REST-API einen OAuth-Token (Open Autorisation), um eine Anfrage zu authentifizieren.[4]
OAuth. ist ein Autorisierungsprotokoll, welche es Nutzern ermöglicht nach einer Anmeldung/Bestätigung Aktionen auszuführen, ohne ihre Anmeldedaten erneut anzugeben. Hierbei wird ein solcher Token von einer API-Schnittstelle angefordert und kann nach erhalt für andere Anfragen an die gleiche API verwendet werden.[5]

Eine REST-API (Representational State Transfer) ist dabei eine spezielle Art API, die auf dem REST-Architekturstil basiert. Es gibt grundsätzlich vier Arten von REST-Anfragen:

- GET – um Ressourcen abzurufen
- POST – um Ressourcen zu erstellen
- PUT – um Ressourcen zu aktualisieren
- DELETE – um Ressourcen zu löschen

Diese werden jedoch durch weitere Arten (Beispielsweiße PATCH, HEAD oder TRACE) erweitert. Bei der Umsetzung des Projekts wird von den unterstützten Salesforce APIs die REST-API genutzt, um auf die Commerce Ressourcen zuzugreifen.[6]

[3] Vgl. [AMA23]
[4] Vgl. Ebena
[5] Vgl. [OKT23]
[6] Vgl. [SAL23]

2.3 Decoupled Frontend

Ein decoupled Frontend ist eine Architektur, bei der die Frontend-Anwendung (z.B. eine Webseite oder eine mobile App) von der Backend-Anwendung (z.B. eine Datenbank) entkoppelt wird (Siehe Abbildung 1). Dieser Lösung stehen grundsätzlich zwei weitere Optionen gegenüber, ein klassisches oder serverseitiges Rendering.

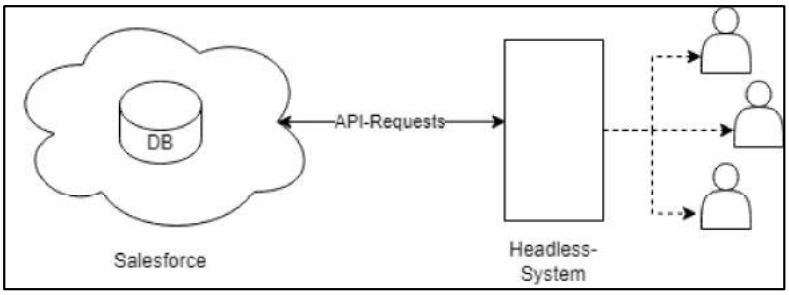

Abbildung 1: Headless-Ansatz

Ein klassisches Frontend hat die Frontend- und Backend-Logik in einer Anwendung zusammengefasst. Dies erleichtert zwar die Entwicklung da keine separate Kommunikation zwischen Front- und Backend erforderlich ist, allerdings kann es schwierig sein, eine solche Anwendung zu skalieren, zu warten oder an die Bedürfnisse des Unternehmens anzupassen. Ein serverseitiges Rendering beinhaltet dagegen, dass die Anwendung auf dem Server gerendert wird, bevor sie an den Client gesendet wird. Dies ermöglicht es, die Anwendung für Suchmaschinen und Benutzer mit langsamen Internetverbindungen besser zugänglich zu machen. Allerdings kann es auch hier schwierig sein, die Anwendung an die Bedürfnisse des Unternehmens anzupassen und zu warten.[7]

Ein decoupled Frontend bietet einige Vor- und Nachteile gegenüber den anderen Lösungen, die bei der Auswahl von Bedeutung sind. Angefangen mit der Skalierbarkeit. Da die Frontend- und Backend-Anwendungen unabhängig voneinander sind, können sie unabhängig voneinander skaliert und gewartet werden, was es ermöglicht, die Leistung der Anwendung insgesamt zu verbessern und Fehler schneller zu beheben, sowie neue Funktionen schneller zu implementieren. Dadurch ist außerdem eine Flexibilität gegeben, die es dem Entwickler ermöglicht die App (Application) leichter an die Bedürfnisse des Kunden anzupassen. Diesem steht wiederum die Größe des Aufwandes entgegen. Durch die Entkopplung müssen beide Anwendungen unabhängig voneinander gewartet und entwickelt, sowie muss die Kommunikation zwischen beiden Modulen gesichert werden. Beide Aktionen benötigen dabei mehr Zeit als eine Komplettlösung. Auch muss dafür eine korrekte und sichere Datenübertragung vorhanden sein. Aufgrund dieser Vorteile kann die Entscheidung für ein decoupled Frontend fallen, besonders wenn die Skalierbarkeit, Wartbarkeit und Flexibilität der Anwendung von großer Bedeutung sind.[8]

[7] Vgl. [CRP23]
[8] Vgl. Ebenda

2.4 Spring Boot & Thymeleaf

Spring Boot ist ein Java-basiertes Framework, mit dem Entwickler Anwendungen schneller und einfacher entwickeln können. Es bietet eine Vielzahl von Funktionen, die die Anwendungsentwicklung vereinfachen, wie z.B. automatische Konfiguration, Ressourcenverwaltung und einfache Integration mit anderen Java-Bibliotheken.[9]

Eine Option, um Spring Boot zu nutzen, besteht darin, das Spring Web MVC-Modul (Model-View-Controller) in Kombination mit Thymeleaf einzusetzen. Thymeleaf ist eine Template-Engine, die auf der Serverseite arbeitet und es ermöglicht, Daten aus einer Java-Anwendung über einfaches HTML (Hypertext Markup Language) darzustellen. Es unterstützt die Verwendung von Ausdrücken, wodurch Daten direkt im HTML-Code aufgelöst und bearbeitet werden können.[10]

MVC ist ein Designmuster der Softwareentwicklung. Es teilt eine Anwendung in drei Hauptkomponenten Model, View und Controller auf. Dabei repräsentiert das Model die Daten und Geschäftslogik einer Anwendung. Der View ist die Benutzeroberfläche der Anwendung und zeigt dem Benutzer die Daten an, die im Model enthalten sind. Der Controller nimmt Benutzeranfragen entgegen und verarbeitet sie. Er aktualisiert das Model und aktualisiert auch die View, um Änderungen an den Daten anzuzeigen.

Spring Boot konfiguriert automatisch die Komponenten, die für die Verwendung von Thymeleaf erforderlich sind, sodass sich Entwickler auf die Anwendungslogik und die Entwicklung der Benutzeroberfläche konzentrieren können. Die Kombination aus Spring Boot und Thymeleaf ermöglicht es, schnell und einfach webbasierte Anwendungen zu entwickeln und passt für diese Anwendung mit den geringen Anforderungen, bezüglich Frontend und dessen Design, perfekt.

[9] Vlg. [VMW23]
[10] Vgl. [THY23]

3 Entwurf und Konzept der Anwendung

3.1 Konzeption

Um einen reibungslosen Entwicklungsprozess zu gewährleisten ist es nötig vor der Realisierung ein Konzept zu erstellen an dem sich der Entwickler orientieren kann. Dieses kann unteranderem zur Erfolgskontrolle und als Orientierungshilfe dienen.

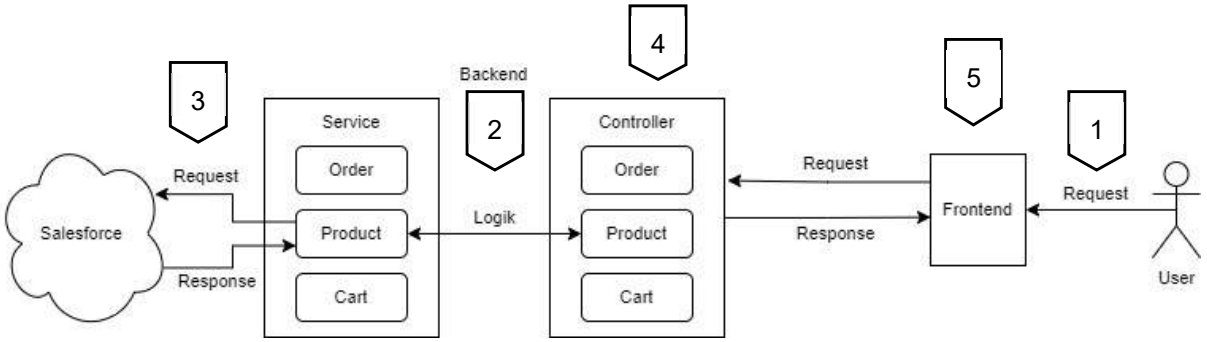

Abbildung 2: Konzept

Das Konzept soll die App in 3 verschiedene Bereiche aufteilen (Siehe Abbildung 2). Ruft der Nutzer eine URL auf wird vom Frontend eine Anfrage an das Backend gestellt (1). Ist die aufgerufene Domain in z.B. dem Product-Controller vorhanden, wird die darin aufgeführte Logik ausgeführt (2). Ist dies geschehen wird über den jeweiligen Service eine API-Request an Salesforce gestellt (3) und bekommt als Response die benötigten Daten und schickt diese an den Controller zurück (4). Dieser verarbeitet die Daten so, dass das Frontend diese anzeigen kann, fügt sie einem Model hinzu und gibt eine HTML-Seite zurück, welches die Daten aus dem Model anzeigen kann (5).

Da der Fokus dieser Arbeit primär auf der Verbindung zur Salesforce-Cloud liegt, verlagert sich ein Großteil der Funktionen auf das Backend. Es soll ausreichen ein Frontend zu erstellen welches lediglich die Funktionen der Backend-Anwendung demonstriert. Die Frontend-Anwendung soll folgende Use Cases beinhalten:

Für die Hauptseite:

- Als Nutzer möchte ich auf der Hauptseite Produkte angezeigt bekommen damit ich eine Übersicht über die verfügbaren Produkte habe
- Als Nutzer möchte ich die Möglichkeit haben, Produkte dem Warenkorb hinzuzufügen, indem ich auf einen entsprechenden Button klicke
- Als Nutzer möchte ich die Möglichkeit haben, die gewünschte Menge eines Produkts in einem Eingabefeld auszuwählen

Für den Warenkorb:

- Als Nutzer möchte ich eine Übersicht über alle Produkte haben, die ich bisher ausgewählt habe
- Als Nutzer möchte ich die Möglichkeit haben, die Menge eines Produkts im Warenkorb zu ändern oder ein Produkt aus dem Warenkorb zu entfernen

Für die Bestellseite:

- Als Nutzer möchte ich die Möglichkeit haben, meine Kontaktdaten und Lieferadresse einzugeben
- Als Nutzer möchte ich die Möglichkeit haben, die Bestellung abzuschließen und nach einer ausgewählten Zahlungsart zu bezahlen

Für jeden Service wird außerdem ein Model benötigt, in der die Service-Layer die Daten aus der Salesforce-Anfrage speichern kann. Der Controller der jeweiligen Klasse kann diese nach der Abfrage interpretieren und für das Frontend aufbereiten welches die Daten mittels Thymeleaf interpretieren und anzeigen kann. Um die Komplexität der Anwendung gering zu halten, wird eine weitere Klasse erstellt, welche die Salesforce-Anfrage beinhaltet.

3.2 Anforderungen an das Front- und Backend

Um das Projekt erfolgreich abzuschließen ist eine Beschreibung der Anforderungen essenziell. An dieser kann sich ein Entwickler während der Entwicklung orientieren und eine wiederkehrende Erfolgskontrolle veranlassen. Nach Abschluss des Projektes dient diese als Messinstrument, ob das Projekt erfolgreich abgeschlossen wurde.

Wie bereits in Kapitel 3.1 erwähnt liegt der Fokus dieses Projektes auf der Funktionalität des Backend. Diesem gegenüberstehend sollen die Anforderungen an das Frontend geringgehalten werden. Mit diesen kann überprüft werden, ob das decoupled Frontend, welches mit Salesforce kommuniziert, als mögliche Kundenlösung in Frage kommen kann. Um die Funktionalität zu prüfen, soll das Backend die in Tabelle 1 stehenden Anforderungen erfüllen.

Anforderung	Beschreibung
Kommunikation über die Salesforce API	Das Backend soll über HTTP-Requests Daten aus der Salesforce Commerce-App beziehen
Interpretieren der Daten aus Salesforce	Die Responses sollen in geeigneten Objekten abgespeichert werden, um eine Zuordnung und Übertragung an das Frontend zu ermöglichen
Authentifizierung der Salesforce abfrage	Um HTTP-Requests an Salesforce zu senden wird eine Authentifizierung benötigt
Bereitstellen einer API-Schnittstelle über die Front- und Backend kommunizieren	Für das Anzeigen der Objekte soll über Spring Boot das Backend eine API-Schnittstelle bereitstellen
Frontend und das Backend sind vollständig voneinander entkoppelt	Um eine Wartung und Wiederverwendung der Komponenten zu ermöglichen, sollten beide Teile unabhängig voneinander funktionieren

Tabelle 1: Anforderungen an das Backend

Auf dieser Grundlage können die Funktionen realisiert werden. Durch die Kommunikation mit Salesforce werden die Produktdaten (Name, Preis, ID und Beschreibung) über eine Klasse angefordert. Um diese zu interpretieren, stellt Spring Boot eine Sammlung von Klassen bereit, die die Request realisiert. Um die Response zu verarbeiten, wird mittels der Gson-Bibliothek die JSON-Response (JavaScript Object Notation) in die jeweilige Klasse konvertiert werden.

Gson ist eine Open-Source-Java-Bibliothek, die Java Objekte in JSON und umgekehrt konvertieren kann. Diese Bibliothek spart das händige Umschreiben der Response in das Java Objekt und kann damit die Realisierung der zweiten Anforderung optimal erfüllen.[1]
Vorher sollte jedoch der Weg der Authentifizierung definiert werden. Um eine REST-Request an Salesforce zu stellen, muss sich der User mittels eines OAuth2-Access-Token und ggf. mit seinen Anmeldedaten Authentifizieren. Demzufolge muss bei Zugriff auf die REST-API zunächst ein solcher Token abgefragt und gespeichert werden.[2]

Spring Boot übernimmt zudem die Kommunikation zwischen Front- und Backend. Wie bereits in Kapitel 2.4 erwähnt, überprüft Spring Boot mit einem Controller die URL (Uniform Resource Locator). Wird auf diese zugegriffen werden die benötigten Funktionen ausgeführt und über Thymeleaf an das Frontend weitergegeben.

[1] Vgl. [GIT23]
[2] Vgl. [SAL23]

3.3 Technische Herausforderungen

Softwareprojekte bringen viele technische und organisatorische Herausforderungen mit sich. Vor allem bei bislang unbekannten Themen. In diesem Abschnitt sollen Herausforderungen und Probleme aufzeigt werden, die möglicherweise bei der Umsetzung berücksichtigt werden müssen.

Um die Salesforce als API-Endpunkt und Datenbank benutzen zu können muss eine Testumgebung erstellt und richtig konfiguriert werden. Da für das Konzept dieser Arbeit noch kein Fall existiert, in dem auf der Basis von Salesforce ein Headless-Shop erstellt wurde muss daher auf die begrenzten Ressourcen der Salesforce Dokumentation[3] zurückgegriffen werden, um die Testumgebung richtig zu konfigurieren.

Ein weiterer Punkt ist die API-Nutzung. Zunächst muss die Authentifizierung der API-Request berücksichtigt werden, da Salesforce dafür das OAuth-Protokoll nutzt. Es muss geklärt werden welche Authentifizierungsschnittstellen es für einen User gibt und über welche Berechtigungen dieser verfügen muss um die Aktionen, die eigentlich über den Shop geschehen, auch per API zu tätigen. Ist diese Herausforderung geklärt müssen auch mögliche Fehlanfragen und damit die verbundenen Responses behandelt werden, um den Programmablauf nicht zu behindern.

Für all diese Probleme und technischen Herausforderungen kann die Dokumentation von Salesforce, sowie deren Lernplattfom Trailhead[4] benutzt werden. Diese beiden Quellen bieten bei Fragen und Problemen erste Anlaufmöglichkeiten.

[3] Vgl. [SAL23]
[4] Vgl. [TRA23]

4 Umsetzung

Für die Umsetzung des Projektes werden grundsätzlich zwei Hauptkomponenten benötigt. Die Salesforce Testumgebung und die auf Spring Boot basierende App. In den folgenden Kapiteln wird die Einrichtung der Testumgebung in einer neuen Organisation und der App behandelt. Darauffolgend wird der Entwicklungsprozess der einzelnen MVC-Komponenten beschrieben. Durch diesen Aufbau kann der Prozess und dessen Bestandteile nachfolgzogen und transparent gestaltet werden. Dies ist für die spätere Dokumentation in die dotSource Knowledge Base von großem Nutzen, damit nachfolgende Entwickler auf diese Informationen zugreifen können.

4.1 Einrichtung der Salesforce Testumgebung

Für dieses Projekt wurde eine neuer Playground in Salesforce erstellt. Diese kann z.B. über die Trailhead-Plattform erstellt werden. Für die Konfiguration wurde die Dokumentation von Salesforce über die Einrichtung eines B2B-Commerce-Systems genutzt. [1] Ein Playground kann genutzt werden, um mit neuen Funktionen zu experimentieren oder sich mit Salesforce vertraut zu machen. Es ermöglicht den Zugriff auf eine Salesforce-Organisation, ohne den Besitz einer kostenpflichtigen Lizenz.[2]

Da eine Vielzahl von Einstellungen in der Organisation getätigt werden müssen, ist eine Übersicht dieser in Tabelle 2 zu sehen. Um Aktionen im Store auszuführen, wie z.B. Produkte zu kaufen oder eine Bestellung auszulösen, benötigt ein Account das Buyer-Permissionset und damit verbunden eine Customer-Community-License oder höher. Da dieses Projekt aber auf einem B2B-Store basiert, wird hier mindestens die Customer-Community-Plus-License benötigt. Nur so kann ein User vollen Zugriff auf die Funktionalitäten haben.

Da dieses vordefinierte Profil standardmäßig keinen Zugriff auf die Salesforce API besitzt, wird ein Permissionset erstellt, welche die Einstellung „API Enabled“ aktiviert hat. Dem jeweiligen Nutzer muss nun dieses Permissionset zugewiesen werden, damit dieser den Shop über die Salesforce-API nutzen kann.[3]

Ist das Permissionset angelegt, kann nun im App Launcher die Commerce-App ausgewählt und konfiguriert werden. Bei der Erstellung des Stores müssen keine weiteren Schritte berücksichtigt werden und dies kann nach der Vorgabe von Salesforce geschehen. Dabei ist jedoch zu beachten, dass die Konfiguration über das B2B-Template genutzt wird.[4]

[1] Vgl. [SLA23]
[2] Vgl. Ebenda
[3] Vgl. Ebenda
[4] Vgl. Ebenda

Einstellung	Beschreibung
Commerce-Plattform und Digital Experiences	Muss in der Organisation aktiviert werden, um der App Zugriff auf Commerce-Ressourcen zu ermöglichen
Customer-Community-Plus-License	Mindestlizenz, die für den B2B-Store benötigt wird, um vollen Zugriff auf die Funktionalitäten zu haben.
Buyer-Permissionset	Benötigt, um Aktionen im Store auszuführen, wie z.B. Produkte zu kaufen oder eine Bestellung auszulösen.
Salesforce-API Permissionset	Kann erstellt werden, um den Zugriff auf die Salesforce-API zu ermöglichen. Hierbei muss „API Enabled“ aktiviert werden
B2B-Template	Wird bei der Konfiguration des Stores genutzt.
Experience Workspace -> Administration -> Members	Hier müssen das Standard-Profil (Customer-Community-Login-Plus) und das erstellte Permissionset dem Store hinzugefügt werden.

Tabelle 2: Benötigte Einstellungen in der Organisation

Für das weitere Vorgehen ist das Wissen, wie ein Salesforce Store aufgebaut ist essenziell. Daher soll im folgenden Abschnitt dieser Aufbau kurz erläutert werden

Ein Store hat Kategorien & Kataloge. Die Kategorie kann dem Katalog hinzugefügt werden und enthält Produkte. Auch kann eine Kategorie Unterkategorien enthalten und je nach Einstellung im Store angezeigt werden. Die verschiedenen Kataloge können außerdem anderen Stores zugeordnet werden. Auch gehört zu jedem Store eine Buyer-Group. Dieser können Price-Books, Members, Stores und Commerce Entitlement Policies zugewiesen werden. Je nach Konfiguration können z.B. mehrere Buyer-Groups existieren, denen jeweils verschiedene Produkte oder andere Preise angezeigt werden. (Siehe Abbildung 3).

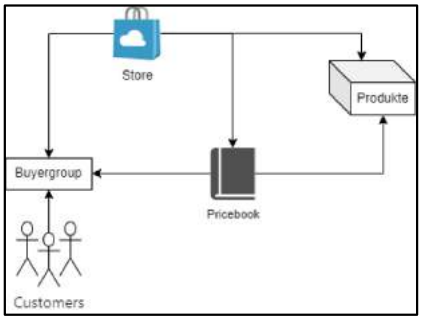

Abbildung 3: Store Inhalte

Salesforce stellt nach der Erstellung des Stores folgende Testdaten zur Verfügung:

- Kategorie „Produkte“
- Buyer-Group
- Testprodukte
- Testaccounts
- Price-Book

Als erstes wird einem Testaccount der Status „Enable as Buyer“ hinzugefügt. Salesforce speichert Kundendaten über Accounts, welche wiederrum die einzelnen Käufer als Kontakte beinhaltet. Jedem Kontakt in dem Account muss nun auch der Status „Enable as “ zugewiesen werden, damit dieser die Berechtigungen für den Store hat. Nun kann der Account der Buyer-Group hinzugefügt werden. Dieses Konzept ist in Abbildung 4 zu sehen.

Dem Kontakt benötigt nun das erstellte Permissionset für den jeweiligen Store und zusätzlich dazu das Buyer-Permissionset. Zuletzt müssen folgende Schritte ausgeführt werden, damit der Shop vollständig und damit von einem User genutzt werden kann:

1. Produkte dem Price-Book hinzufügen
2. Produkte dem Commerce-Entitlement-Policies hinzufügen
3. Dem Store die Buyer-Group hinzufügen
4. Suchindex bauen
5. Den Store aktivieren

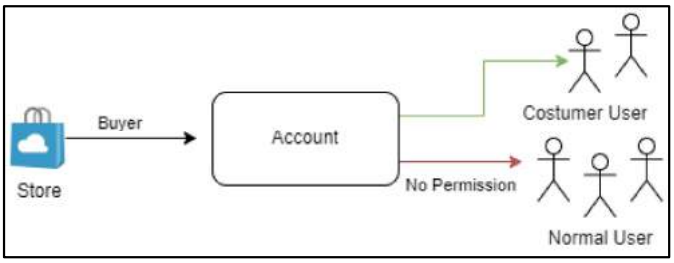

Abbildung 4: Account Sample

4.2 Einrichten des Spring Boot Projektes

Um eine Spring Boot-Anwendung zu initialisieren, gibt es unter der Website start.spring.io ein Konfigurator mit diesem ein Projekt erstellt und konfiguriert werden kann. Für dieses Projekt wurde die Programmiersprache Java und das Build-Tool Maven gewählt. Maven ist eine Projektmanagement-Software und überminnt die Erstellung, Kompilierung und Berichterstattung eines Software-Projekts.[5]

Wie bereits in Kapitel 2.4 beschrieben wird für das Projekt das Framework Thymeleaf genutzt und muss daher auch als Erweiterung hinzugefügt werden. Weitergehend wird Spring Web und Spring Web Services hinzugefügt. Damit kann das in Kapitel 3.1 beschriebene Konzept abgebildet werden, da Spring Web die Funktionen zur Erstellung von Controllern, Services und REST beinhaltet. Spring Web Services wurde gewählt, um die Authentifizierung für Salesforce zu realisieren. Diese Konfiguration wurde auf Grundlage eines Spring Boot-Guides[6] gewählt.

4.3 Logik der Auth

Salesforce verwendet OAuth für seine REST-Requests (siehe Kapitel 2.2) und benötigt einen gültigen Token für jede Request. Um den Token für einen Nutzer zu erhalten, muss eine SOAP-Request an die Salesforce SOAP-API gesendet werden, die die Organisations-ID, den Benutzernamen und das Passwort des Users enthält. Salesforce bietet hierfür einen zentralen Authentifizierungspunkt unter folgender URL an: https://login.salesforce.com/services/Soap/u/56.0

[5] Vgl. [APA23]
[6] Vgl. [APE23]

Ist die Anfrage erfolgreich, enthält die Antwort eine SessionID (Token) und weitere User-Informationen. In der Klasse Salesforce-Service wird eine Methode implementiert, welche die SOAP-Request mit einem XML-Body (Extensible Markup Language) an Salesforce sendet und den Access-Token in einer Variable speichert. (Siehe Abbildung 5)

```
1. public static void login() throws UsernameNotFoundException
2. {
3.
4.          HttpHeaders headers = new HttpHeaders();
5.          headers.setContentType(MediaType.TEXT_XML);
6.          headers.add("SOAPAction", "login");
7.
8.          HttpEntity<String> entity = new HttpEntity<>(setXML(), headers);
9.          RestTemplate restTemplate = new RestTemplate();
10.
11.         try
12.         {
13.          String response = restTemplate.postForObject(LOGIN_URL, entity, String.class);
14.          DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
15.          InputSource src = new InputSource();
16.          src.setCharacterStream(new StringReader(response));
17.          Document doc = builder.parse(src);
18.
19.          NodeList sessionIdNode = doc.getElementsByTagName("sessionId");
20.
21.          if (sessionIdNode.getLength() > 0) {
22.                 Node node = sessionIdNode.item(0);
23.                 accessToken = node.getTextContent();
24.          }
25.
26.         } catch (Exception e)
27.         {
28.            throw new UsernameNotFoundException("Authentication error");
29.          }
30.}
```

Abbildung 5: Logik der Authentisierung

4.4 Logik der REST-Schnittstelle

4.4.1 Salesforce Service

Die Salesforce-Service-Klasse implementiert die zentrale Funktion, um HTTP-Anfragen an die Salesforce API zu senden und die Antwort an die entsprechenden Dienste weiterzuleiten. Dies beinhaltet zudem die Request-Authentifizierung, die in Kapitel 4.3 erwähnt wurde. Die Klasse enthält zusätzlich eine Funktion zur Realisierung der API-Request, die als Parameter die URL, den Body und die geforderte HTTP-Methode akzeptiert.

Zunächst wird eine Instanz der Klasse RestTemplate erstellt, die für die Verwaltung von REST API-Requests verantwortlich ist und von Spring Boot zur Verfügung gestellt wird. Für die Request wird im Header nur ein Parameter (der Authentifizierungstoken) benötigt.
Der übergebene Body und der erstellte Header werden einer Anfrageklasse hinzugefügt, die die HTTP-Anfrage repräsentiert.

Schließlich wird die exchange-Methode von RestTemplate aufgerufen, um die API-Request auszuführen, indem URL, HTTP-Methode und Anfrageklasse als Argumente übergeben werden. Bei erfolgreicher Anfrage gibt die Methode eine ResponseEntity zurück, die die benötigten Daten enthält. Ist die Anfrage nicht erfolgreich, liest die jeweilige Service-Klasse den Fehler aus und gibt diesen aus. Sollte aus irgendeinem Grund der OAuth-Token nicht zur Verfügung stehen, wird nochmals die Authentifizierung-Funktion aufgerufen. (Siehe Abbildung 6)

```
 1. public static ResponseEntity<String> salesforceApiCall(String url, String body, HttpMethod
method)
 2.{
 3.         RestTemplate restTemplate = new RestTemplate();
 4.         HttpHeaders headers = new HttpHeaders();
 5.
 6.         if (accessToken == null || accessToken.equals("")) login();
 7.
 8.         headers.setContentType(MediaType.APPLICATION_JSON);
 9.         headers.set("Authorization", "Bearer " + accessToken);
10.
11.         HttpEntity<String> entity = new HttpEntity<String>(body, headers);
12.         return restTemplate.exchange(MAINURL + url, method, entity, String.class);
13.}
```

Abbildung 6: Salesforce API-Request

4.4.2 Produkt Service

Um an die Produktdaten eines Webstores zu gelangen, gibt es viele Möglichkeiten. Dabei werden grundsätzlich die jeweiligen IDs der Produkte, Kategorien oder Kataloge benötigt. Da diese Arbeit nur die Funktionalität des Konzeptes überprüfen soll, wird die bereits angelegte Kategorie (die mit Produkten gefüllt ist) als Abfrageobjekt benutzt. Die Abfrage der Produkte erfolgt dabei mit einer POST-Request über folgende URL:

$$< \text{Instanz} > /\text{services/data/v56.0/commerce/webstores/} < \text{WebstoreId} > /\text{search/product} - \text{search}$$

Da diese URL als Standard für jede andere Abfrage gilt, wird in den nachfolgenden Kapiteln der URL-Teil nach der WebstoreId angegeben. Der Request-Body enthält lediglich die categoryId. Die Salesforce Dokumentation beschreibt hierbei, dass ein Suchterm als Parameter übergeben werden sollte. Wird dieser Weggelassen, gibt die Suche alle Werte in der Kategorie zurück. Ist die Response erfolgreich wird mittels der in Kapitel 3.2 erwähnten GSON-Bibliothek der Response-Body in eine neue Instanz der Produkt-Klasse umgewandelt, in eine Liste gespeichert und zurückgegeben:
Für einen Shop ist die Funktion nach einzelnen Produkten zu suchen sehr nützlich. Deshalb wird eine zweite Funktion in die ProduktService-Klasse integriert, welche den Webstore über die folgende URL das jeweilige Produkt abfragt und zurückgibt. Der Syntax erfolgt gleichwertig nach der Beschreibung der vorherigen Funktion. (Siehe Abbildung 7)

$$/\text{products}/< \text{productid} >$$

```
 1. public static List<Products> getAllProducts()
 2. {
 3.   Gson gson = new Gson();
 4.   JSONObject jsonObject = new JSONObject();
 5.   jsonObject.put("categoryId", "0ZG68000000LRpYGAW");
 6.
 7.   ResponseEntity<String> response = SalesforceService.salesforceApiCall("/search/product-
search", jsonObject.toString(), HttpMethod.POST);
 8.
 9.   if(response.getStatusCode().isError())
10.   {
11.       throw new HttpClientErrorException(response.getStatusCode());
12.   }
13.   else
14.   {
15.       ProductRequest request = gson.fromJson(response.getBody(),ProductRequest.class);
16.       List<Products> mylist = Arrays.asList(request.getProductsPage().getProducts());
17.
18.       return mylist;
19.   }
20.}
```

Abbildung 7: Produkt-Abfrage

4.4.3 Cart Service

Der Warenkorb ist das zentrale Element des Shops und muss somit viele Funktionen abbilden können. Daher ist die Implementierung dieser Klasse komplexer ausgefallen. Da die CartService-Klasse für jede Funktion eine ähnliche Syntax benutzt und sich grundsätzlich in der Abfrage-Methode und der URL unterscheidet, soll dabei nur ein Abbild äquivalent zur Beschreibung der anderen Funktionen genutzt werden. Wird ein Objekt von der jeweiligen Abfrage benötigt, wird dieses mit der bereits genannten GSON-Bibliothek umgewandelt und zurückgegeben. Eine Übersicht der Funktionalitäten ist in Tabelle 2 zusehen und wird hierbei nicht weiter beschrieben, da die Implementierung der einzelnen Funktionen ähnlich und nach dem gleichen Schema wie in den anderen Services abläuft. Hierbei wird der Salesforce-Service mit der jeweiligen URL, der Methode und dem Body aufgerufen. Die Antwort wird danach, wenn nötig interpretiert.

Bei dem Hinzufügen eines Produktes zum Warenkorb prüft Salesforce intern, ob bereits ein Warenkorb existiert. Daher muss für diesen Fall keine Absicherung implementiert werden. Fügt man ein Produkt zu einem nicht existenten Warenkorb hinzu, erstellt Salesforce diesen Warenkorb.

Url	Http Methode	Body	Response	Beschreibung
/carts/active/	GET		Cart-Objekt	Ruft den aktuellen Warenkorb ab
/carts/active/	PUT		Cart-Objekt	Erstellt einen neuen Warenkorb
/carts/active/	DELETE			Löscht den aktuellen Warenkorb
/carts/active/cart-items	GET		Cart-Objekt	Ruft alle Produkte im Warenkorb ab
/carts/active/cart-items	POST	Produkt-Menge, ID, Type		Fügt dem Warenkorb ein Produkt hinzu
/carts/active/cart-items/<id>	POST	Menge		Ändert die Menge eines Produktes
/carts/active/cart-items/<id>	DELETE			Löscht ein Produkt aus dem Warenkorb
/carts/active/cart-items/<id>	GET			Gibt das jeweilige Produkt zurück

Tabelle 3: Übersicht Order Funktionen

4.4.4 Order Service

Die OrderService-Klasse benutzt den gleichen Syntax der Funktionen für den Cart-Service und umfasst folgende Funktionen. Um das Angeben der Informationen im Zusammenhang mit dem Status des Check-outs zu setzen, ist in Abbildung 10 auf der nächsten Seite der gesamte Ablauf mit den jeweiligen Schritten gekennzeichnet. Warum nun nur auf bestimmte Schritte zugegriffen wird, soll in einem späteren Kapitel aufgegriffen werden.

Den aktuellen Warenkorb auf Check-out setzen und den aktuellen Check-out abbrechen

Sendet eine POST-Anfrage über den Salesforce-Service mit der URL `/checkouts/active`, um mit dem aktuellen Warenkorb einen Check-out zu starten (1). Hierbei wird die aktuelle CartId des Warenkorbs im Request-Body benötigt. Salesforce intern wird hierbei ein neues Objekt erstellt, welches die aktuelle Check-out-Session repräsentiert. Um diese zu beenden, wird auf die gleiche URL eine DELETE-Request gesendet.

Setzen der Check-out-Informationen

Die Anfragen erfolgen über die die gleich URL. Um die aktuelle Lieferadresse zu setzen, muss ein JSON-Body erstellt werden, welcher dem Salesforce-Syntax entspricht (2).

Dieser beinhaltet ab Version 57.0 der API mindestens die Informationen Name, City, Street, Country und optional Lieferanweisungen, Shipping-Date und einen Vor- und Nachnamen. Das setzten erfolgt über eine PATCH-Request.
Ist diese Adresse gesetzt, aktualisiert Salesforce die aktuelle Check-out-Session auf Billing und setzt zudem automatisch eine vorgegebene Liefermethode. Nun kann über eine POST-Request die Zahlungsinformationen übergeben werden. Dafür wird wiederum ein Request-Body erstellt, welcher einen Payment-Token, die Authentifizierungsmethode und die Rechnungs-Adresse beinhaltet (Siehe Abbildung 8).

```
{
   "paymentToken":"AF045455-69B0",
   "requestType":"Auth",
   "billingAddress":{
      "name":"John Doe",
      "street":"123 Acme Drive",
      "city":"Los Angeles",
      "region":"California",
      "regionCode":"CA",
      "country":"United States",
      "countryCode":"US",
      "postalCode":"90001"
   }
}
```

Abbildung 8: Setzen der Bezahlmethode
Quelle: [SAL23]

Der Payment-Token für die aktuelle Zahlungsweise wird über die die URL: payments/token/ abgefragt. Diese POST-Request benötigt wiederum einen JSON-Body, welcher wie in Abbildung 9 zu sehen, aussehen könnte (3).

```
{
   "cardPaymentMethod":{
      "cardHolderName":"John Doe",
      "cardNumber":"4242424242424242",
      "expiryMonth":"12",
      "expiryYear":"24",
      "cvv":"141",
      "cardType":"Visa"
   }
}
```

Abbildung 9: JSON-Body Bezahlmethode
Quelle: [SAL23]

Aufgeben der Bestellung

Um die aktuelle Checkout-Session abzuschließen, wird an die URL: /checkouts/active/orders eine POST-Request ohne Body gesendet. Dies startet in Salesforce einen Prozess, welcher die aktuelle Cart mit den übergebenen Informationen in eine Order umwandelt (4). Damit ist der Bestellprozess abgeschlossen.

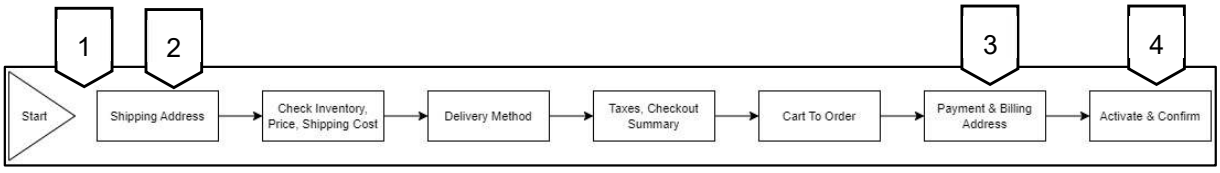

Abbildung 10: Checkout Flow

4.5 Logik der Spring Boot Schnittstelle

Die Logik der Schnittstelle unterteilt sich in 3 Teile, welche jeweils die Anfragen für die Produkte, den Warenkorb oder die Order regeln. Diese Aufgabe übernehmen die jeweiligen Controller (Siehe Kapitel 3.1). Um die Funktionen der Services zu nutzen, implementiert jeder Controller eine neue Instanz des benötigten Services.
Außerdem ist zu beachten das zu jeder Klasse @Controller hinzugefügt werden muss. Dadurch erkennt Spring Boot, dass es sich um eine Controller-Klasse handelt, welche die jeweilige URL abfängt und verarbeitet.

4.5.1 Produkt Controller

Da sich die ProduktController-Klasse nur um das Anzeigen der Produkte und das hinzufügen des jeweiligen Produktes zum Warenkorb kümmert, fällt die Komplexität dieser gering aus. Wenn die URL `/home` aufgerufen wird, wird eine GET-Request an das Backend gestellt und über die ProduktService-Klasse dem Model die Liste der aktuellen Produkte hinzugefügt. Zurückgegeben wird das Home-Page-Template. Das Hinzufügen zum Warenkorb wird über eine POST-Request an die gleiche URL geregelt. Dabei wird nach einem Klick auf den „add-to-Cart-Button“ die POST-Request mit dem aktuellen Produkt als Body und als Parameter die Anzahl gesendet. Dieses wird danach über die CartService-Klasse hinzugefügt und die Home-Page wieder zurückgegeben.

4.5.2 Cart Controller

Die CartController-Klasse muss insgesamt 4 Funktionen erfüllen. Angefangen mit dem Aufruf des Warenkorbs über die URL `/cart` wird der aktuelle Warenkorb aus Salesforce abgerufen. Ist kein aktiver Warenkorb vorhanden, wird einer erstellt. Dieser wird über die getCartItems-Methode aus der CartService-Klasse abgerufen und dem Model hinzugefügt. Zurückgegeben wird das Cart-Template. Über die URL `/cart/delete` kann der aktuelle Warenkorb gelöscht werden.

Hinzufügen und Entfernen eines Produktes aus einem Warenkorb

Die Aktion update erfolgt über eine POST-Request an die URL /updateCartItem. Dabei werden als Parameter die Produktid und gewünschte Anzahl übergeben. Mit diesen wird dann die Funktion editCartItem aus dem Cart-Service aufgerufen und die Warenkorb-Seite wieder zurückgegeben. Dagegen benötigt das Entfernen eines Produktes lediglich die ID als Parameter und wird über die URL /removeFromCart/{id} aufgerufen und gibt auch das Cart-Template zurück. (Siehe Abbildung 11)

```
 1. public static void editCartItem(String id, String quantity)
 2. {
 3.
 4.    if(quantity.equals("0")) removeCartItems(id);
 5.
 6.    JSONObject jsonObject = new JSONObject();
 7.    jsonObject.put("quantity", quantity);
 8.
 9.    SalesforceService.patchRequest("/carts/active/cart-items/"+id, jsonOb-
ject.toString());
10. }
11.
12. public static void removeCartItems(String id)
13. {
14.       ResponseEntity<String> response = SalesforceService.salesforceApiCall("/carts/cur-
rent/cart-items/"+id,"", HttpMethod.DELETE);
15.
16.       if(response.getStatusCode().isError())
17.       {
18.             throw new HttpClientErrorException(response.getStatusCode());
19.       }
20. }
```

Abbildung 11: Update & Remove from Cart

4.5.3 Order Controller

Die Implementierung des Controllers hat sich auf 5 Funktionen beschränkt. Dabei ist darauf zu achten, dass die richtige Reinfolge[7] der API-Requests an Salesforce eingehalten wird.

Wie bereits in Kapitel 4.4.4 erwähnt erstellt Salesforce für eine Check-out-Session ein neues Objekt, bei dem der jeweilige Status die benötigten Daten repräsentiert. Ist beispielsweise die aktuelle Session nicht im richtigen Status, wird bei einer API-Request die syntaktisch richtig ist eine Fehlermeldung zurückgegeben.

Setzten und abbrechen eines Check-outs

Das Erstellen eines neuen Check-outs erfolgt über eine GET-Request an die URL /checkout. Dabei wird zuerst geprüft, ob der aktuelle Warenkorb leer ist. Ist dies der Fall, wird eine Fehlermeldung dem Model hinzugefügt und dem Nutzer angezeigt. Ist der Warenkorb korrekt, wird ein Hilfsobjekt „Address“ hinzugefügt und die startCheckout-Funktion der OrderService-Klasse aufgerufen.

[7] Vgl. Kapitel 4.4.4

Das Frontend erkennt dieses Objekt und kann dieses nach befüllen der Eingabefelder, welche sich auf dem zurückgegebenen Template befinden, wieder zurück an das Backend geben. Um den aktuellen Check-out abzubrechen kann die URL `/cancelCheckout` aufgerufen werden. Diese Funktion ruft die in der OrderService-Klasse zugehörige Funktion auf.

Setzten der Check-out Informationen und Abschluss

Nach dem das zugehörige Formular für die Adresse ausgefüllt wurde wird über einen Button die URL `/setOrder` über eine POST-Request mit dem Adress-Objekt im Request-Body aufgerufen und überprüft. Ist die Eingabe falsch wird die aktuelle Seite zurückgegeben und der jeweilige Fehler angezeigt. Ist diese wiederum richtig wird dem Model ein weiteres Hilfsobjekt, welches die Adresse und Zahlungsinformationen beinhaltet hinzugefügt. Zurückgegeben wird das Template zur Eingabe der Zahlungsinformationen. Vorher wird jedoch über den Order-Service die Adresse an die aktuelle Checkout-Session gesendet. Hierfür wurde eine Hilfsfunktion gebaut, welche das Adress-Objekt in das benötigte JSON-Format bringt (siehe Abbildung 12)

```
 1. public static String transformAddress(Address address)
 2. {
 3.         JsonObject deliveryAddress = new JsonObject();
 4.         deliveryAddress.addProperty("name", address.getName());
 5.         deliveryAddress.addProperty("country", address.getCountry());
 6.         deliveryAddress.addProperty("city", address.getCity());
 7.         deliveryAddress.addProperty("street", address.getStreet());
 8.         deliveryAddress.addProperty("postalCode", address.getPostalCode());
 9.
10.         JsonObject output = new JsonObject();
11.         output.add("deliveryAddress", deliveryAddress);
12.         output.addProperty("shippingInstructions", address.getShippingInstructions());
13.
14.         return output.toString();
15. }
```

Abbildung 12: Adress-Konvertierungsfunktion

Die Eingabe der Zahlungsinformationen sowie die Rechnungsadresse erfolgt sinngemäß zum Setzen der Adresse und wird auch hier über eine POST-Request mit dem jeweiligen Objekt im Request-Body übergeben und über eine Konvertierungsfunktion an Salesforce übergeben. Sind alle Daten korrekt wird nach der Übermittlung der Prozess mit einem Aufruf der placeOrder-Funktion aus der OrderService-Klasse abgeschlossen (siehe Abbildung 13).

```
 1. @PostMapping("/payment")
 2. public String setPayment(@ModelAttribute("PaymentMethod") PaymentMethod paymentMethod,
BindingResult bindingResult)
 3. {
 4.   OrderService.setPayment(Transformers.transformPayment(paymentMethod, addressSafe));
 5.
 6.   OrderService.setOrder();
 7.
 8.   return "redirect:/home";
 9. }
```

Abbildung 13: Überprüfen der Zahlungsinformationen und platzieren der Bestellung

4.6 Logik des Frontend

Um die notwendigen Daten aus dem Backend anzuzeigen wird, wie bereits in den vorherigen Kapiteln erwähnt, das jeweilige Objekt einem Model hinzugefügt. Thymeleaf interpretiert dieses dann mit den jeweiligen Funktionen so, dass der Browser es anzeigen kann.

Hauptseite

Um die Vielzahl von Produkten anzuzeigen, reicht es aus, wenn dem Model die Liste von Produkten hinzugefügt wird. Über den Befehl im Eltern-Element th:each = "product:${productList}" verarbeitet Thymeleaf die Liste so, dass für jedes Produkt ein Identisches Element erstellt wird. Dem Form-Element wird über th:action die dazugehörige URL für die POST-Request und über th:objekt das Objekt hinzugefügt. Über die jeweiligen Kind-Elemente werden dann die Attribute des Objektes ausgelesen.
Damit nach Absenden des Formulars auch die ID des Objektes übergeben werden kann, wurde ein Hilfs-Element erstellt, welches die ID des jeweiligen Produktes beinhaltet. Über das Attribut „Name“ kann Thymeleaf dieses Objekt mit den jeweiligen Werten befüllen. Dabei ist darauf zu achten, dass dieses Objekt die notwendigen Attribute besitzt. (Siehe Abbildung 14)

```
<div class="card product" style="width: 18rem;" th:each="product : ${allProductList}">
   <img src="tree-736885__480.png" class="card-img-top" alt="...">
   <div class="card-body">
   <form th:action="@{/home}" th:object="${product}" method="POST">
          <h5 class="card-title" th:text="${product.name}"></h5>
          <p class="card-text" th:text="${product.id}"></p>
          <input type="hidden" th:value="${product.id}" name="id"/>
          <div style="display: flex; align-items: flex-start; justify-content: center;">
                   <button type="submit" >Add to cart</button>
                   <input style="width:50px; margin-left: 10px;" type="number" name="quan-
tity" placeholder="0" min="1"/>
          </div>
   </form>
</div>
```

Abbildung 14: Template für die Hauptseite

Warenkorb

Für die Übersicht der Produkte, die sich im Warenkorb befinden wurde für die Übersichtlichkeit eine Tabelle erstellt. Im Body dieser Tabelle wurde auch hier für jedes Produkt eine neue Reihe kreiert. Die Reihe beinhaltet dann für jede Spalte der jeweilige Name, die aktuelle Anzahl und den Preis. Wiederum existiert für jedes Produkt ein Button, welcher die URL /removeFromCart/{id} mit der jeweiligen ID des Produktes aufruft, um dieses aus dem Warenkorb zu entfernen. Am Ende wurde ein Input-Element hinzugefügt, welches die aktuelle Anzahl des Produktes im Warenkorb beinhaltet. Verändert man diesen, kann über einen Button die URL /updateCartItem aufgerufen werden. Hierbei wird dann der jeweilige Wert und die ID übergeben (siehe Abbildung 15).

Abbildung 15: Beispiel Warenkorb

Check-out

Wird der Button „Start Checkout“ angeklickt, wird eine GET-Request an die URL `/checkout` gesendet. Ist nach Maßgabe des in Kapitel 4.5.3 beschriebenen Prozesses die angefragte Ressource verfügbar, wird das Eingabefeld für die Adresse aufgerufen (Siehe Abbildung 16). Ist die Eingabe hierbei wiederum richtig, wird das in Abbildung 17 zu sehende Formular aufgerufen. Hier kann der Nutzer eine Zahlungsart auswählen (hier Kreditkarte) und die jeweiligen Informationen eingeben. Sind alle Informationen gültig, wird die Bestellung platziert. Möchte man einen Check-out abbrechen, wird über den Button die URL `/cancelCheckout` aufgerufen.

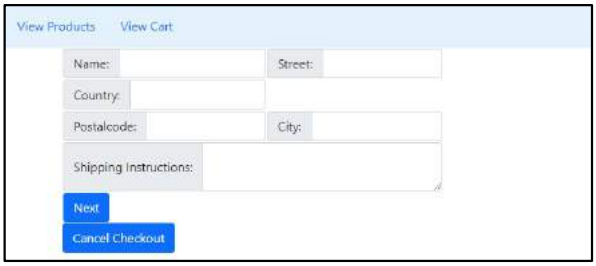

Abbildung 17: Adress-Input

Abbildung 16: Zahlungsart-Input

4.7 Test, Analyse und Fazit

Das Testen der App ist für die Fehleranalyse essenziell. Hierbei können, vor allem bei größeren Projekten, mögliche Fehler ausgeschlossen und verschiedene Use-Cases abgedeckt werden. Abgeglichen wurde hierbei die in Kapitel 3.2 aufgezeigten Anforderungen sowie die Use-Cases (zu sehen in Tabelle 4).

Für die Erfüllung der ersten Anforderung wurde eine Klasse geschrieben, welche sich global um die Salesforce Requests kümmert. Diese wurde so aufgebaut, dass andere Services diese aufrufen und so gebrauchen können wie benötigt. Durch die globale Nutzung dieser Klasse und der jeweiligen Fehlerbehandlung kann, sobald die anderen Use-Cases abgedeckt wurden, diese Anforderung als getestet und erfüllt angesehen werden.

Dies gilt auch für die Anforderungen zwei, drei und vier, da die Daten aus der Salesforce Abfrage interpretiert und an das Frontend über eine API-Schnittstelle weitergeleitet werden.
Durch die Unabhängigkeit der jeweiligen Klassen (vor allem die Services) könnten diese auch in anderen Projekten benutzt werden, welche beispielsweise auf einem anderen Framework für das Frontend basieren. Damit ist auch die letzte Anforderung erfüllt.

Nun werden die Use Cases überprüft. Bei dem Aufruf der Home-Seite werden die jeweiligen Produkte, die in dem jeweiligen Salesforce Shop vorhanden sind, angezeigt. Dies ist auch über einen Login für verschiedene Nutzer möglich, welche damit unterschiedliche Produkte sehen können. Hierfür wurde ein Login implementiert, mit dem sich ein Nutzer mit seinen Anmeldedaten aus Salesforce anmelden kann. Hat der Nutzer keine Berechtigung oder existiert nicht, werden ihm diese Produkte nicht angezeigt. Auch wurde ein Button hinzugefügt, welcher die angegebene Menge des Produktes dem Warenkorb hinzufügen kann (Siehe Abbildung 18). Somit sind die ersten beiden Use-Cases erfüllt.

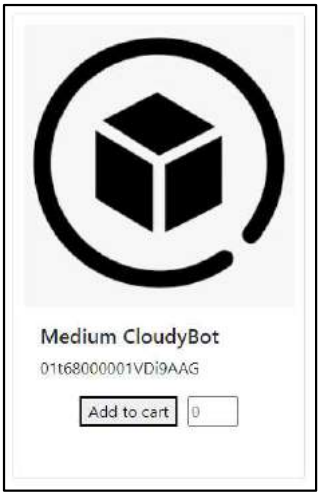

Abbildung 18: Beispiel Produkt-Cart

In vielen Online-Shops können die meisten Nutzer die Produkte angezeigt bekommen, auch wenn diese nicht eingeloggt sind. Dieses Feature wurde nebenbei getestet und es stellte sich heraus, dass nach derzeitigen Stand Gastnutzer sich nicht über die API authentifizieren können, da hierfür ein Passwort nötig ist. Daher wäre eine Überlegung hierbei, ein eigenes Gastnutzer-Profil anzulegen, welcher einmalig die Produktdaten aus Salesforce abfragt und diese auf dem Server zwischenspeichert. So könnten viele Nutzer, ohne sich anzumelden oder eine Menge von API-Requests zu senden, die jeweiligen Produkte sehen. Hierbei müsste aber eine Lösung für das Anzeigen von Produkten aus anderen Ländern gefunden werden. Möchte nun ein Nutzer ein Produkt zum Warenkorb hinzufügen müsste er sich Einloggen und kann so die bereits implementierten Funktionen nutzen.

Ein eingeloggter Nutzer kann zudem seinen Warenkorb betrachten. Loggt man sich aus der aktuellen Session aus und als ein anderer Nutzer wieder ein wird der jeweilige Warenkorb angezeigt. Über die jeweiligen Übersichten kann dieser dann bearbeitet werden. Dies würde das Erstellen unnötiger Daten in Salesforce verhindern. Dabei ist darauf zu Achten, dass jedes Produkt, welches zum Warenkorb hinzugefügt werden soll, auch in Salesforce einen Preis hinterlegt haben muss. Sonst schlägt das hinzufügen fehl, da der Warenkorb den Preis nicht berechnen kann. Desweitern ist darauf zu achten, dass das jeweilige JSON-Format der Request-Bodys eingehalten werden muss. Leider ist die Dokumentation von Salesforce dahingehend fehlführend, da einige dort angegebene Parameter nicht zur Verfügung stehen oder der angegebene Syntax für die API-Request nicht stimmt. Hierfür wurde über Postman[8] (ein API-Tool) die jeweiligen Requests geprüft und so angepasst, bis der angeforderte Body funktioniert hat. Diese Ergebnisse wurden dann in der dotSource Knowledge-Base festgehalten, dass weitere Entwickler diesen Schritt überspringen können und nicht auf diese Probleme stoßen.

[8] Vgl. [POS23]

Hierbei ist vielleicht die Überlegung angebracht, vor allem bei vielen Nutzern, die jeweiligen Warenkörbe und Bestellungen über die jeweilige ID und nicht über das Keyword aktiv abzufragen. Damit könnten die Objekte eindeutig zugeordnet und Überschneidungen vermieden werden. Damit sind auch die Use Cases 3 und 4 erfüllt.

Anforderungen	Erfüllt
Kommunikation über die Salesforce API	✓
Interpretieren der Daten aus Salesforce	✓
Authentifizierung der Salesforce abfrage	✓
Bereitstellen einer API-Schnittstelle über die Front- und Backend kommunizieren	✓
Frontend und das Backend sind vollständig voneinander entkoppelt	✓
Use Cases	
1. Als Nutzer möchte ich auf der Hauptseite Produkte angezeigt bekommen	✓
2. Als Nutzer möchte die Möglichkeit haben Produkte in der gewünschten Menge dem Warenkorb hinzuzufügen	✓
3. Als Nutzer möchte ich eine Möglichkeit haben die Menge eines Produktes im Warenkorb zu ändern oder zu entfernen	✓
4. Als Nutzer möchte ich eine Übersicht über alle Produkte haben, die ich bisher ausgewählt habe	✓
5. Als Nutzer möchte ich die Möglichkeit haben meine Kontaktdaten und Lieferadresse einzugeben	✓
6. Als Nutzer möchte ich die Möglichkeit haben eine Bestellung mit der ausgewählten Zahlungsart aufzugeben	✓

Tabelle 4: Anforderungsabgleich

Möchte der Nutzer nun die zum Warenkorn hinzugefügten Produkte bestellen kann dieser mittels eines Buttons eine Bestellung auslösen. Zuerst muss jedoch in Salesforce folgende Einstellungen getroffen werden, damit ein Nutzer einen Check out ausführen kann. Benötigt wird hierbei ein Payment-Gateway welches ankommende Zahlungsarten autorisiert und einen Authentifizierungs-Token zurückgibt. Mit diesen autorisiert Salesforce, und damit das Gateway, eingehende Zahlungen. Hierfür wurde ein Gateway implementiert, welches nach außen eine vollwertige Authentifizierung zurückgibt, aber nach innen nur als Test-Klasse fungiert und keine Requests an z.B. einen externen Serviceanbieter stellt. Hierbei ist aber zu erwähnen, dass dieses Gateway nur als Authentifizierung für die Anfrage genutzt wird. Salesforce erstellt hierbei intern einen Token. Dieser Prozess ist jedoch ohne weiteres nicht einsehbar. Dieses Gateway sollte bei einem Kunden aber nicht implementiert werden, da der erhaltene Token nur zu Testzwecken verwendet wird. Auch muss über den sogenannten „Experience Builder“ ein Check-out Flow implementiert werden, damit die vom Nutzer eingegebenen Daten verarbeitet werden können[9]. Hierfür wurde ein Flow benutzt, welcher von Salesforce als ein Check-out Flow-Template vorgegeben wurde. Dieser ist auch in Abbildung 10 zu sehen.

[9] Ein Salesforce Flow kann bestimmte Prozesse automatisieren und im Hintergrund automatisch Aufgaben erledigen, die sonst über eine aufwendige API-Request getätigt werden müssen.

Ist das Gateway integriert müssen nun Services implementiert werden welche die Berechnungen für z.B. die Liefermethode, aufkommende Steuern oder die aktuell verfügbaren Waren prüfen. Hierfür wurden, ähnlich wie bei dem Payment-Gateway, Klassen integriert, welche diese Aufgaben simulieren. Durch diese Integration kann ein Nutzer nun eine vollwertige Bestellung aufgeben.

Der Nutzer kann im Headless-Shop nun die Adresse, seine Zahlungsart und Rechnungsadresse eingeben. Damit sind auch die letzten zwei Use-Cases abgedeckt und der Prototyp fertig gestellt. Weiterhin ist aber bei der Umsetzung eines solchen Konzeptes darauf zu achten, dass ein System aufgebaut wird, welches die Verfügbaren Adress-Daten überprüft, um Fehler auszuschließen. Beispielsweise benötigt die angegebene Region ein bestimmtes Format, damit Salesforce dieses erkennt. Auch wurde über eine dieser Test-Klassen eine Liefermethode händig festgelegt. Sendet man nun die API-Request zum Festlegen einer Lieferadresse, setzt Salesforce diese automatisch fest. Hierbei wäre es noch möglich zu prüfen, wie man diese Liefermethode über die API steuern könnte, da oftmals für verschiedene Produkte andere Liefermethoden zur Verfügung stehen.

Zusammenfassend lässt sich sagen, dass das größte Problem in der Salesforce Dokumentation liegt. Diese ist zu diesem Thema nur begrenzt bis gar nicht verfügbar und teilweise fehlleitend, was die Implementation erschwert, da immer wieder Probleme auftreten, zudem keine Lösungsmöglichkeiten angeboten wurden. Dies widerlegt dabei die Aussage aus Kapitel 3.3. Auch wurde nach der Fertigstellung des Shops ein Update seitens Salesforce ausgespielt. Danach konnte der Headless-Shop nicht vollumfänglich benutzt werden, da die API-Requests nicht mehr akzeptiert wurden. Dieses Problem konnte aber schnell behoben werden. Für zukünftige Kundenprojekte ist daher darauf zu achten mögliche Projekte vor einem Salesforce-Update auf ihre Komptabilität zu prüfen und rechtzeitig Anpassungen vorzunehmen, um mögliche Ausfälle zu vermeiden. Informationen über Updates können den Salesforce-Release-Notes entnommen werden. Auch kann eine Testumgebung aus dem aktuellen Projekt erstellt und damit die Funktionalität und die Features getestet werden. Außerdem wurden die Ergebnisse in der dotSource-Knowlagebase festgehalten, damit sich Entwickler bei zukünftigen Projekten an diesem Beispiel orientieren können, um die bereits gelösten Probleme zu umgehen. Damit bietet sich außerdem die Möglichkeit einer Aufwandsschätzung. Es ist aber darauf zu Achten, dass die Implementation dieses Proof of Concept nur als Anregung und Orientierung dienen soll. Zwar bietet dieses System einige wiederverwendbare Komponenten (z.B. SalesforceService-Klasse und die jeweiligen Models), aber für ein Kundensystem müssen viele weitere Aspekte z.B. Sicherheit und Belastbarkeit berücksichtigt werden.

5 Abschließende Worte

Es lässt sich sagen, dass die prototypische Umsetzung eines decoupled Frontend ein vielversprechender Ansatz ist, um die Performance und Flexibilität von E-Commerce-Plattformen zu verbessern. Durch die Entkopplung von Backend- und Frontend-Funktionalitäten können Änderungen an der Benutzeroberfläche unabhängig von der Backend-Logik vorgenommen werden, was die Wartbarkeit und Skalierbarkeit des Systems erhöht.

Durch das in dieser Arbeit erstellte System ist ein Anwendungsfall entstanden, in dem Commerce-Daten über die Salesforce REST-API abgerufen und erstellt werden. Darüber hinaus können andere Entwickler die erworbenen Kenntnisse modular nutzen, um ihre eigenen Projekte wie z.B. Headless-Shops umzusetzen. Diese Grundlage kann auch als Proof of Concept für zukünftige Kundenprojekte dienen, jedoch müssen die spezifischen Anforderungen jedes Projekts berücksichtigt werden. Es ist jedoch sicher, dass Schnittstellen auch in Zukunft ein wichtiges Thema für Software-Projekte bleiben werden und dass die Salesforce-Plattform mit dieser Grundlage verwendet werden kann.

Literaturverzeichnis

Internetquellen

[AMA23] Amazon Web Services Inc. 2023. [Online] 23. Januar 2023. https://aws.amazon.com/de/what-is/api/.

[APA23] Apache Software Foundation. [Online] [Zitat vom: 19. Januar 2023.] https://maven.apache.org/.

[CRP23] Crownpeak Technology GmbH. 2023. [Online] 23. Januar 2023. https://www.e-spirit.com/de/blog/headless_hybrid_und_decoupled_eine_kurze_einfuehrung_in_die_wichtigsten_cms_konzepte.html.

[GIT23] Google LLC. [Online] [Zitat vom: 19. Januar 2023.] https://github.com/google/gson/blob/master/UserGuide.md.

[OKT23] Okta inc. [Online] [Zitat vom: 16. März 2023.] https://auth0.com/de/intro-to-iam/what-is-oauth-2.

[POS23] Postman inc. [Online] [Zitat vom: 16. März 2023.] https://www.postman.com/.

[SAL23] Salesforce Inc. [Online] [Zitat vom: 19. Januar 2023.] https://developer.salesforce.com/docs/atlas.en-us.chatterapi.meta/chatterapi/intro_what_is_chatter_connect.htm.

[SFH23] Salesforce Inc. 2023. [Online] 23. Januar 2023. https://www.salesforce.com/de/.

[SLA23] Salesforce Inc. [Online] [Zitat vom: 19. Januar 2023.] https://help.salesforce.com/s/articleView?id=sf.comm_configure_org.htm&type=5.

[THY23] Thymeleaf. 2023. [Online] 23. Januar 2023. https://www.thymeleaf.org/.

[TRA23] Salesforce inc. [Online] [Zitat vom: 03. März 2023.] https://trailhead.salesforce.com/de.

[VMW23] VMware Inc. [Online] [Zitat vom: 19. Januar 2023.] https://spring.io/.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Projektarbeit mit dem Thema:

Entwurf und prototypische Umsetzung eines decoupled Frontend für einen Onlineshop auf Basis von Salesforce B2B Commerce

ohne fremde Hilfe angefertigt habe,

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

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

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

Jena 23.03.2023

_____	_____
Ort, Datum	Unterschrift