HAW HAMBURG

Bachelorarbeit

Talal Tabia

Konzeption und Entwicklung eines Gutscheinsystems für ein mittelständisches Einzelhandelsunternehmen unter Verwendung von Microservice-Architekturen

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Talal Tabia

Konzeption und Entwicklung eines Gutscheinsystems für ein mittelständisches Einzelhandelsunternehmen unter Verwendung von Microservice-Architekturen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 07. Februar 2019

Talal Tabia

Thema der Arbeit

Konzeption und Entwicklung eines Gutscheinsystems für ein mittelständisches Einzelhandelsunternehmen unter Verwendung von Microservice-Architekturen

Stichworte

Microservice, Gutscheinsystem, Docker, Softwarearchitektur

Kurzzusammenfassung

Das Ziel dieser Arbeit ist die Konzeption und Entwicklung eines Systems, welches dem Unternehmen P&C die Möglichkeit gibt, Gutscheine zu verwalten. Dabei werden Microservices und REST als Grundarchitekturen verwendet. Der Service soll den externen Nachbarsystemen seine Nutzung mithilfe von Schnittstellen ermöglichen und eine bestehende Datenbank für die Speicherung der Daten verwenden. Die Arbeit beschäftigt sich mit der Beschreibung und Umsetzung der gewählten Architektur und deren Entwurf sowie die Bewertung aller eingesetzten technischen Aspekte.

Talal Tabia

Title of Thesis

Conception and development of a voucher system for a medium-sized retail company under the use of microservice architectures

Keywords

Microservice, Voucher System, Docker, Software Architecture

Abstract

The aim of this work is the conception and development of a system which gives the company P&C the possibility to manage vouchers. Microservices and REST are used as architectural basis. The service should enable the external neighboring systems to use it with the help of interfaces and use an existing database for data storage. The thesis deals with the description and implementation of the chosen architecture and design as well as the evaluation of all technical aspects.

Inhaltsverzeichnis

Abbildungsverzeichnis **v**

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Zielsetzung . . . 1
1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **4**
2.1 Microservice-Architektur . . . 4
2.2 RESTful-API und Sicherheit . . . 5
2.3 Containerisierung von Anwendungen mit Docker . . . 6
2.4 Spring-Frameworks . . . 7

3 Anforderungsanalyse und Spezifikation **8**
3.1 Analyse . . . 8
3.1.1 Nachbarsysteme . . . 8
3.1.2 User-Rollen . . . 9
3.1.3 User-Stories . . . 10
3.2 Spezifikation . . . 12
3.2.1 Use-Cases . . . 12
3.2.2 Geschäftsprozesse . . . 17
3.2.3 Fachliches Datenmodell . . . 18

4 Entwurf und Architektur **20**
4.1 Architektursichten . . . 20
4.1.1 Kontextsicht . . . 20
4.1.2 Bausteinsicht . . . 21
4.1.3 Laufzeitsicht . . . 26
4.1.4 Verteilungssicht . . . 30

4.2 Entwurf 31
4.2.1 SOLID-Prinzipien 31
4.2.2 Schnittstellenentwurf 32

5 Implementation **35**
5.1 Geschäftslogik 35
5.1.1 Entitäten 35
5.1.2 Persistenz der Entitäten 37
5.2 REST Schnittstellen 38
5.3 Einstellung der Applikation 40
5.4 Docker 41

6 Evaluierung **42**
6.1 Unit-Tests 42
6.2 Integrationstests 43
6.3 REST-Schnittstellentests 44
6.4 Performanz-Tests 46
6.5 Überblick über die Umsetzung 46

7 Fazit **47**
7.1 Bewertung 47
7.2 Ausblick 48

Selbstständigkeitserklärung **52**

Abbildungsverzeichnis

3.1 BPMN Freund und Rücker (2014): Gutscheinarten verwalten 18
3.2 BPMN Freund und Rücker (2014): Gutscheine verwalten 18
3.3 Fachliches Datenmodell. 19

4.1 Kontextsicht. 21
4.2 Bausteinsicht: Level1. 22
4.3 Bausteinsicht: Level2. 23
4.4 Bausteinsicht: Level3 - Teil1. 24
4.5 Bausteinsicht: Level3 - Teil2. 25
4.6 Sequenzdiagramm: Generieren und Aufladen von Gutscheinen 27
4.7 Sequenzdiagramm: Abfragen und Entladen eines Gutscheins 28
4.8 Sequenzdiagramm: Sperren und Entsperren eines Gutscheines 29
4.9 Verteilungssicht 30
4.10 Aufbau der Kommunikation zwischen Client und Server 32
4.11 Swagger-Spezifikation für die Verwaltung von Gutscheinen 33
4.12 Swagger-Spezifikation für die Verwaltung von Gutscheinarten 34

1 Einleitung

1.1 Motivation

Gutscheine sind eine Form von Zahlungsmitteln, die bei Einzelhändlern als Marketinginstrumente benutzt werden. Sie dienen vor allem der Neukundengewinnung und werden bereits seit dem 19. Jahrhundert in Papierform eingesetzt. Der Geschäftsmann Griggs Candler hatte als Erster die Idee, handgeschriebene Zettel für Gratisgetränke zu verschenken. Seit den 1990er Jahren werden Gutscheine auch in digitaler Form angeboten. In der Regel werden Gutscheine mit Hilfe von Codes identifiziert und verfügen über einen Geldwert, der eingelöst werden kann.
Um die Qualität eines Unternehmens zu steigern und seinen Erfolg zu beschleunigen, werden immer mehr Arbeitsprozesse automatisiert - mithilfe von digitalen Gutscheinsystemen können Gutscheine flexibler und effizienter verwaltet werden.
Automatisierte Prozesse verringern sowohl den Aufwand als auch die Kosten, die während der Be- und Verarbeitung von Gutscheinen entstehen. Durch die große Zahl von Transaktionen pro Jahr verursacht die Nutzung eines externen Gutscheinsystems hohe Gebühren. Daher soll im konkreten Fall ein internes, eigenständiges und flexibles System entwickelt werden, welches einen Großteil der bisher extern verwalteten Gutscheine ablösen kann.
Um ein solches System zu entwickeln, muss vorher analysiert werden, welche Anforderungen das System erfüllen muss und wie diese am besten zu erreichen sind.

1.2 Zielsetzung

Ziel der Arbeit ist der Entwurf und die Umsetzung einer Microservice-Architektur zur Verwaltung unterschiedlicher Gutscheinarten, die im Rahmen des *„P&C Voucher Management“* zum Einsatz kommen sollen.

Auf diese Weise sollen die Kosten, die infolge der Transaktionsgebühren für das Aufladen, Abfragen und Entwerten der Gutscheine entstehen, reduziert werden.
Der Fokus liegt momentan bei den Gutscheinen, die in den Kundenkartensystemen benutzt werden. Diese werden dem Kunden als Welcome- oder Geburtstagsgutschein geschickt und als Zahlungsmittel betrachtet. Es wird auch gewünscht, dass außerhalb des Kundenkartenbereichs weitere Gutscheinarten verwaltet werden können.
Zu den Gutscheinen sollen diverse Informationen hinterlegt werden können, die den Gutschein an sich ausmachen (z.B. Gutscheintypen, Laufzeit, Gutscheinhöhe, etc.).
Verschiedene Transaktionen werden bei den Gutscheinen benötigt, unter anderem Gutscheine prägen, entladen und abfragen.
Die Entwicklung des Gutscheinsystems soll mithilfe von unterschiedlichen Technologien durchgeführt werden (*Spring*, *Java*, *DB2*, *Docker*, *etc.*).

1.3 Aufbau der Arbeit

Kapitel 1: Einleitung

Im ersten Kapitel werden Ziel und Zweck des gewählten Projekts erläutert, sowie dessen Vorteil, welcher vielen Unternehmen nutzen könnten.

Kapitel 2: Grundlagen

Im diesem Kapitel werden die gewählten Softwarearchitekturen und die eingesetzten Technologien erläutert.

Kapitel 3: Anforderungsanalyse und Spezifikation

Dieses Kapitel definiert im Analyse-Teil die Erwartungen der potentiellen Nutzer an das System und welche Nachbarsysteme mit der Anwendung interagieren werden. In der Spezifikation werden Anwendungsfälle, Geschäftsprozesse und das fachliche Datenmodell beschrieben.

Kapitel 4: Entwurf und Architektur

Im vierten Kapitel wird die Architektur der Anwendung mittels verschiedener Sichten erläutert und wichtige Entwurfsmuster beschrieben. Zusätzlich werden die Schnittstellen des Systems entworfen.

Kapitel 5: Implementation

Im Kapitel Implementation wird die Umsetzung der entworfenen Architektur durch Code-Ausschnitte und Erläuterungen der benutzten Technologie erklärt.

Kapitel 6: Evaluierung

Im vorletzten Kapitel wird das Testen während der Entwicklung erläutert. Anhand von Code-Ausschnitten wird gezeigt, wie Tests definiert werden. Zusätzlich wird ein Überblick über die erfüllten Anforderungen gegeben.

Kapitel 7: Fazit

Zum Schluss werden die bei der Arbeit eingesetzten Technologien bewertet und ein Ausblick über mögliche Erweiterungen des entwickelten Services gezeigt.

2 Grundlagen

2.1 Microservice-Architektur

Um Produkte schneller und flexibler entwickeln zu können, werden in der Softwareentwicklung immer häufiger Microservice-Architekturen eingesetzt. Jeder *Microservice* ist eine kleine Systemeinheit, die so aufgebaut ist, dass sie von einem einzigen Team entwickelt, deployed und betrieben werden kann.
Microservices sind primär ein organisatorisches Muster und oft nur zum kleinen Teil ein Architekturmuster. Der Einsatz von Microservices sorgt dafür, dass sich verschiedene Teams anstelle von Technologie auf Fachlichkeiten fokussieren. Die homogenen Teams arbeiten funktionsübergreifend - sie sind also für ein bestimmtes Produkt von der Konzeption bis zur Produktion verantwortlich. Oder wie Amazon es ausdrückt: *„You build it, your run it“* (vgl. Starke, 2017, S.341). Diese Philosophie unterstützt das Gesetzt von Conway: „Organisationen, die Systeme entwerfen, sind gezwungen, Entwürfe zu erstellen, die die Kommunikationsstrukturen dieser Organisationen abbilden (Conway, 1968, vgl.).
Im Gegensatz zu Microservices steht das monolithische System, das als eine einzige, in sich geschlossene Einheit entwickelt wird. Erweiterungen sind aufwendig, kleine Änderungen erzwingen eine Aktualisierung des gesamten Systems.
Microservices konzentrieren sich dagegen auf die Entwicklung von Modulen, die nur auf einer fachlichen Funktion basieren. Jedes dieser Module ist ein eigener Service mit klar definierten Schnittstellen(Fowler, 2014, vgl.). Das erweitert die Flexibilität der Software, in dem jeder einzelne Service in einem eignen Prozess unabhängig und einfach ausgeführt werden kann. Im Unterkapitel 2.4 wird erklärt, wie ein Microservice mittels Containerisierung und Docker deployt werden kann.
Microservices bauen auf *Representational State Transfer* (REST), welches ein Programmierparadigma bzw. ein Architekturstil für die Kommunikation zwischen verteilten Systemen und Webservices ist. Mit REST profitieren Systeme während der Datenübertragung von der zustandslosen Kommunikation. Das bringt den Vorteil, dass Services keine

Sitzungsverwaltung betreiben müssen. Darüber hinaus erfolgt die Datenübertragung in einem einzigen Seitenaufruf. Im kommenden Kapitel dieser Arbeit wird REST genauer beschrieben.
Wie jedes System müssen Microservices auch vor unerwünschten Zugriffen sicher sein. Erhöht man allerdings den Schutz der Daten, z. B. indem man die Zugriffsmöglichkeiten einschränkt, hat das normalerweise Auswirkungen auf andere Eigenschaften des Systems, wie zum Beispiel auf die Benutzbarkeit und somit auch auf die Produktivität (vgl. Starke, 2017, S.276 f.). Die Sicherheit eines Microservice beginnt mit Schnittstellen. Wie das erreicht wird, wird im nächsten Kapitel erklärt.

2.2 RESTful-API und Sicherheit

Webdienste bieten mit Hilfe einer vordefinierten Programmierschnittstelle kurz (*API*) Systemfunktionalitäten nach außen an. Auf Basis eines Standard Protokolls HTTP (*Hypertext Transfer Protocol*) oder HTTPS für ein sicheres Kommunikationsprotokoll ermöglichen Webdienste eine Maschine-zu-Maschine-Kommunikation und werden über einen standardisierten Mechanismus Uniform Resource Identifier (URI) identifiziert.
RESTful basiert auf REST und ist ein Architekturstil für Webtechnologien, die über APIs kommunizieren.
Wie wird ein REST(ful) Service implementiert? Dafür gibt es das Richardson Maturity Modell, ein von Leonard Richardson entwickelter Maßstab, welcher die Hauptelemente eines REST-Ansatzes in drei Schritte unterteilt. Dadurch werden Ressourcen für die primäre Datendarstellung, HTTP-Verben und Hypermedia-Steuerelemente eingeführt(Fowler, 2010, vgl.).
Die Abschlussarbeit fokussiert auf das Prinzip der Stufe drei des Models, das sogenannte „RESTfull“[1]. Eine RESTful API profitiert von den Vorteilen des HTTP-Protokolls, welches im RFC (2019) definiert ist. Das Protokoll definiert bestimmte Regeln für den Einsatz der HTTP-Verben; Eine neue Ressource wird mit *POST* erstellt, mit *DELETE* gelöscht, mit *GET* abgerufen und mit UPDATE verändert. Außerdem wird zusätzlich das Entwurfsprinzip HATEOAS (*Hypertext As The Engine Of Application State*) eingesetzt; es definiert die Einführung von *Hypermedia* mit denen Informationen über Hyperlinks vernetzt werden. Der Client bekommt beim Aufruf einer bestimmten REST-Schnittstelle Informationen über den nächsten Zustand mitgeteilt und somit ist klar was als nächstes

[1] RESTfull ist eine Bezeichnung für konkrete Architekturen, die Webtechnologien so benutzen, wie es die Prinzipien des Architekturstils vorschreiben (Starke, 2017, S.130).

aufgerufen werden kann.
Der Einsatz von REST eignet sich ganz gut mit Microservices, da diese von den zustandslosen Aufrufen profitieren, um einen komplett unabhängigen Betrieb und eine problemlose Skalierung zu sichern Wikpedia (2018).
Konkrete Beispiele werden in der Implementierung vorgeführt (Siehe Kapitel 5).

REST-APIs können mithilfe von Basic Authentification, einem HTTP-Header Standard, den Zugriff auf bestimmte Ressourcen bzw. Schnittstellen einschränken. Diese Technik verschlüsselt über das Base64 Kodierungsverfahren die Authentifizierung zu einem Server. Wenn die Anmeldedaten über eine nicht verschlüsselte Verbindung übertragen werden, kann sich ein *Man-in-The-Middle* einschalten und unbemerkt die ausgetauschten Daten aufzeichnen. Um sich gegen solche Art von Angriffen zu schützen, wird das bereits in diesem Kapitel erwähnte Kommunikationsprotokoll HTTPS verwendet. Dieses bietet eine zusätzliche Sicherheitsschicht, um die zu transportierenden Daten zu verschlüsseln. Nur so ist eine Authentifizierung einer REST-API sicher.
Diese Verfahren und Techniken werden bei der Implementierung des Gutscheinsystems eingesetzt.

2.3 Containerisierung von Anwendungen mit Docker

Container sind eine Verkapselung einer Anwendung und ihrer Abhängigkeiten. Die Technologie bietet den Vorteil, dass Anwendungen schneller und zuverlässig von einer Umgebung zu anderen laufen(vgl. Mouat, 2016, S.3).
Um eine aktive Instanz eines Containers zu erzeugen, wird ein Speicherabbild (*Image*) benötigt. Dieses beinhaltet alles was benötigt wird, um eine Anwendung zu starten. Dazu gehören Code, Laufzeit, Systemwerkzeuge, Systembibliotheken und Einstellungen.
Bei Containern und virtuellen Maschinen (*VMs*) findet sich eine isolierte Instanz eines Betriebssystems, in dem Anwendungen laufen. Doch die Technologien unterscheiden sich in ihrer Funktionsweise, weil Container das Betriebssystem an Stelle der Hardware virtualisieren, was mehr Portabilität und Effizienz bietet. Docker ist eine Open-Source-Software und basiert auf Linux-Containertechnologie. Sie bietet durch portable Images und eine benutzerfreundliche Schnittstelle, eine vollständige Lösung für das Erstellen und Verteilen von Containern.
Im letzten Prozess des praktischen Teils der Arbeit wird mit Hilfe von Docker versucht,

die geschriebene Microservice-Anwendung auf einem Container laufen zu lassen (Docker, 2018, vgl.).

2.4 Spring-Frameworks

Spring ist ein Open-Source-Framework (kurz *Spring*) für die Java-Plattform. Mit Spring lassen sich Java-Projekte einfacher entwickeln und erweitern. Außerdem hilft Spring Applikationskomponenten zu entkoppeln. Eines der wichtigsten Prinzipien in Spring ist das Entwurfsmuster *Dependency Injection*, welches im Kapitel 4.2 genauer beschrieben wird. Spring ist die Kerntechnologie, die im Projekt eingesetzt wird. Weitere Projekte von Spring werden ebenfalls verwendet. Dies sind unter anderem:
Spring Boot: Erweitert das Spring Framework durch das Softwaredesign-Paradigma Konvention vor Konfiguration, um mit minimalem Aufwand eigenständige und lauffähige Anwendungen zu entwickeln (Spring-Boot, 2019, vgl.).
Spring Data: Erleichtert die Verwendung von Datenzugriffstechnologien, relationalen und nicht relationalen Datenbanken.
Spring Security: Bietet diverse Authentifizierungs- und Zugriffssteuerungs-Funktionalitäten an, mit der man mit geringem Aufwand die Anwendung sichert.
Spring HATEOAS: Bietet einige APIs, um die Erstellung von REST-Repräsentationen zu vereinfachen, die dem im Kapitel 2.2 vorgestellten HATEOAS-Prinzip folgen.

3 Anforderungsanalyse und Spezifikation

3.1 Analyse

Um ein Entwicklungsprojekt zum Erfolg führen zu können, muss zunächst bekannt sein, was die Anforderungen an das System sind und diese müssen geeignet dokumentiert sein (Rupp und Pohl, 2015, vgl.).

In diesem Kapitel werden die Anforderungsquellen (Nachbarsysteme, Stakeholder) systematisch betrachtet und ihre Wünsche und Anforderungen mittels User-Stories gesammelt.

3.1.1 Nachbarsysteme

Das Gutscheinsystem wird von unterschiedlichen Nachbarsystemen aufgerufen, um aus unterschiedlichen Quellen Gutscheintransaktionen durchführen zu können.
Es sind drei Nachbarsysteme zu betrachten:

- Das *Kassensystem* wird vom Unternehmen Ratio entwickelt. Die Kommunikation mit dem P&C System erfolgt über Schnittstellen, die im *PuCRatio-Programm* (*Ratio-Gateway*) definiert sind.
- Der P&C *Onlineshop* wird vom Unternehmen *Novomind* entwickelt und benötigt den Zugriff auf das Gutscheinsystem, um bei Kundenbestellungen Gutscheine einlösen zu können.
- Das *CRM-System* nutzt der Kundenservice auch für die Verwaltung der Kundengutscheine. Hierfür werden bei dem Gutscheinsystem Gutscheine abgefragt und verändert.

- Das *WAWI-System* ist ein internes Backend-System der P&C Warenwirtschaftsprozesse, dieses benötigt einen Zugriff, um einen Vorrat von Gutscheinrohlingen zu erstellen oder für das Kampagnen-Management-System (*KMS*) Gutscheine aufzuladen.

3.1.2 User-Rollen

Eine Nutzerrolle ist eine Menge von Attributen, die eine Population von Nutzern und ihre beabsichtigten Interaktionen mit dem System beschreibt Cohn (2004).

Rolle 1: Verkäufer
Ein Verkaufsmitarbeiter kann für einen Kunden bei der Zahlung an der Kasse einen Gutschein einlösen.
Nach dem Einscannen oder der manuellen Eingabe der Gutscheinnummer wird eine Verbindung von der Kasse zum Gutscheinsystem über Ratio-Gateway eröffnet, um die Gültigkeit des Gutscheins zu prüfen und die Gutscheindaten abzurufen. Das Gutscheinsystem liefert eine Antwort auf die Anfrage an das Kassensystem zurück. Der Mitarbeiter kann bei Erfolg den Befehl geben, den Gutschein zu entladen.
Rolle 2: Kunde
Ein Kunde kann im externen Onlineshop einen erworbenen Gutschein einlösen. Die Gutscheinnummer und die dazugehörige PIN werden beide vom Kunden eingegeben. Dabei verbindet sich der Onlineshop mit dem Gutscheinsystem und vermittelt die eingegebenen Daten vom Kunden. Das Gutscheinsystem liefert bei Erfolg die dazugehörigen Gutscheininformationen zurück. Der Onlineshop kann anschließend den entsprechenden Gutscheinwert in der Bestellung abrechnen.
Rolle 3: Prüfer
Ein Mitarbeiter an der Kassenkontrolle (*Treasury*) prüft über eine Weboberfläche die Gutscheinhistorie einer Kasse ab, führt Änderungen an einem Gutschein durch und liest dessen Status, um Kassenvorgänge zu kontrollieren.
Rolle 4: Entwickler
Mitarbeiter der IT-Anwendungsentwicklung benötigen einen kompletten Zugriff, um Gutscheine in anderen Systemen zu integrieren.
Der Entwickler möchte eine Menge von leeren Gutscheinen einer bestimmten Art generieren, um diese später an Kunden zu vergeben und anschließend mit einem bestimmten Betrag aufzuladen.

Rolle 5: Kundenservice Mitarbeiter
Ein Kundenservice Mitarbeiter führt Änderungen im Zusammenhang von Kundenkonten (*CRM*) Gutscheine aus. Zusätzlich möchte er den Status eines Ersatz-Gutscheins ändern, Kundengutscheine anzeigen, sperren oder wieder freigeben.

3.1.3 User-Stories

Eine *User-Story* beschreibt eine Funktionalität, die für den Kunden oder Benutzer eines Produkts oder Systems von Wert ist. Sie besteht aus der schriftlichen Beschreibung der Funktionalität, Gesprächen über die Funktionalität und Akzeptanzkriterien, die Details vermitteln und festlegen, wann eine User-Story vollständig umgesetzt ist (Cohn, 2004, vgl.).

Rolle: „Kassierer“
Als Kassierer möchte ich einen Gutschein an der Kasse abfragen.
Als Kassierer möchte ich einen Gutschein an der Kasse einlösen.

Rolle: „Kunde“
Als Kunde möchte ich mit einem Gutschein im Onlineshop zahlen.
Als Kunde möchte ich mit einem Gutschein an der Kasse zahlen.

Rolle: „Kassenprüfer“
Als Prüfer möchte ich im Zusammenhang von Kassier-Vorgängen Gutscheine verwalten.
Als Prüfer möchte ich im Zusammenhang von Kassier-Vorgängen Gutscheine einsehen.
Als Prüfer möchte ich im Zusammenhang von Kassier-Vorgängen Gutscheine sperren.
Als Prüfer möchte ich im Zusammenhang von Kassier-Vorgängen Gutscheine entsperren.
Als Prüfer möchte ich im Zusammenhang von Kassier-Vorgängen Gutscheinhistorien abfragen.

Rolle: „Entwickler“
Als Entwickler möchte ich eine Gutscheinart erstellen.
Als Entwickler möchte ich eine Gutscheinart einsehen.

Als Entwickler möchte ich eine Gutscheinart bearbeiten.
Als Entwickler möchte ich eine Gutscheinart löschen.
Als Entwickler möchte ich den Status abgelaufener Gutscheine ändern.

Rolle: „Kundenservice Mitarbeiter“
Als Kundenservice Mitarbeiter möchte ich Kundengutscheine verwalten.
Als Kundenservice Mitarbeiter möchte ich einen Kundengutschein sperren.
Als Kundenservice Mitarbeiter möchte ich einen Kundengutschein entsperren.
Als Kundenservice Mitarbeiter möchte ich einen Kundengutschein verlängern.

Aus den User-Stories ergeben sich folgende *Haupt-Epics*:

- Gutschein verwalten.
- Gutscheinart verwalten.

Die ermittelten *Epics* werden in folgende User-Tasks aufgesplittet:

- Gutschein prägen.
- Gutschein abfragen.
- Gutschein prüfen.
- Gutschein aufladen.
- Gutschein entladen.
- Gutschein sperren.
- Gutschein entsperren.
- Gutschein verlängern.
- Gutschein entwerten.
- Gutscheinart erstellen.
- Gutscheinart abfragen.
- Gutscheinart verändern.
- Gutscheinart löschen.

3.2 Spezifikation

Eine Spezifikation ist eine formalisierte Beschreibung eines Produktes, eines Systems oder einer Dienstleistung.
Eine formalisierte Beschreibung erlaubt die Prüfung, ob die Anforderungen auch das wiedergeben, was die verschiedenen Interessengruppen während ihrer Sammlung und Ermittlung ausgesprochen hatten Ebert (2008).
In diesem Kapitel wird das System durch *Use-Cases*, *BPMNs* und ein Datenmodell fachlich beschrieben.

3.2.1 Use-Cases

Ein *Use-Case* beschreibt eine typische Interaktion eines Anwenders mit einem System, die zu einem vom Anwender gewünschten Ergebnis führt. Es wird in einem Use-Case-Diagramm dargestellt, das meist durch eine textuelle Use-Case-Beschreibung ergänzt wird Rupp und Pohl (2015).

Es werden nur einige Use-Case-Diagramme angezeigt, da es eine hohe Anzahl von Anwendungsfällen entstanden ist.

Use-Case 1: Gutschein an der Kasse abfragen
Akteur: Kassierer
Ziel: Der Kassierer bekommt die Gutscheindaten eines eingelesenen Gutscheines
Auslöser:
Der Kassierer möchte für einen Kunden an der Kasse die Gutscheindaten prüfen.
Der Entwickler möchte Daten eines Gutscheins abfragen.
Vorbedingungen:
Das Format der Gutscheinnummer ist gültig.
Erfolgsszenario:
1: Die Kassierer gibt die Gutscheinnummer ein.
2: Das Gutscheinsystem liefert für die gegebene Gutscheinnummer die dazugehörigen Daten zurück.

Use-Case 2: Gutschein an der Kasse einlösen
Akteur: Kassierer
Ziel: Der Kassierer löst für einen Kunden einen Gutschein an der Kasse ein
Auslöser:
Ein Kunde möchte mit einem Gutschein an der Kasse zahlen.
Vorbedingungen:
Das Format der Gutscheinnummer ist gültig.
Nachbedingungen:
Der Gutschein ist an der Kasse entladen.
Erfolgsszenario:
1: Verkäufer gibt die Gutscheinnummer ein.
2: Das Gutscheinsystem prüft die gegebene Gutscheinnummer: Ist das Format gültig? Ist der dazugehörige Gutschein vorhanden? Ist der Status aktiv? Ist das Gültigkeitsdatum noch nicht abgelaufen?
3: Das Gutscheinsystem liefert die Gutscheindaten zurück.
4: Der Verkäufer prüft die vom System gelieferten Gutscheindaten und bestätigt die Einlösung durch einen erneuten Aufruf.
5: Das Gutscheinsystem entlädt den gegebenen Gutschein.
6: Das Gutscheinsystem trägt die Einlösung in die Historie ein.
Fehlerfälle:
2. a Das Format der eingegebenen Gutscheinnummer ist ungültig: System meldet den Fehler.
2. b Für die gegebene Gutscheinnummer gibt es keinen Gutschein: System meldet den Fehler.
2. c Der Gutscheinstatus ist nicht aktiv: System meldet den Statusfehler.

Use-Case 3: Gutscheinrohlinge erstellen
Akteur: Entwickler
Ziel: Der Entwickler prägt die gewünschte Anzahl an leeren Gutscheinen
Auslöser:
Der Entwickler möchte im Rahmen einer Aktion n Gutscheine generieren.
Vorbedingungen:
Anzahl der Gutscheine ist gültig.
Die Gutscheinart existiert bereits im System.
Nachbedingungen:
Der gewünschte Anzahl der Gutscheine sind generiert.
Erfolgsszenario:
1: Entwickler gibt die gewünschte Anzahl und die Gutscheinart der zu generierenden Gutscheine ein.
2: Das Gutscheinsystem prüft die gegebene Anzahl und Gutscheinart: Ist die Anzahl gültig? Ist die Gutscheinart vorhanden?
3: Das Gutscheinsystem generiert die gewünschten Gutscheine.
4: Das Gutscheinsystem trägt die Einlösung in die Historie ein.
5: Das Gutscheinsystem liefert die Daten der generierten Gutscheine zurück.
Fehlerfälle:
2. a Die eingegebene Anzahl ist ungültig: System meldet den Fehler.
2. b Die eingegebene Gutscheinart ist nicht vorhanden: System meldet den Fehler.

Use-Case 4: Gutscheine aufladen
Akteur: Kundenkarten-Team
Ziel: Das Kundenkarten-Team lädt für einen oder mehrere Kunden einen Gutschein auf
Auslöser:
Das Kundenkarten-Team möchte im Rahmen einer Aktion einen Gutschein an einem Kunden vergeben.
Vorbedingungen:
Die gewünschte Gutscheinart ist gültig.
Die gewünschte Gutscheinmenge ist gültig.
Die gewünschte Gutscheinhöhe ist gültig.
Das Ablaufdatum des zu aufladenden Gutscheines liegt in der Zukunft.
Die gegebene Aufladbarkeit ist gültig.
Nachbedingungen:
Der Gutschein ist für die gewünschte Gutscheinhöhe aufgeladen.
Erfolgsszenario:
1: Kundenkarten-Team gibt die Gutscheinart, Gutscheinmenge, Gutscheinhöhe und das Ablaufdatum ein.
2: Das Gutscheinsystem prüft die gegebene Gutscheinart, Gutscheinmenge, Gutscheinhöhe und das Ablaufdatum: Ist die Gutscheinart-, Menge und Höhe verfügbar? Ist das gegebene Ablaufdatum gültig? Ist die gegebene Aufladbarkeit gültig?
3: Das Gutscheinsystem lädt den Gutschein mit der gewünschten Höhe auf.
4: Das Gutscheinsystem trägt das Aufladen des Gutscheins in die Historie ein.
5: Das Gutscheinsystem bestätigt das Aufladen des Gutscheins.
Fehlerfälle: 2. a Das Format der eingegebenen Gutscheinnummer ist ungültig: System meldet den Fehler.
2. b Die Gutscheinhöhe ist ungültig: System meldet den Fehler.
2. c Der Gutscheinstatus ist nicht inaktiv: System meldet den Statusfehler.
2. d Das Ablaufdatum liegt nicht in der Zukunft: System meldet den Fehler.
2. e Die gegebene Aufladbarkeit ist ungültig: System meldet den Fehler.

Use-Case 5: Gutschein abfragen
Akteur: Prüfer | Kundenkarten-Team
Ziel: Der Prüfer oder das Kundenkarten-Team bekommt für seine Anfrage eine Menge von Gutscheindaten zurück.
Auslöser: Der Prüfer möcht
e Kassenvorgänge kontrollieren und fragt dafür die Gutscheindaten ab.
Der Entwickler möchte Daten eines Gutscheins abfragen.
Vorbedingungen:
Keine.
Nachbedingungen:
Keine.
Erfolgsszenario:
1: Entwickler oder Prüfer geben die Gutscheinnummern ein.
2: Das Gutscheinsystem liefert für die gegebenen Gutscheinnummern die dazugehörigen Daten zurück.

Use-Case 6: Gutscheinart erstellen
Akteur: Entwickler
Ziel: Die Gutscheinart wird erstellt.
Auslöser:
Der Entwickler möchte eine neue Gutscheinart erstellen
Vorbedingungen:
Die gegebene Gutscheinart ist gültig.
Nachbedingungen: Die gegebene Gutscheinart ist hinzugefügt.
Erfolgsszenario:
1: Entwickler gibt eine neue Gutscheinart ein.
2: Das Gutscheinsystem prüft die gegebene Gutscheinart: Ist das Format der Gutscheinart gültig? Existiert keine andere Gutscheinart mit denselben Namen oder Präfix?
3: Das Gutscheinsystem fügt die neue Gutscheinart hinzu.
4: Das Gutscheinsystem bestätigt das Hinzufügen der Gutscheinart.
Fehlerfälle:
2. a Das Format der Gutscheinart ist ungültig: System meldet den Fehler.

Use-Case 7: Gutscheinart verändern
Akteur: Entwickler
Ziel: Die Gutscheinart wird verändert.
Auslöser:
Der Entwickler möchte eine alte Gutscheinart anpassen.
Vorbedingungen:
Die alte Gutscheinart ist gültig.
Die neue Gutscheinart ist gültig.
Nachbedingungen:
Die alte Gutscheinart ist durch die Neue verändert.
Erfolgsszenario:
1: Entwickler gibt eine neue Gutscheinart ein.
2: Das Gutscheinsystem prüft die alte und neue Gutscheinart: Ist das Format der Gutscheinart gültig?
3: Das Gutscheinsystem passt die alte Gutscheinart durch die Neue an.
4: Das Gutscheinsystem bestätigt das Verändern der Gutscheinart.
Fehlerfälle:
2. a Das Format der alten oder neuen Gutscheinart ist ungültig: System meldet den Fehler.

3.2.2 Geschäftsprozesse

Ein Geschäftsprozess ist ein Bündel von Aktivitäten, für das ein oder mehrere Inputs benötigt werden und das für den Kunden ein Ergebnis von Wert erzeugt Rupp und Pohl (2015).
Die Geschäftsprozessmodellierung ist mit dem BPMN Standard durchgeführt worden.
BPMN steht für Geschäftsprozessmodell und -notation und ist eine grafische Spezifikationssprache zur Modellierung von Geschäftsprozessen.

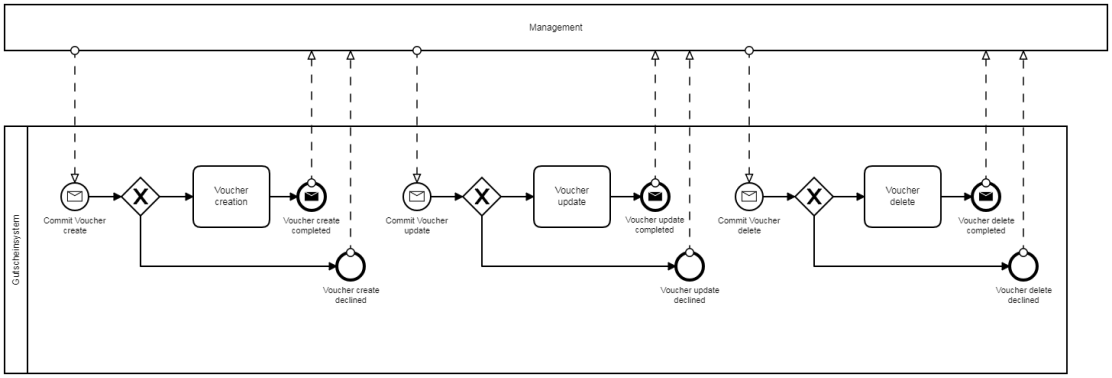

Abbildung 3.1: BPMN Freund und Rücker (2014): Gutscheinarten verwalten

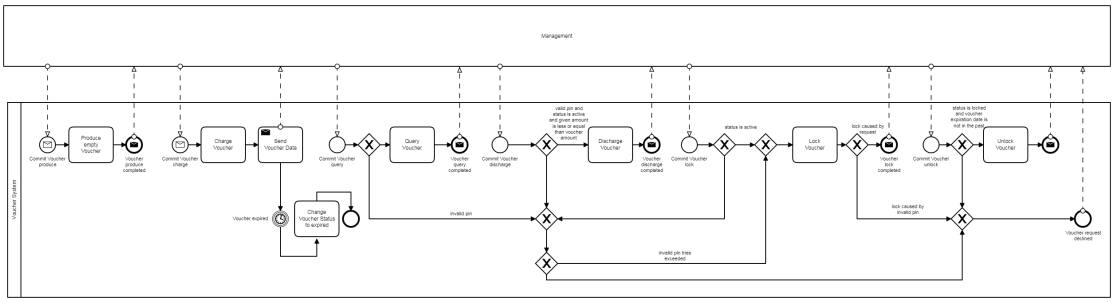

Abbildung 3.2: BPMN Freund und Rücker (2014): Gutscheine verwalten

Die Abbildungen 3.1 und 3.2 stellen die Modelle der Geschäftsprozesse für die Verwaltung von Gutscheinarten und Gutscheinen dar. Der Management-Pool wird dabei als Blackbox betrachtet.

3.2.3 Fachliches Datenmodell

Ein fachliches Datenmodell bietet gute Möglichkeiten, Begriffe und deren Beziehungen zueinander zu beschreiben. Diese Begriffe sind ein Teil der Domäne, die im System verwaltet werden Rupp und Pohl (2015).
In diesem Kapitel werden die Konzepte der Domäne mithilfe des fachlichen Datenmodells beschrieben.

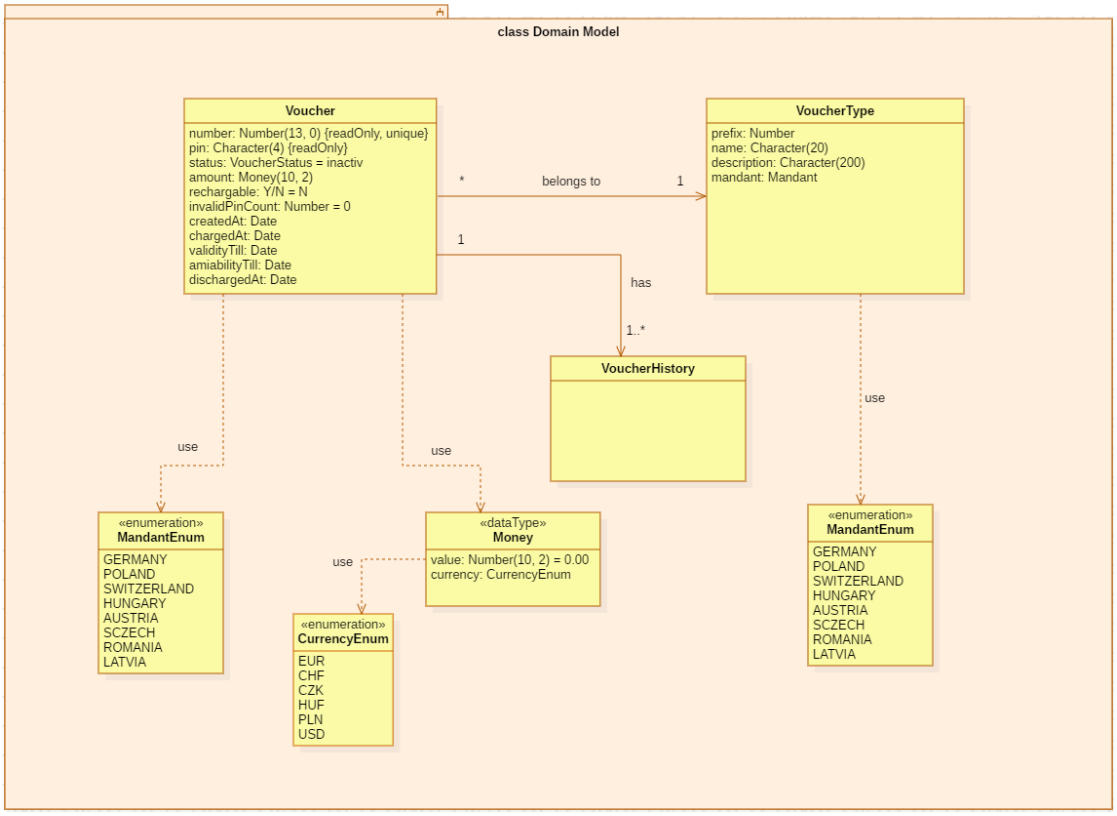

Abbildung 3.3: Fachliches Datenmodell.

Das Datenmodell in der Abbildung 3.3 zeigt die Modellierung der Daten und deren Beziehungen. Sie gelten als *Entitäten* und werden während der Implementierung in der Datenbank abgebildet.

4 Entwurf und Architektur

4.1 Architektursichten

Architektursichten helfen bei der Beschreibung von Softwarearchitekturen und können für jede Art von Projektanforderung verwendet werden. Darüber hinaus dienen sie unterschiedlichen Projektbeteiligten. Eine einzige Sicht drückt allerdings nicht die Vielschichtigkeit und Komplexität einer Architektur aus, daher werden im Folgenden die wichtigsten Sichten genauer betrachtet (vgl. Starke, 2017, S.154 ff.).

4.1.1 Kontextsicht

Eine Kontextsicht oder Kontextabgrenzung stellt das System als Blackbox in seinem Kontext dar, indem sie die Schnittstellen zu Nachbarsystemen, Interaktionen mit wichtigen Stakeholdern sowie die wesentlichen Teile der umgebenden Infrastruktur demonstriert (Starke, 2017, S.158).
Die Kontextsicht wird in der Abbildung 4.1 dargestellt. Diese zeigt die Interaktionen mit den Nachbarsystemen (*Wawi*, *Shop*, *Kasse*) im Paket *Third Party* an.

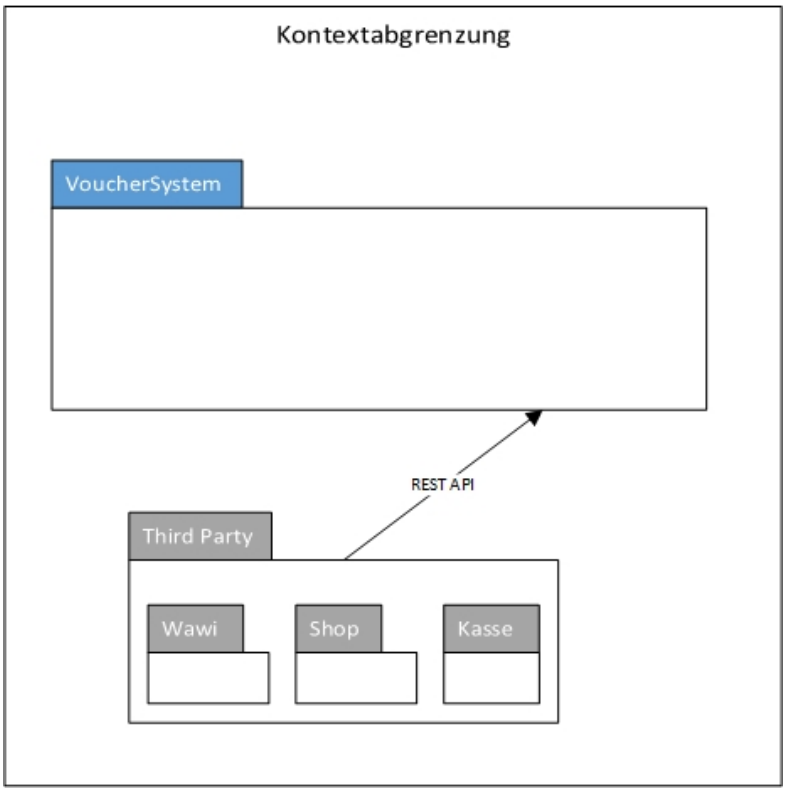

Abbildung 4.1: Kontextsicht.

4.1.2 Bausteinsicht

Mithilfe einer Bausteinsicht wird der interne Aufbau des Systems veranschaulicht, indem die statische Struktur einzelner Architekturbausteine dargestellt wird.
Das Level 1 der Bausteinsicht zeigt eine grobe Übersicht des Systems. Dort befindet sich das Hauptpackage *applicationcore*, welches die REST-Schnittstellen für Nachbarsysteme anbietet. Die Datenbank *DB2* wird vom Package *applicationcore* benötigt, um auf Daten zuzugreifen. (siehe Abbildung 4.2).

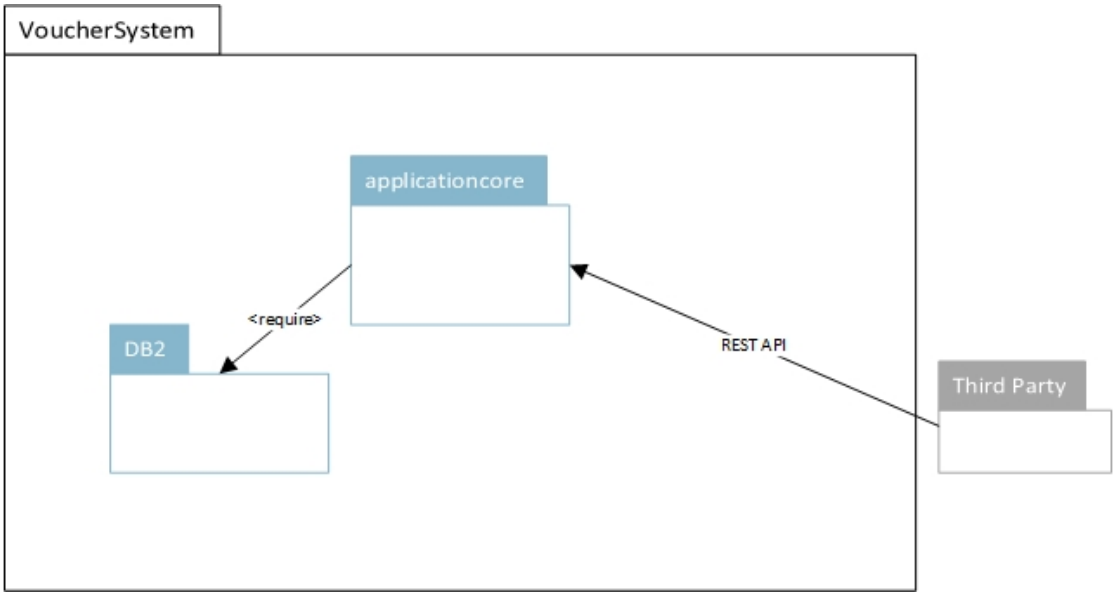

Abbildung 4.2: Bausteinsicht: Level1.

Das Level 2 der Bausteinsicht stellt den Aufbau der *Geschäftslogik* dar. Das Package *services* beinhaltet die *Repositories*, die für die Persistenz der Entitäten zuständig sind. Im Package *usecases* befindet sich die Implementation der Systemfunktionalitäten. Sie benutzen das Package *statemachine*, um Zustandsübergänge für Gutscheine durchzuführen (Siehe Level 3 in der Abbildung 4.5).
Das Package *controllers* bietet über die Controller Klassen alle REST-Schnittstellen des Systems an. Dabei wird auf weitere Packages zugegriffen, z.B auf das Package *services*, um interne Funktionalitäten vom *Interface VoucherService* aufrufen zu können.

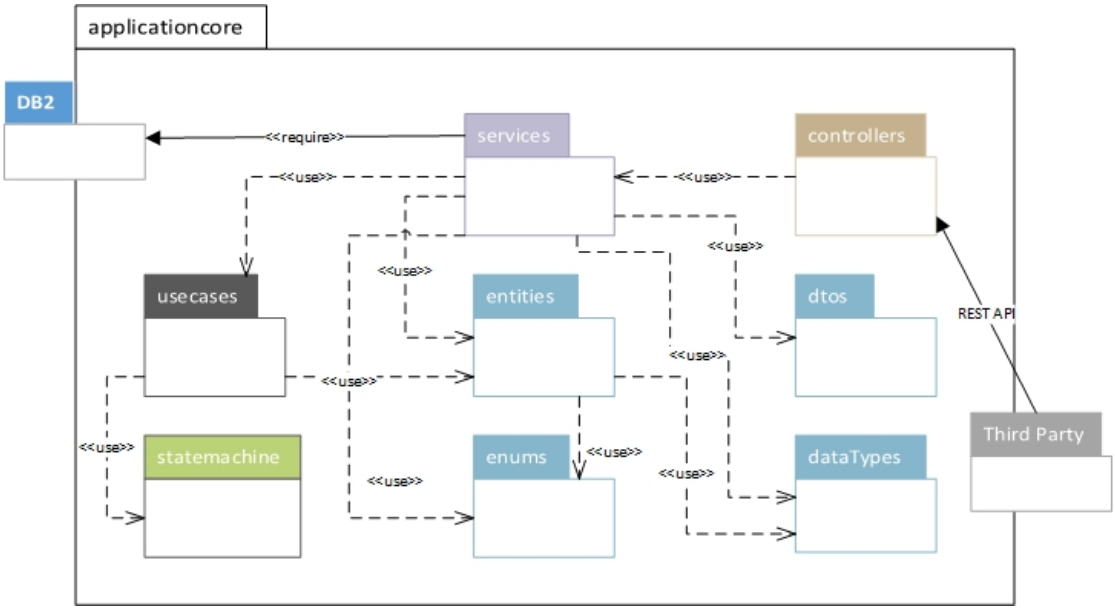

Abbildung 4.3: Bausteinsicht: Level2.

In den Abbildungen 4.4 und 4.5 wird die letzte Verfeinerungsstufe der Bausteinsicht dargestellt. Level 3 zeigt den Aufbau der internen einzelnen Bausteine und wie diese voneinander abhängen.
Die Bausteinsicht entspricht der implementierten Architektur der Packages, Klassen und Interfaces.

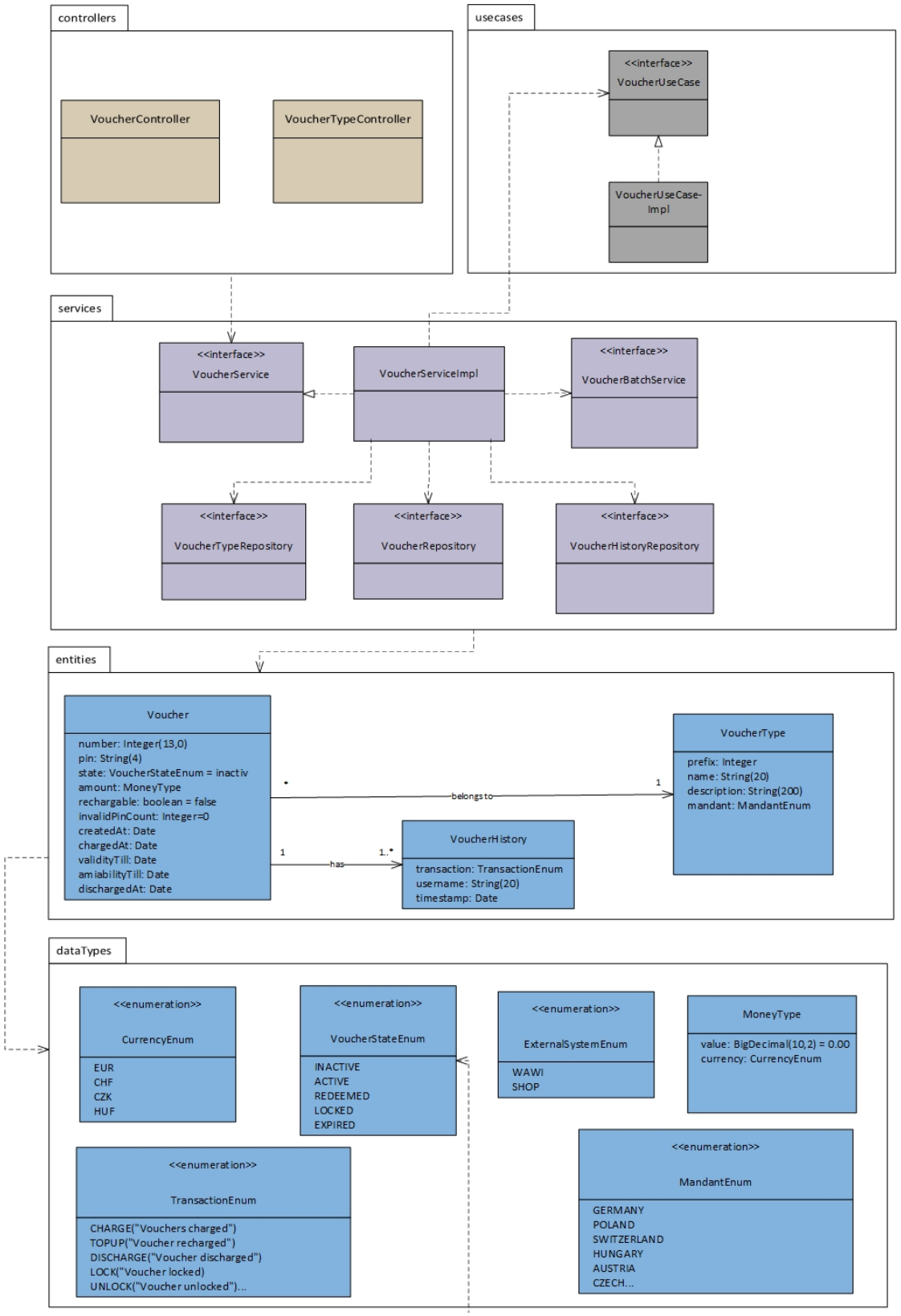

Abbildung 4.4: Bausteinsicht: Level3 - Teil1.

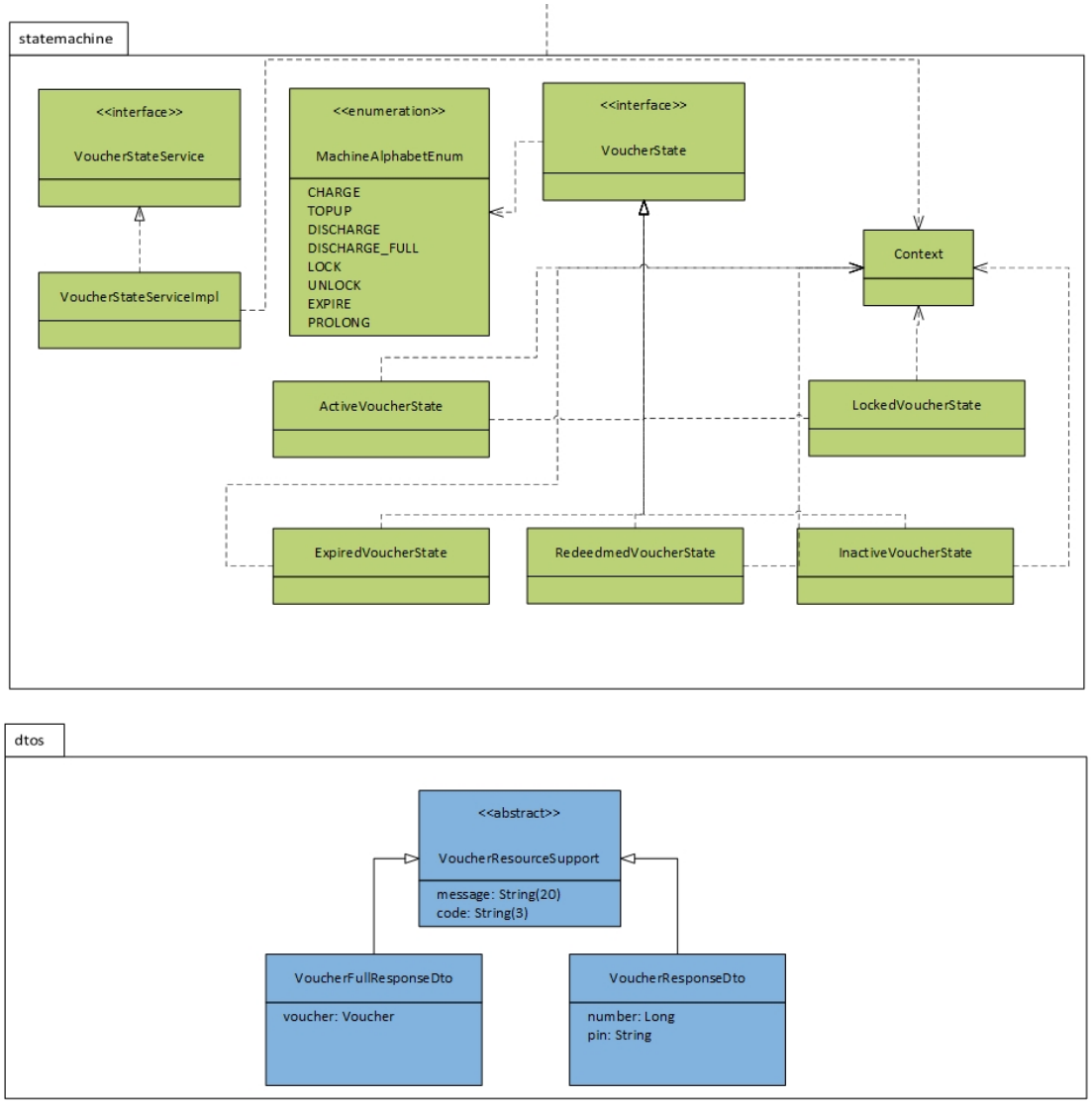

Abbildung 4.5: Bausteinsicht: Level3 - Teil2.

4.1.3 Laufzeitsicht

Die Laufzeitsicht beschreibt wie unterschiedliche Komponenten des Systems zur Laufzeit zusammenwirken (vgl. Starke, 2017, S.158). Der besseren Übersicht halber werden nur die wesentlichen Aufgaben des Systems als Sequenzdiagramme dargestellt.

Das Sequenzdiagramm (Abbildung 4.6) zeigt die unterschiedlichen Methodenaufrufe, die beim Generieren und Aufladen von Gutscheinen ausgeführt werden.
Der Entwickler schickt über das WAWI-System eine REST-Anfrage an den Voucher-Service. Diese wird sofort bearbeitet, indem neue Gutscheine generiert und in der Datenbank gespeichert werden. Bei Erfolg schickt der *VoucherService* zuletzt eine *HTTP-Entität* mit den generierten Gutscheininformationen und den *HTTP-Response 200* zurück. Dem Entwickler werden danach die Gutscheindaten angezeigt. Das Kundenkarten-Team kann dann die Gutscheine über das WAWI-System aufladen. Das WAWI-System ruft die REST-Schnittstelle für das Aufladen auf. Diese wird vom VoucherService bearbeitet, indem er die vorhandenen Gutscheine mit dem gegebenen Geldbetrag auflädt. Zum Schluss wird das Ergebnis dem WAWI-System geliefert und wenige Sekunden später werden dem Kundenkarten-Team die Gutscheindaten angezeigt.

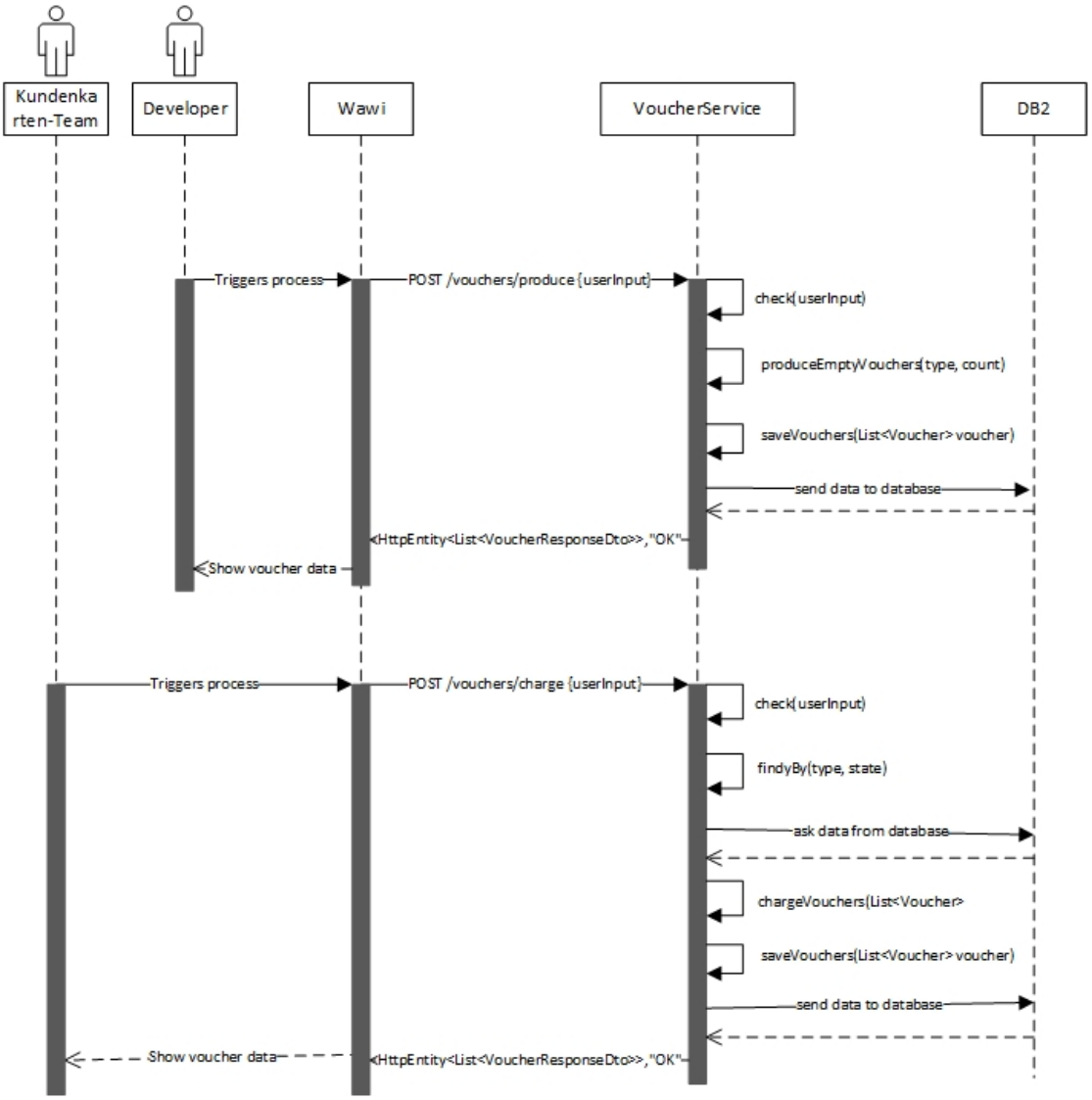

Abbildung 4.6: Sequenzdiagramm: Generieren und Aufladen von Gutscheinen

Die Abbildung 4.7 veranschaulicht die Abläufe für das Abfragen und Entladen eines Gutscheins. Der *VoucherService* püft den im JSON-Format gegebenen Request-Body und lädt über das *VoucherRepository* die Gutscheindaten. Als Nächstes wird eine HTTP-Response mit den Gutscheindaten und dem Code 200 ('OK') gesendet. Nach einer Bestätigung vom Kunden ruft das jeweilige System (Onlineshop oder Kasse) die REST-Schnittstelle zum Entladen des Gutscheins auf. Der Voucher-Service lädt zunächst mithilfe des Voucher-Repositorys den Gutschein aus der Datenbank, entlädt ihn und schickt

anschließend eine Rückmeldung. Mit dem *HTTP-Response* OK wird dem Nachbarsystem mitgeteilt, dass der Gutschein erfolgreich entladen wurde, damit dieses zuletzt dem Kunden bestätigt werden kann.

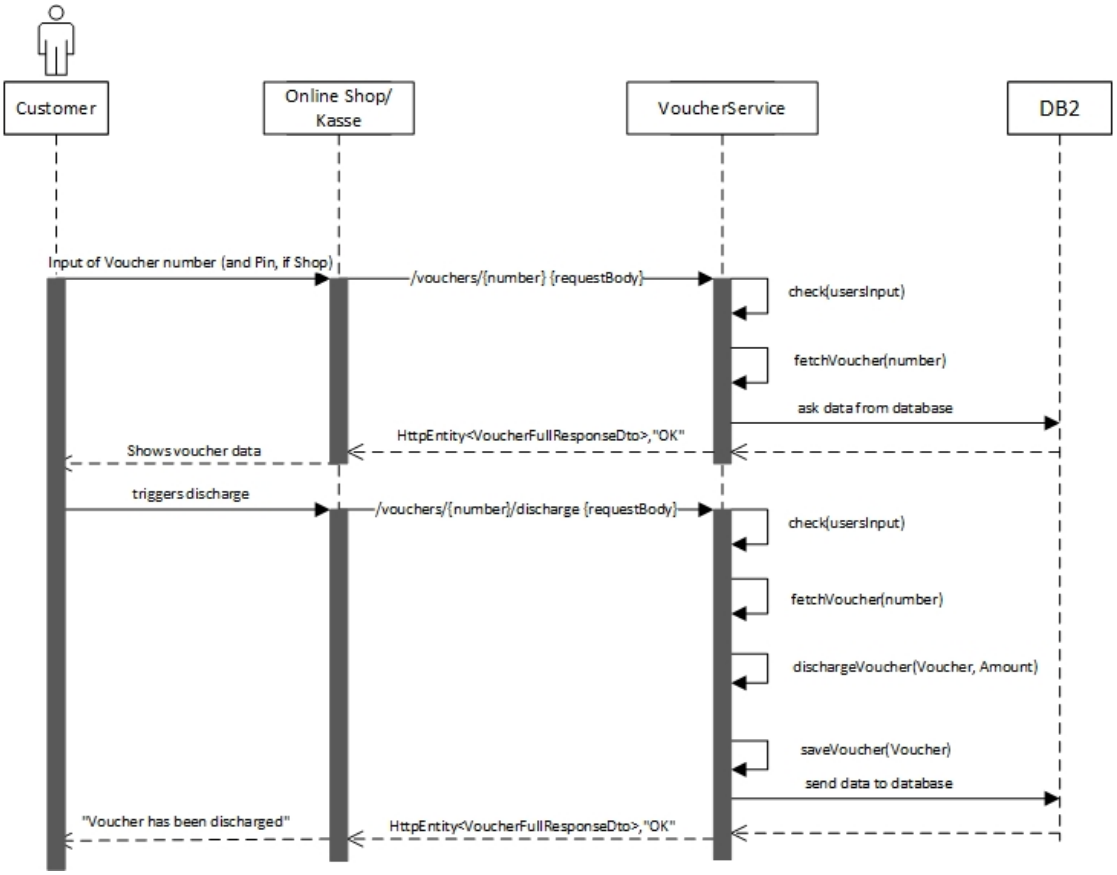

Abbildung 4.7: Sequenzdiagramm: Abfragen und Entladen eines Gutscheins

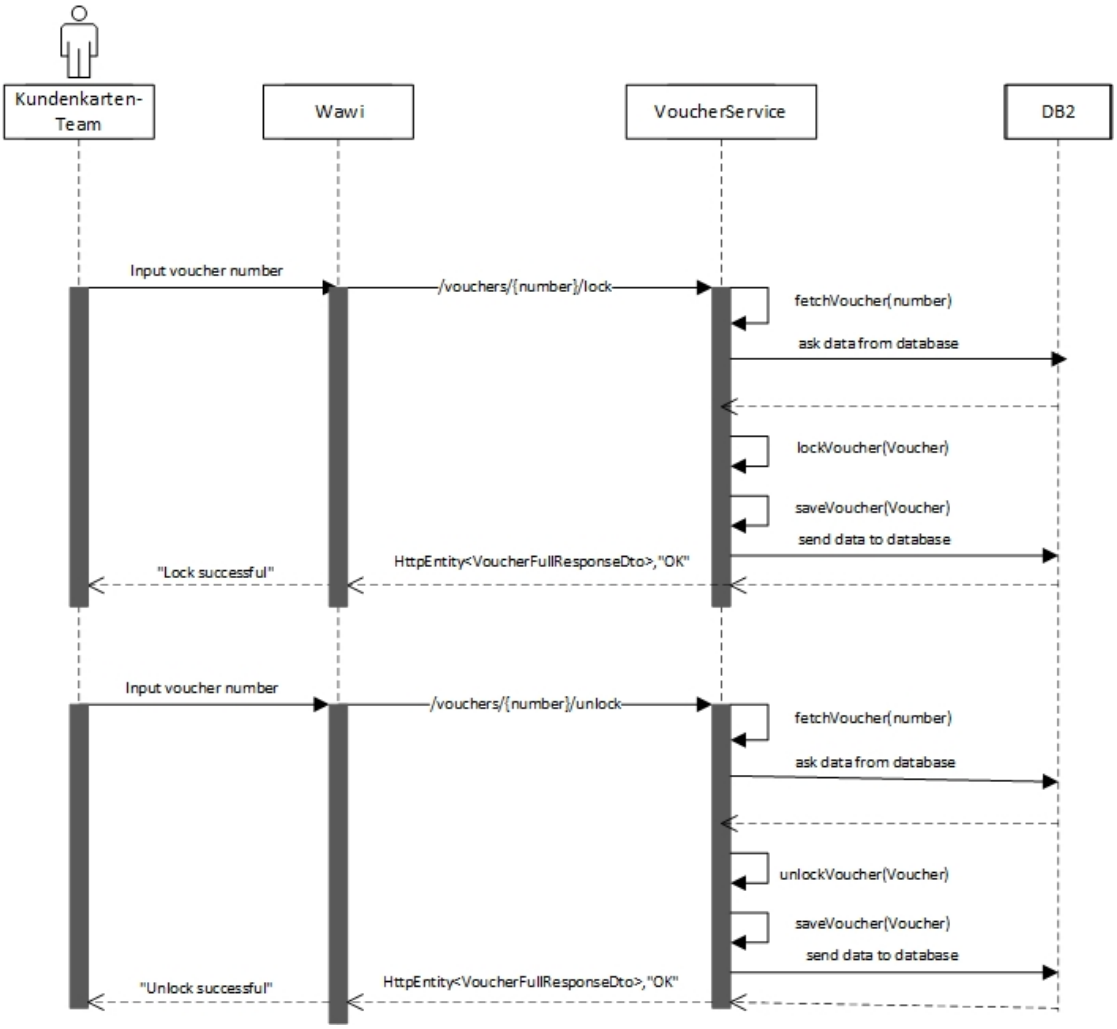

Abbildung 4.8: Sequenzdiagramm: Sperren und Entsperren eines Gutscheines

Das Sequenzdiagramm in der Abbildung 4.8 erklärt die Funktionsschritte für das Sperren und Entsperren eines bestimmen Gutscheins.
Das Kundekarten-Team übergibt die Nummer des zu sperrenden Gutscheins an das WAWI-System. Das WAWI-System ruft die Schnittstelle für das Sperren des Gutscheins auf, indem es die gegebene Gutscheinnummer als Pfad-Variable einfügt. Der VoucherService lädt als Nächstes die Gutschein-Entität aus der Datenbank und sperrt danach den Gutschein. Anschließend wird eine *HTTP-Response* geliefert, um zu signalisieren, dass die Sperrung erfolgreich war.
Die gleichen Wege gelten auch für das Entsperren des Gutscheins.

4.1.4 Verteilungssicht

Die Verteilungssicht stellt die Infrastruktur dar, auf der das System abläuft. Dabei werden drei Elemente beschrieben: Die Knoten, die Laufzeitelemente und die Kanäle.
Der Microservice läuft auf einem Docker-Container ab und wird mittels HTTP-Protokoll mit den Nachbarsystemen und der Datenbank verbunden. Für die Anwendung wird ein Tomcat als APP-Server zur Verfügung gestellt (vgl. Abbildung 4.9). Der Docker-Container *voucher-service* und der Server vom *Third* Party bilden die Knoten ab. Die Systeme tauschen mithilfe einer verschlüsselten Verbindung Informationen über die Transportschicht aus.

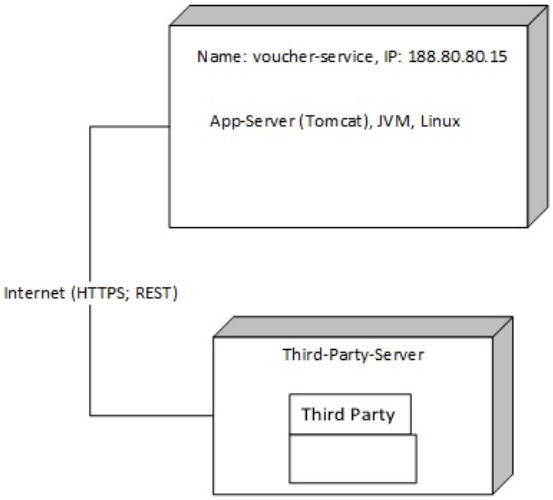

Abbildung 4.9: Verteilungssicht

4.2 Entwurf

Entwurfsentscheidungen sind der Schlüssel für eine langlebige Software. Ziel ist es, einen Weg zu finden, um Entwürfe für permanent wechselnde Anforderungen zu rüsten. Dies wird mittels *SOLID-Prinzipien* erreicht.[1].

4.2.1 SOLID-Prinzipien

In diesem Unterkapitel wird der Einsatzzweck der SOLID-Prinzipien im Allgemeinen beschrieben.

- *Single Responsibilty Principle* (SRP) basiert auf der Trennung von Verantwortlichkeiten. Die Microservice-Architektur unterstützt dieses Prinzip, da jeder Microservice nur für eine bestimmte Aufgabe verantwortlich sein soll. Zusätzlich werden innerhalb des Services Komponenten getrennt, die ebenfalls nur für eine bestimmte Aktivität verantwortlich sind (Siehe Abbildung 4.3). Der Einsatz von SRP beeinflusst die Kopplung und Kohäsion von Einzelteilen eines Systems untereinander. Bei geringer Kopplung (geringe Abhängigkeiten zwischen Komponenten) und hoher Kohäsion (enge Zusammengehörigkeit der Verantwortlichkeiten innerhalb einer Komponente) ist das System flexibler und gut erweiterbar.

- Das Offen-Geschlossen-Prinzip (engl: *Open-Close Principle*, OCP) besagt, dass Komponenten ohne Änderungen zukünftig erweiterbar sein sollen. Diesem Prinzip wird ebenfalls im Projekt gefolgt, z.B. bei der Implementierung des Zustandsautomaten.

- *Liskov-Substitutionsprinzip* (LSP) definiert strikte Regeln für den Einsatz von Unterklassen: sie sind nur für ihre Oberklassen einsetzbar. Dadurch werden Verträge, die z.B durch Vererbung entstehen, eingehalten.
  Diese Regeln wurden während der Implementierung bei dem Einsatz von Vererbung sichergestellt.

- *Interface Segregation Principle* (ISP) ermutigt die Verwendung von spezifischen feingranularen Schnittstellen. Das heißt, jede Komponente im System muss mehrere

[1] SOLID-Prinzipien sind bekannte Prinzipien, die als Grundlage für den objektorientierten Entwurf dienen. (Siehe Starke, 2017, S.71).

spezifische Schnittstellen enthalten. So bleiben einzelne Schnittstellen einfacher und sind weniger komplex (vgl. Starke, 2017, S.74).

- *Dependency Inversion Principle* (DIP) stellt fest, dass Abhängigkeiten von Systemfunktionen nicht auf konkrete Implementierungen zurückweisen, sondern nur auf Abstraktionen. Dadurch sind Implementierungskomponenten austauschbar, was mehr Flexibilität und Erweiterbarkeit bietet. Mittels des Entwurfsmusters *Dependency Injection* wird eine Komponente zur Laufzeit bereitgestellt, die für abstrakte Schnittstellen eine konkrete Instanz bestimmt. Spring implementiert dieses Prinzip, indem es die Verwaltung der Abhängigkeiten über einfache Konfigurationen anbietet (vgl. Starke, 2017, S.78).

4.2.2 Schnittstellenentwurf

Beim Schnittstellenentwurf wurden die spezifizierten Anforderungen beachtet, um REST-Schnittstellen mit *Swagger*[2] zu entwerfen.
Der Voucher-Service wird über HTTPS erreicht und bietet dem Consumer seine Dienste (REST-Schnittstellen) mittels synchroner Kommunikation an. Da die Schnittstellen über Basic Authentification gesichert sind, muss der Client für jede Anfrage seine Authentifizierungsdaten mit dem HTTP-Header *Base64-codiert* senden. Außerdem werden alle ausgetauschten Daten mit *SSL/TLS* verschlüsselt, um eine vertrauliche Kommunikation zu gewährleisten. Der Voucher-Service liefert für jede authentifizierte Anfrage eine Antwort vom Typ *application/hal+json* zurück (Siehe Abb. 4.10).

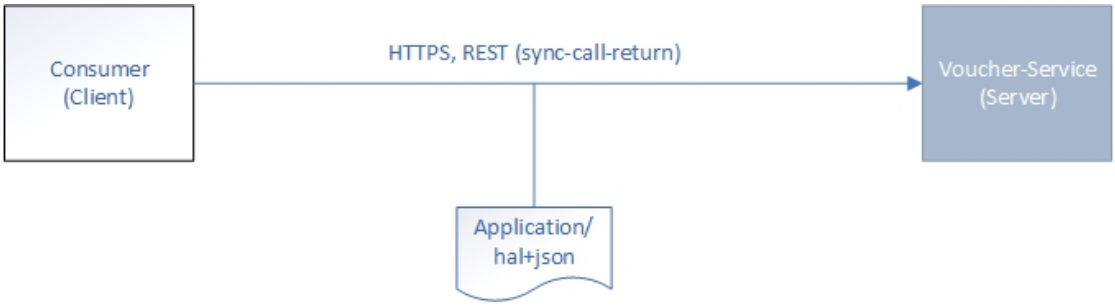

Abbildung 4.10: Aufbau der Kommunikation zwischen Client und Server

[2]Swagger ist ein Open-Source-Framework, welches auf den Entwurf von RESTful-Schnittstellen spezialisiert ist.

Nachdem die Schnittstellen des Voucher-Services implementiert waren, wurde die Swagger-Spezifikation mithilfe von *Springfox*[3] automatisch generiert (Siehe Abbildung 4.11 und 4.12). Mit dieser Automatisierung bleibt die Spezifikation immer auf dem neuesten Stand und muss nicht wiederholt manuell geschrieben werden.

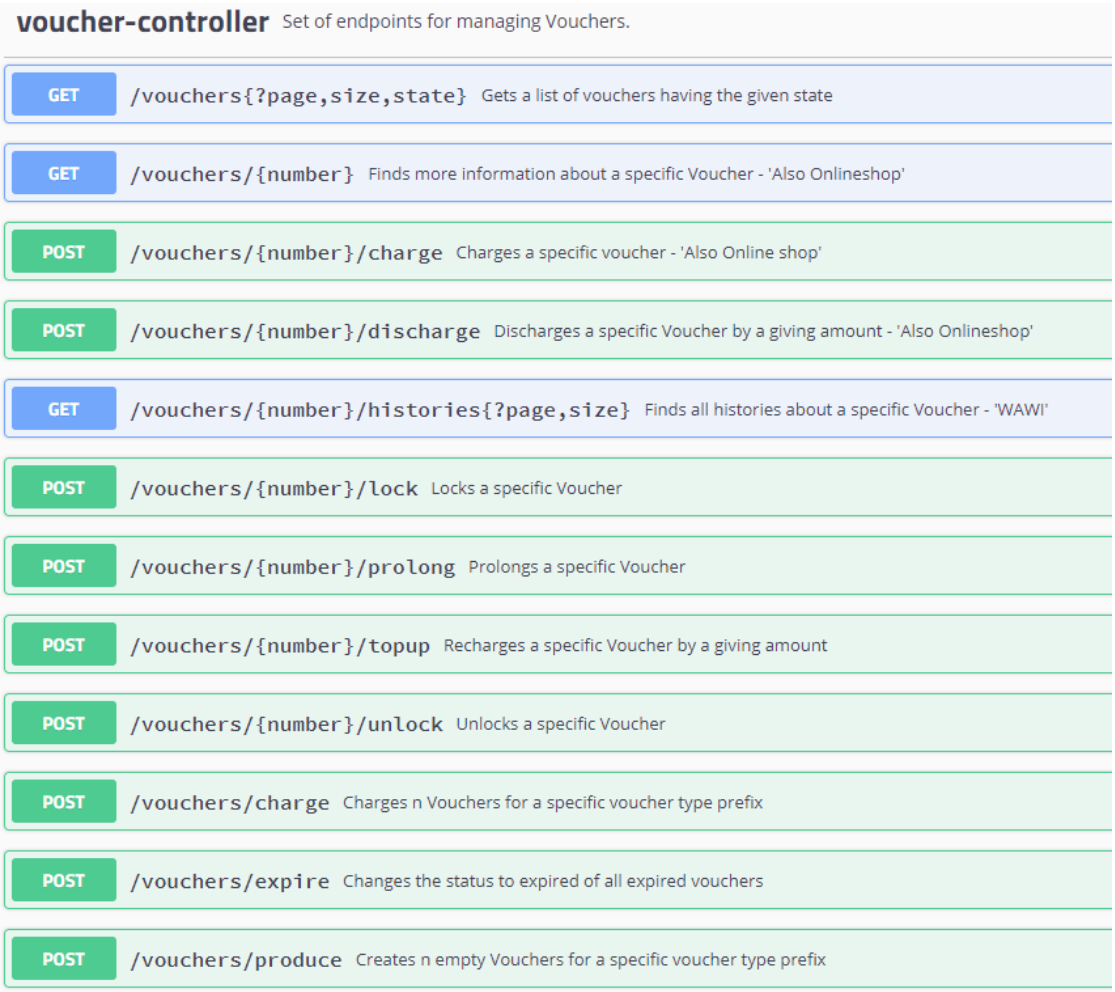

Abbildung 4.11: Swagger-Spezifikation für die Verwaltung von Gutscheinen

[3] Springfox ist eine Library, die integriert werden kann, um aus dem Quellcode eine automatisch Swagger-Spezifikation zu generieren.

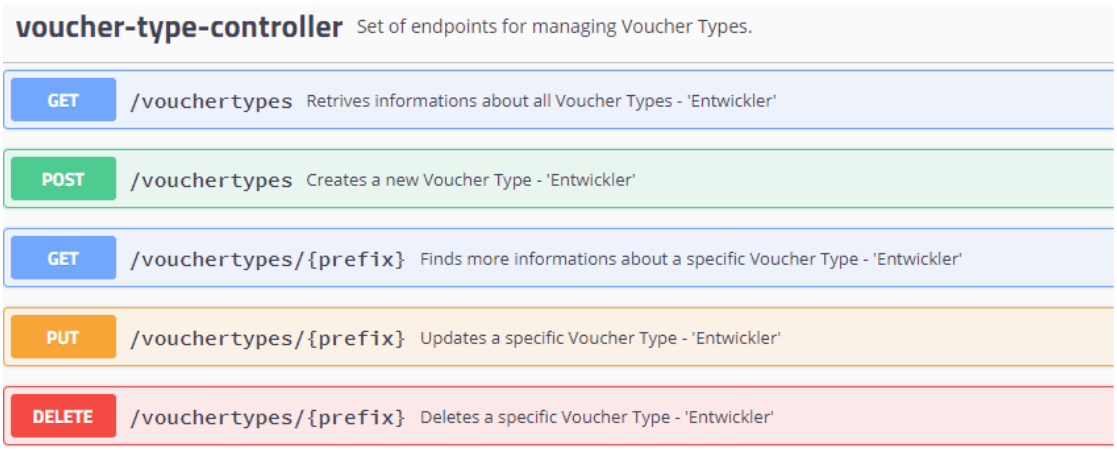

Abbildung 4.12: Swagger-Spezifikation für die Verwaltung von Gutscheinarten

5 Implementation

Der Gutschein Service ist mit dem Einsatz von Spring Boot und Spring Data implementiert worden. Er dient als eigenständiger Service und bietet Nachbarsystemen durch klar definierte REST-Schnittstellen Funktionalitäten zur Verwaltung von Gutscheinen nach außen an.

In diesem Kapitel werden wichtige Bestandteile des Projekts mithilfe von Ausschnitten aus dem Quellcode erklärt.

5.1 Geschäftslogik

5.1.1 Entitäten

Das entwickelte Datenmodell wurde mittels *Entitätsklassen* abgebildet, damit dessen Persistenz durch den Einsatz von Spring Data durchgeführt werden kann. Der folgende Code zeigt, wie eine *Voucher* Entität implementiert ist und mit Spring Data konfiguriert wurde:

```
1  @Entity
2  @Table(name = "voucher", schema = "vouchers")
3  public class Voucher implements Serializable {
4
5    @Id
6    @GeneratedValue(strategy = GenerationType.AUTO)
7    private Long id;
8
9    @Column(nullable = false, length = 13)
10   @NaturalId
11   private Long number;
12
13   @Column(nullable = false, length = 4)
```

```
14    private String pin;
15
16    @ManyToOne(optional = false)
17    private VoucherType voucherType;
18
19    @Column(nullable = false)
20    @Enumerated(EnumType.STRING)
21    private VoucherStateEnum state = VoucherStateEnum.INACTIVE;
22
23    @Column(nullable = false)
24    @Convert(converter = MoneyConverter.class)
25    private MoneyType amount = MoneyType.INIT_VALUE;
26
27    @Column(nullable = false)
28    private boolean rechargeable = true;
29
30    @Column(nullable = false)
31    private int invalidPinCount;
32
33    @Transient
34    @Value("${vouchers.max-pin-count}")
35    private final int MAXPINCOUNT;
36
37    @Column(nullable = false)
38    private LocalDate createdAt = LocalDate.now();
39
40    private LocalDate chargedAt;
41
42    private LocalDate validityTill;
43
44    private LocalDate amiabilityTill;
45
46    private LocalDate dischargedAt;
47    ...
48 }
```

Die *Voucher* Klasse ist mit `@Entity` annotiert worden, damit diese durch die JPA als Tabelle abgebildet werden kann. Mit der Annotation `@Table` wird der Tabellenname und das dazugehörige Schema bestimmt. Die *ID* der Tabelle ist mit `@Id` annotiert, damit sie von der JPA entsprechend abgebildet werden kann.
Mit der Annotation `@GeneratedValue` wird außerdem bestimmt, dass die *ID* autogeneriert werden soll. Alle weiteren Attribute der Klasse bildet Spring dann automatisch

als zugehörige Spalten der Tabelle ab. Mit der Annotation `@Column` werden besondere Eigenschaften der Attribute wie zum Beispiel ihre Länge oder die Notwendigkeit eines Pflichtfeldes hinzugefügt. Das Voucher Modell verfügt über eine `@ManyToOne` Annotation, um zu kennzeichnen, dass es zu der *VoucherType* Entität in einer *many-to-one-Beziehung* steht. Mit *optional=false* wird festgelegt, dass *Voucher* Entitäten immer einer vorhandenen *VoucherType* Entität zugeordnet sein sollen.
Die Konstante *MAXPINCOUNT* wird im Datenmodell als Konfiguration für alle Voucher-Instanzen definiert, sie muss nicht jedes Mal mitgespeichert werden. Aus diesem Grund ist sie als `@Transient` annotiert worden.
Die Annotation `@NaturalId` bildet das Attribut *number* als eine fachliche ID ab, da jede Voucher-Instanz eine eindeutige Nummer haben soll.
Mit der Annotation `@Enumerated(EnumType.String)` teilt man der JPA mit, dass der Status als Text gespeichert werden soll.
Damit das Attribut *amount* - welches aus zwei weiteren Attributen besteht - abgebildet werden kann, wurde ein Konverter mit `@Convert` hinzugefügt. Er beschreibt, wie das Persistieren des Attributs erfolgen soll. Die Tabelle hat infolgedessen statt einer *amount* Spalte zwei Spalten: die beiden Werte *value* und *currency*, aus denen sich das Attribut zusammensetzt.

5.1.2 Persistenz der Entitäten

Um Entitäten zu persistieren und darauf zugreifen zu können, wird für jede Entität ein eigenes Repository Interface deklariert. Dieses Interfaces kann durch zusätzlichen Funktionalitäten erweitert werden. Als Beispiel wird das *VoucherRepository* Interface erläutert:

```
1  interface VoucherRepository extends CrudRepository<Voucher, Long> {
2  ...
3  Voucher findByNumberAndPin(Long number, String pin);
4
5  @Query("select v "
6  + "from Voucher v where v.amiabilityTill <= current_time ")
7  List<Voucher> findExpiredVouchers();
8
9  Page<VoucherDto> findByState(VoucherStateEnum state, Pageable pageable);
10 ...
11 }
```

Das benutzte *CrudRepository* Interface bietet anspruchsvolle *CRUD-Funktionalitäten* per Default an. Diese können erweitert werden, um personalisierte Query-Methoden zu definieren. Als Methodennamen nutzt man das Präfix *findBy* gefolgt mit dem Namen des Attributes, nachdem gesucht wird. Zum Beispiel: *findByNumberAndPin(Long number, String pin)*. Die JPA wird beim Aufruf dieser Methode eine SQL-Anfrage ausführen und als Ergebnis das zugehörige *Voucher* Objekt, welches über die beiden Attribute *number* und *pin* verfügt, liefern.
Die Annotation `@Query` wird benutzt, um eine personalisierte native SQL-Anfrage zu definieren. Beim Aufruf der *findExpiredVouchers* Methode wird der deklarierte native SQL-Befehl benutzt, um nach abgelaufenen *Voucher* Objekten zu suchen.
Größere Datensätze werden dem Benutzer in kleinen Abschnitten geliefert, damit sie ihm schneller präsentiert werden können. Mit dem *Paging-Konstrukt* von Spring Data kann man das Ergebnis von Repository Methoden beliebig steuern indem ein zusätzlicher Parameter vom Type *Pageable* mit übergeben wird. Ein Beispiel für die Übergabe des Parameters ist *PageRequest.of(0, 2)* Dabei ist 0 die Nummer und 2 die Größe der angefragten Seite. Die Methode *findByState(stage, pageable)* würde in diesem Fall die ersten zwei *Voucher* Objekte, die den gegebenen Status *state* haben, zurück liefern.

5.2 REST Schnittstellen

Controller Klassen bieten REST-Schnittstellen an, um von außen den Zugriff auf interne Ressourcen zu ermöglichen. Auch hier wird nur ein kleiner Abschnitt vom Quellcode angezeigt und dabei die Umsetzung der spezifizierten Microservice-Schnittstellen zu zeigen.

```
1  @RestController
2  @RequestMapping("/vouchers")
3  @EnableHypermediaSupport(type = EnableHypermediaSupport.HypermediaType.HAL)
4  public class VoucherController {
5
6  @Autowired
7  private final VoucherService voucherService;
8
9  private final String APPLICATION_HAL_JSON = "application/hal+json";
10 private final HttpStatus OK = HttpStatus.OK;
11 private final HttpStatus BAD_REQUEST = HttpStatus.BAD_REQUEST;
12 private final HttpStatus FORBIDDEN = HttpStatus.FORBIDDEN;
13
```

```
14 public VoucherController(VoucherService voucherService) {
15 this.voucherService = voucherService;
16 }
17
18 @PostMapping(path = "/produce", produces = APPLICATION_HAL_JSON)
19 public HttpEntity<List<VoucherResponseDto>> produceVouchers(@RequestBody
       ProduceDto userInput) {
20
21 try {
22 checkProduceInput(userInput);
23 final List<VoucherResponseDto> vouchers = voucherService.produceVouchers(
       userInput.getPrefix(),
24 userInput.getQuantity());
25 return new ResponseEntity<>(vouchers, OK);
26 } catch (AbstractVoucherException e) {
27 return new ResponseEntity<>(Arrays.asList(new VoucherResponseDto(e.getCode
       (), e.getMessage())), BAD_REQUEST);
28 }
29 }
30 ...
31 }
```

Durch die Annotation @Autowired bringt Spring die Abhängigkeit anderer Objekte zur Laufzeit ein. Mit ihrer Hilfe wird bei der Implementierung der Einsatz vom *Dependency Injection* Entwurfsmuster ermöglicht.
Die Annotation @RestController teilt Spring mit, dass diese Controller Klasse bei der Bearbeitung eingehender Web-Anfragen berücksichtigt werden soll.
Die Annotation @RequestMapping(„/vouchers“) bietet Routing-Informationen an. So kann Spring jede HTTP-Anfrage über den */vouchers* Pfad zu einer der gegebenen Methoden der Klasse abbilden. Welche Methode das letzen Endes sein wird, ist in ihrer Deklaration definiert. Mit @PostMapping sind weitere Details zu der Methode *produceVouchers* bestimmt worden: Sie wird bei POST-Anfragen über den Pfad *path = /produce"* ausgeführt und liefert danach eine Response vom Type *application/hal+json* zurück. Durch die Annotation @RequestBody bildet Spring den *JSON-Body*, der über die Anfrage geschickt wird, im Methodenparameter *userInput* ab.
Spring führt bei POST-Anfragen über die Adresse */vouchers/produce* die Methode *produceVouchers* aus. Bei korrekter Eingabe werden die gewünschten Gutschein-Rohlinge erzeugt. Sowohl ein *JSON-Objekt*, welches die Details über die erzeugten Gutscheine enthält, als auch der *HTTP-Status OK* werden dem Sender als Antwort auf die Anfrage mitgeteilt. Kommt es zu Fehlern, wird dem Sender ein Response mit einem *Error-Code*

und eine Beschreibung des Fehlers mitgeteilt.

5.3 Einstellung der Applikation

In der *application.properties* Datei der Applikation sind unter *src/main/resources* personalisierte Einstellungen für den Datenbank- und Server-Zugriff deklariert worden. Die Datei ist folgendermaßen aufgebaut:

```
spring.datasource.url=<URL-DER-DATENBANK>
spring.datasource.username=<USERNAME>
spring.datasource.password=<PASSWORD>
spring.datasource.driver-class-name=com.ibm.db2.jcc.DB2Driver

spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update

server.port=8443
server.ssl.key-store=classpath:ssl/keystore.p12
server.ssl.key-password=<PASSWORD>
server.ssl.key-store-password=${server.ssl.key-password}
```

Die Attribute von *spring.datasource* definieren die Zugriffsdaten zur Datenbank. Beim Setzten der *spring.jpa* Attribute wird zum Beispiel die TEST-Datenbank so eingestellt, dass bei jedem Neustart Änderungen an Schema und Tabellen automatisch ausgeführt werden sollen.
Mit *server.port* wird die Portnummer, auf dem der Server erreichbar ist, eingesetzt. Mit den Attributen von *server.ssl* werden Informationen über das *SSL-Zertifikat* eingegeben, damit der Service über HTTPS serviert werden kann.
Mit der Main-Klasse *VoucherApplication.java* wird die Microservice-Anwendung in einem eingebetteten Tomcat Servlet-Container bereitgestellt und ausgeführt. Der zuständige Code sieht wie folgt aus:

```
@SpringBootApplication
public class VoucherApplication {
  public static void main(String[] args) {
    SpringApplication.run(VoucherApplication.class, args);
  }
}
```

Die @SpringBootApplication Annotation startet eine automatische Konfiguration und sucht nach allen vorhandenen Komponenten, die sich im selben *Package* wie die Applikation befinden. Zum Schluss wird durch den Aufruf der *main()* Methode der gesamte Service gestartet. Der Zugriff auf das Gutscheinsystem erfolgt dann über *https://<IP-Adresse>:8443/vouchers*

5.4 Docker

Damit Docker ein Image aufbaut, wird eine Datei mit dem Namen *Dockerfile* benötigt. Sie beinhaltet alle notwendigen Instruktionen, um ein gegebenes Image automatisch aufzubauen. Docker kann dann dieses Image ausführen und somit einen Container generieren.

Ein Docker-Image besteht aus mehreren Schichten, von denen jede aus einer Anweisung von der Docker-Datei stammt. Die *Dockerfile* Datei wurde folgendermaßen geschrieben:

```
1 FROM openjdk:8-jdk-alpine
2 VOLUME /tmp
3 ARG DEPENDENCY=target/dependency
4 COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
5 COPY ${DEPENDENCY}/META-INF /app/META-INF
6 COPY ${DEPENDENCY}/BOOT-INF/classes /app
7 ENTRYPOINT ["java","-cp","app:app/lib/*","-Dserver.contextPath=/v2/api-docs
      ",
8 "de.puc.voucher.applicationcore.VoucherApplication"]
```

Mit FROM wird eine Schicht aus dem *openjdk:8-jdk-alpine* Docker-Image erstellt.
Mit VOLUME ist ein Bereich ausgesetzt worden, der auf den Ordner */tmp* zeigt, weil sich dort das Arbeitsverzeichniss von Tomcat befindet.
Spring schreibt die *JAR* Datei der Applikation in den Ordner *target/dependency/lib* und die Applikation-Klassen in den Ordner *target/dependency/classes*. Diese Ordner müssen für Docker bekannt sein. Mit ARG wird ein neuer Parameter, welcher auf den Hauptordner der bereits erwähnten Ressourcen zeigt, erzeugt
Mit ENTRYPOINT wird der Befehl, der beim Start des Containers die *JAR* Datei ausführen soll, bestimmt.
Nachdem das Docker-Image gebaut ist, kann es mit dem Befehl *docker run -p 8443:8443 -t de.puc/voucher-service* in einem Container gestartet werden. Der Service ist dann durch die IP-Adresse des Containers und die Portnummer 8443 erreichbar.

6 Evaluierung

Um möglichst früh Fehler in der Software zu entdecken und dadurch die Qualität zu sichern, werden unterschiedliche Testfälle definiert.
Automatisierte Tests können bei jeder Codeveränderung erneut ausgeführt werden, um das Systemverhalten kontinuierlich zu prüfen. Diese werden vor dem Build-Prozess mit *Gradle* [1] automatisch ausgeführt.

6.1 Unit-Tests

Unit-Tests werden angewendet, um die kleinste überprüfbare Einheit vom Rest zu isolieren und einzeln zu prüfen. Hierfür wurde das *JUnit* Test-Framework für die Testautomatisierung eingesetzt.
Der folgende Code zeigt, wie der Unit-Test eines Voucher Objekts definiert ist:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class VoucherUnitTest {
  private Voucher voucher;
@Before
public void setUp() {
  voucher(1111567891234L, "1234")
}
@Test
public void whenSetVoucherType_thenItShouldBeFound() {
  VoucherType voucherType = new VoucherType();
  voucher.setVoucherType(voucherType);
  assertThat(voucher.getVoucherType()).isEqualTo(voucherType);
}
}
```

[1] Gradle ist ein auf Java basierendes Build-Management-Automatisierungs-Tool

Mit der Annotation @SpringBootTest werden die Features von Spring Boot in die Testklasse eingebunden, sodass der *ApplicationContext* [2] den Testmethoden übergeben wird. Damit dies funktioniert, muss für die JUnit 4 Version die Annotation @RunWith(SpringRunner.class) vorhanden sein, sonst wird der Kontext der Applikation nicht geladen. Mittels der JUnit Funktionen *assertThat()* und *isEqualTo()* wird sichergestellt, dass zwei Objekte gleich sind und damit auch die korrekte Funktionsweise der *setVoucherType()* Methode.

6.2 Integrationstests

Nachdem alle Einheiten einzeln erfolgreich getestet wurden, kommt jetzt der Einsatz von Integrationstests, um das Zusammenspiel abhängiger Einzelteile zu prüfen. Der folgende Code zeigt, wie das Persistieren eines Gutschein-Objekts mittels *H2*[3] Datenbank getestet wurde:

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class VoucherRepositoryIntegrationTests {
4
5  @Autowired
6  VoucherRepository voucherRepository;
7
8  @Autowired
9  VoucherTypeRepository voucherTypeRepository;
10
11 @Autowired
12 VoucherUseCase voucherUseCase;
13
14 Long voucherNumber = 1111567891234L;
15 String voucherPin = "1234";
16 String PREFIX = "1111";
17 VoucherType voucherType;
18
19 @Before
20 public void setUp() {
```

[2] Eine Spring Boot Applikation ist ein Spring ApplicationContext, wo viele Eigenschaften von Spring installiert sind.

[3] H2 steht für H2 Database Engine und ist eine eingebettete relationale Datenbank für Java-Anwendungen.

```
21    voucherType = voucherTypeRepository.findByPrefix(PREFIX);
22  }
23
24  @Test
25  public void saveCorrectly() {
26  Voucher voucher = voucherUseCase.produceEmptyVoucher(voucherType,
        voucherType.getSerialNumber());
27  voucherRepository.save(voucher);
28  Voucher result = voucherRepository.findByNumberAndPin(voucher.getNumber(),
        voucher.getPin());
29  assertThat(result).isNotNull();
30  assertThat(result).isEqualToComparingFieldByField(voucher);
31  }
32  }
```

In der *import.sql* Datei, wurde mittels SQL-Befehl ein neues *VoucherType* Objekt hinterlegt, welches in diesem Test für die Erstellung neuer Gutschein-Rohlinge verwendet wurde.
Nachdem ein *VoucherType* Objekt geladen wurde, wird es gemeinsam mit einem neuen Voucher Objekt gespeichert und danach anhand seiner Nummer und Pin aus der Datenbank geladen, um zu überprüfen, ob neue Voucher Objekte erfolgreich gespeichert und geladen werden.

6.3 REST-Schnittstellentests

Da mit `@SpringBootTest` standardmäßig kein Server gestartet wird, wurde für die Schnittstellentests das Spring Test Framework *MockMvc* eingesetzt, um mit dessen Hilfe die Serverumgebung für die Webendpunkte zu mocken. Das folgende Beispiel zeigt, wie der Controller Test aufgebaut ist:

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class VoucherControllerTest {
4    @Autowired
5    WebApplicationContext context;
6    MockMvc mvc;
7
8    String PRODUCE_PATH = "/vouchers/produce";
9
10   Integer QUANTITY = 10;
```

```
11   String PREFIX = "1111";
12
13   @Value("${users.wawi.username}")
14   private String wawiUsername;
15   @Value("${users.wawi.password}")
16   private String wawiPassword;
17 }
18
19 @Before
20 public void setup() {
21   mvc = MockMvcBuilders
22   .webAppContextSetup(context)
23   .apply(springSecurity())
24   .build();
25 }
26 @Test
27 public void produceCorrectly() throws Exception {
28   ProduceDto userInput = new ProduceDto(PREFIX, QUANTITY);
29   mvc.perform(post(PRODUCE_PATH)
30   .with(httpBasic(wawiUsername, wawiPassword))
31   .contentType(CONTENT_TYPE)
32   .content(getJson(userInput)))
33   .andExpect(status().isOk())
34   .andExpect(content().contentType(CONTENT_TYPE))
35   .andExpect(jsonPath("$").isArray())
36   .andExpect(jsonPath("$.*").isNotEmpty())
37   .andExpect(jsonPath("$.*", hasSize(QUANTITY)));
38 }
39 ...
40 }
```

Das *WebApplicationContext* Objekt wird benötigt, um alle Beans und Controller Klassen in einen Kontext zu laden. Anschließend kann dieser Kontext im MockMvc Objekt hinzugefügt werden, um ihn dem Test zur Verfügung zu stellen.
Da die Schnittstellen durch *Spring Security* mittels Basic-Authentifikation geschützt sind, muss der Aufruf vor dem Test authentifiziert werden.
Der Test ruft die Schnittstelle */vouchers/produce* mit POST auf und übermittelt dabei einen JSON-Body mit der gewünschten Menge neuer Gutschein-Rohlinge. Zum Schluss wird erwartet, dass der Aufruf erfolgreich war und dass die Menge der erzeugten Gutscheine der gegebenen Menge entspricht.

6.4 Performanz-Tests

Da es vorgesehen war, eventuell eine große Menge von Gutscheinen zu erzeugen und aufladen zu können, wurden Performanz-Tests geschrieben, um die Laufzeit bei aufwändigen Transaktionen zu messen. Sie unterscheiden sich von den Integrationstests durch die hohe Menge von Gutscheinen. Es wurde ein Test-Fall definiert der *10.000* Gutscheine erzeugt und diese danach alle mit einer weiteren Anfrage auflädt. Die dabei benötigte Laufzeit beträgt *ca. 9 Sekunden*. Dieses Ergebnis wurde allerdings erst erreicht, nachdem die Batchverarbeitung mithilfe von *Hibernate* eingeführt wurde.

6.5 Überblick über die Umsetzung

Alle entstandenen User-Stories aus den Anforderung wurden vollständig umgesetzt. Dieses war nach dem vollständigen Testen der implementierten Funktionalitäten nachvollziehbar. Das Nutzen des Systems über REST-Schnittstellen ist ebenfalls umgesetzt worden, sowie die Bereitstellung der Anwendung über Docker.
Im Kapitel 7 Fazit werden das Design und die benutzten Technologien bewertet, sowie der Ausblick für die möglichen Schritte, die technisch noch umgesetzt werden müssen, um mehr von der Microservice-Architektur profitieren zu können.

7 Fazit

Der Microservice für die Verwaltung von P&C Gutscheinen konnte innerhalb der geplanten Zeit konzipiert und entwickelt werden. Die Integration in einer Produktivumgebung war nicht Teil der Bachelorarbeit, deshalb wurde der Betrieb der Anwendung im Docker-Container der eigenen Maschine simuliert und zuletzt mithilfe von Postman[1] das Verhalten der Anwendung durch verschiedene manuelle Aufrufe geprüft.
Im kommenden Unterkapitel folgt die Bewertung der bei der Arbeit eingesetzten Techniken.

7.1 Bewertung

Microservice-Architekturen zeichnen sich durch zwei Hauptcharakteristiken aus: optimaler Komplexitätsgrad und Isolation (vgl. Takai, 2017, S.18 f.). Beim Entwurf wurde das Single-Reponsibilty-Prinzip eingehalten, so dass jede Klasse nur für eine einzige Aufgabe zuständig ist. Des Weiteren hat der Voucher-Service nur die Aufgabe, Gutscheine zu verwalten. Dies verringert die Komplexität, da der Service infolgedessen leichter zu formen ist. Dabei wird die hohe Änderbarkeit als Qualitätsmerkmal von Geschäftssystemen sichtbar.
Der Gutschein-Microservice ist fachlich unabhängig vom Rest der P&C-Software. Er wird in einem eigenen Respository versioniert und lässt sich von einem einzigen Team optimal entwickeln und betreiben. Außerdem wird er in einem einzigen Prozess isoliert ausgeführt und kann deswegen ohne Einfluss auf andere Anwendungen jederzeit beliebig angepasst, verteilt und skaliert werden. Durch die Einheitlichkeit der Schnittstellen kann der Service von den Nachbarsystemen einfach genutzt werden, so dass die Details dabei versteckt bleiben.

[1] Postman ist eine grafische Benutzerschnittstelle, mit deren Hilfe HTTP-basierende Schnittstellen aufgerufen werden können.

Mit dem Einsatz von Spring Boot ist es gut gelungen, ein Anwendungssystem auf Basis einer Microservice-Architektur zu konstituieren. Spring Boot erleichtert zudem das Nutzen weiterer Spring-Framework-Funktionalitäten, mit deren Hilfe eine eigenständig lauffähige Anwendung erschaffen wurde, die keine externe Laufzeitabhängigkeit benötigt (vgl. Simons, 2018, S.4). Darüber hinaus war die Integrierung von Persistenz (JPA) und Webtechnologien (REST) sowie das getrennte Testen dieser Technologien realisierbar. Das im Kapitel 4.2.1 erklärte Dependency Injection Entwurfsmuster konnte mit dem Einsatz von Spring ohne großen Aufwand eingesetzt werden. Alle diese Eigenschaften führten zu einer hochwertigen Software. Mit dem Softwaredesign-Paradigma Konvention vor Konfiguration, welches von Spring bereitgestellt wird, lag der Fokus stets auf der eigentlichen Aufgabe, die funktionalen Anforderungen umzusetzen.
Durch den Einsatz der Container-Technologie mittels Docker (siehe Unterkapitel 5.4) ist sichergestellt worden, dass die Software ohne einen weiteren Aufwand in jeder Docker-Umgebung laufen kann. Das System ist dadurch portierbarer und funktioniert in jedem Server mit Docker einwandfrei.

7.2 Ausblick

Der Hauptzweck von einem Gutscheinsystem war die Reduzierung der Verwaltungskosten, indem ein internes System gebaut wird. Dessen Erfolg kann sich aber erst zeigen, wenn das System im normalen Betrieb eingesetzt und bewertet wird. Außerdem bleibt noch offen, ob sich Schwierigkeiten im Zusammenspiel mit externen Systemen ergeben.
Um den Mehrwert des Services zu erhöhen, könnten automatische Prozesse ausgeführt werden, wie z.B. Gutscheine automatisch bereitzustellen und zu entwerten, die im Rahmen dieser Arbeit nicht vereinbart waren. Weiterhin wäre interessant zu beobachten, wie sich der Service in einem laufenden Betrieb verhält und wie dieser auf Erweiterungen und ihrer Veröffentlichung wirkt. Momentan werden die Daten in einer bestehenden Datenbank gespeichert. Eine Alternative dazu wäre, eine neue Datenbank zu implementierten, um die Kompatibilität anderer Datenbank-Technologien zu untersuchen. Dies könnte jedoch mit zusätzlichem Aufwand verbunden sein.
Die Nutzung von Cloud-Technologien für den Betrieb von Systemen bringt Kostenvorteile mit und bildet gute Grundlagen für Microservice-Architekturen. Das Betreiben von Container-Images wird mittlerweile schon von vielen Cloud-Anbietern[2] unterstützt.

[2] etwa: Heroku, Amazon AWS, Microsoft Azure, IBM Bluemix, Google Cloud etc.

Literaturverzeichnis

[Cohn 2004] Cohn, M.: *User Stories Applied.* ADDISON-WESLEY, 2004

[Conway 1968] Conway, M.: How Do Committees Invent? (1968). – URL http://www.melconway.com/research/committees.html

[Docker 2018] Docker: Docker Doc. (2018). – URL https://docs.docker.com/

[Ebert 2008] Ebert, C.: *Systematisches Requirements Engineering und Management.* dpunkt.verlag, 2008

[Fowler 2010] Fowler, M.: Richardson Maturity Model. (2010). – URL https://martinfowler.com/articles/richardsonMaturityModel.html

[Fowler 2014] Fowler, M.: Microservices. (2014). – URL https://martinfowler.com/articles/microservices.html

[Freund und Rücker 2014] Freund, J. ; Rücker, B.: *Praxishandbuch BPMN 2.0.* HANSER, 2014. – ISBN 978-3446442559

[Mouat 2016] Mouat, A.: *Docker Software entwickeln und deployen mit Containern.* dpunkt.verlag, 2016. – ISBN 978-3864903847

[RFC 2019] RFC: Hypertext Transfer Protocol – HTTP/1.1 (RFC2616). (2019)

[Rupp und Pohl 2015] Rupp, C. ; Pohl, K.: *Basiswissen Requirements Engineering.* dpunkt.verlag, 2015

[Simons 2018] Simons, M.: *Spring Boot 2: Moderne Softwareentwicklung mit Spring 5.* dpunkt.verlag, 2018. – ISBN 978-3864905254

[Spring-Boot 2019] Spring-Boot: Spring Boot Reference Guide. (2019). – URL https://docs.spring.io/spring-boot/docs/2.1.2.RELEASE/reference/htmlsingle/

[Starke 2017] Starke, G.: *Software-Architekturen.* HANSER, 2017

[Takai 2017] Takai, D.: *Architektur für Websysteme*. HANSER, 2017

[Wikpedia 2018] Wikpedia: Representational State Transfer. (2018). – URL https://de.wikipedia.org/wiki/Representational_State_Transfer

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Konzeption und Entwicklung eines Gutscheinsystems für ein mittelständisches Einzelhandelsunternehmen unter Verwendung von Microservice-Architekturen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original