Dokumentation

GPA-WS1617-TEAM2

Exported on Okt 30, 2017

Table of Contents

1.1 Aufgabenstellung 5
1.2 Qualitätsziele 5
1.3 Stakeholder 6
2.1 Technische Randbedingungen 8
2.2 Organisatorische Randbedingungen 8
2.3 Konventionen 9
2.3.1 Definition of Done 9
2.3.2 Coding Guidelines 9
2.3.3 Git 10
3.1 Fachlicher Kontext 11
3.2 Technischer Kontext 12
4.1 Austauschbarkeit 15
4.2 Eingenständiges UI 15
4.3 Entity-Control-Boundary Pattern 15
5.1 Ebene 1 16
5.1.1 Übersicht 17
5.1.2 Beschreibung der Blackboxen 18
5.2 Ebene 2 18
5.2.1 Recommendation Service 19
5.2.2 Rating Service 19
5.3 Ebene 3 20
6.1 Laufzeitszenario 1: Meine Bücher 21
6.2 Laufzeitszenario 2: Empfehlungen 22
6.3 Laufzeitszenario 3: Bewertung 23
6.4 Laufzeitszenario 4: Autorisierung 24
6.5 Laufzeitszenario 5: CI/CD-Pipeline 24
8.1 Fachliches Datenmodell 26
8.2 Typische Muster, Strukturen und Abläufe 27
8.2.1 Typische Muster und Strukturen 27
8.2.2 Abläufe 27
8.3 Persistenz 29
8.3.1 Customers 29
8.3.2 BookRatings 29
8.3.3 EJB 30
8.3.4 Flyway 31
8.4 Benutzungsoberfläche 31
8.5 Ablaufsteuerung 32
8.6 Transaktionsbehandlung 33
8.7 Sessionbehandlung 33
8.8 Validierung, Ausnahme- und Fehlerbehandlung 34
8.8.1 Validierung 34
8.8.2 Ausnahme- und Fehlerbehandlung 34
8.9 Konfigurierbarkeit 35
8.9.1 Konfiguration des Projekts mittels Maven 35
8.9.2 Konfiguration von Jenkins 40
8.9.3 Konfiguration der JPA 42
8.9.4 Einsatz eines Stubs 43
8.10 Migration 44
8.11 Testbarkeit 44
8.11.1 CI/CD Pipeline 44
8.11.2 Unit Tests 45
8.11.3 Integration Tests 46
8.12 Buildmanagement 46
8.12.1 Struktur 46
8.13 CI/CD-Pipeline 48
8.13.1 Entwicklung 48
8.13.2 Integration 49
8.13.3 Deployment 49
8.14 API Dokumentation 49
9.1 Empfehlungssystem 52

9.1.1 Algorithmus ... 52
9.1.2 Alternativen ... 52
9.1.3 Probleme ... 53
9.2 Teilung der Microservices ... 53
9.2.1 Projektstruktur ... 53
9.3 Dateninkonsistenz: Recommendation und Book Microservice ... 54
11.1 Sprint Planung ... 59
11.2 Sprint ... 59
11.3 Daily Scrum ... 60
11.4 Scrum of Scrums ... 60
11.5 Sprint Review ... 60
11.6 Retrospektive ... 61
11.7 Abschluss ... 61
12.1 Microservices ... 62
12.1.1 Ungelöste Herausforderungen ... 62
12.1.2 Probleme ... 63
12.1.3 Gesammelte Erfahrungen ... 64
12.2 Java EE7 vs. OASP ... 64
12.2.1 Eindrücke Java EE 7 ... 64
12.2.2 Vergleich zu OASP ... 65
13.1 B ... 66
13.1.1 Book Created Mär 23, 2017 (13:12) ... 66
13.2 C ... 66
13.2.1 Continuous Integration Created Feb 12, 2017 (20:45) ... 66
13.2.2 Customer Created Mär 23, 2017 (13:04) ... 67
13.2.3 Context and Dependency Injection Created Mär 24, 2017 (14:27) ... 67
13.2.4 Continuous Delivery Created Mär 24, 2017 (14:32) ... 67
13.3 D ... 67
13.3.1 Definition of Done Created Okt 10, 2016 (22:23) ... 67
13.4 E ... 67
13.4.1 Entity-Control-Boundary Created Mär 23, 2017 (13:20) ... 67
13.5 R ... 68
13.5.1 Rating Created Mär 23, 2017 (13:15) ... 68
14.1 Sprint Planning ... 69
14.2 User Story ... 69
14.3 Task ... 69
15.1 Style Conventions ... 70
15.1.1 Allgemein ... 70
15.1.2 Leerzeichen ... 70
15.1.3 Klammern ... 71
15.1.4 Einrückung ... 72
15.1.5 Methodenparameter ... 72
15.1.6 Mehrzeilige Kommentare ... 72
15.1.7 this ... 73
15.1.8 Benennungen ... 73
15.2 Guidelines ... 75
15.2.1 "Keep the code simple" ... 75
15.2.2 Kommentare ... 75
15.2.3 Tests ... 75
15.3 Versionskontrolle ... 75

1 Einführung und Ziele

1.1 Aufgabenstellung

Im Rahmen des Projektes sollte die Gruppe den Microservice-Architekturansatz anhand der Implementierung mithilfe von Java EE7 kennen lernen und erproben.

Als Domäne wurde ein Online-Buchhandel gewählt, welcher in Form eines Prototypen umgesetzt werden sollte. Die zu entwickelten Funktionalitäten wurden dabei auf zwei Teams verteilt, die ebenfalls mit unterschiedlichen Technologien arbeiten sollten. Das erste Team (Team 1), welches OASP verwenden sollte, implementierte die Verwaltung eines Bücherkatalogs, eine Such-Funktion, um diesen zu durchsuchen und eine Import-Funktion, um Bücher, die nicht bereits im Katalog vorhanden sind, durch Amazon importieren zu können.

Das zweite Team (Team 2) realisierte die Funktionalitäten der Kundenverwaltung, des Bewertungs- und Empfehlungssystems unter Einsatz von Java EE7. In der nachfolgenden Tabelle werden diese Funktionalitäten kurz beschrieben.

Funktionalität	Beschreibung
Kundenverwaltung	Ein einfaches Kundenverzeichnis, welches initial mit Kundendaten befüllt wird. Es können keine Kunden hinzugefügt, geändert oder gelöscht, sondern lediglich abgerufen werden.
Bewertungssystem	In der von Team 1 entwickelten Buchsuche besteht für Kunden die Möglichkeit, Bücher zu bewerten. Diese Bewertungen werden vom Bewertungssystem verarbeitet. In der "Meine Bücher"-Ansicht werden die bewerteten Bücher eines Kunden dargestellt und können von diesem gelöscht oder angepasst werden.
Empfehlungssystem	Anhand der von einem Kunden bewerteten Bücher werden diesem bis zu drei Empfehlungen in der "Meine Bücher"-Ansicht angezeigt, sofern vorhanden. Diese werden durch einen Vergleich mit allen vorhandenen Kunden ermittelt.

Da der (imaginäre) Kunde die Weiterentwicklung des Systems nach der Entwicklung des Prototypen übernehmen wollte, lag der Schwerpunkt der Teams neben der eigentlichen Erstellung der Microservice-Architektur auf dem Aufbau einer stabilen Continuous Integration und Delivery (CI & CD) Pipeline.

Zusätzlich erhoffte sich der Projektpartner Capgemini, vertreten durch Axel Burghof, Erkenntnisse über die Nutzbarkeit der beiden verwendeten Technologien in der Praxis, unter Verfolgung der DevOps-Methodologie, sammeln zu können.

Neben den genannten technischen Aspekten, waren die Betreuer des Projektes, vor allem aufseiten der TH Köln, ebenfalls an der Evaluation der Eignung des im Hochschulumfeld durchgeführten Scrum Vorgehensmodells interessiert.

1.2 Qualitätsziele

Auch wenn keine Qualitätsziele von den Stakeholdern vorgegeben wurden, bzw. diese nicht vom Team überprüft wurden, lassen sich aufgrund der Aufgabenstellung und dem akademischen Hintergrund der Anwendung folgende Qualitätsziele als besonders wichtig herausstellen:

- Austauschbarkeit & Testbarkeit

- Verfügbarkeit

Diese gehen aus der Aufgabe hervor, eine DevOps Pipeline zu erstellen, welche CI und CD ermöglicht.
Ein wichtiger Bestandteil stellen dabei die automatisierten Tests dar, die es trotz automatisierter Auslieferung ermöglichen sollen, dass die Funktionalität der Anwendung nach wie vor gewährleistet ist.

Um eine grundsätzliche Testbarkeit der Anwendung zu erreichen, wurde der testgetriebene Entwicklungsansatz (Test Driven Development - TDD) verfolgt.

Die Testbarkeit sowie DevOps impliziert, dass die Austauschbarkeit von Komponenten im Design berücksichtigt werden muss. Dies spielt eine wichtige Rolle, um bspw. während des Deployments der Anwendung die genutzte Datenbank austauschen zu können. Ebenso wichtig ist die Austauschbarkeit für das Durchlaufen der Unit Tests, um externe Ressourcen, wie fremde Webdienste, deren Performance und Verfügbarkeit nicht garantiert werden kann, gegen verlässliche Test-Implementation, wie Mocks oder Stubs, austauschen zu können.

Ebenfalls durch DevOps impliziert ist die Verfügbarkeit des Systems. Durch den automatisierten Auslieferungsprozess soll versucht werden zu gewährleisten, dass das System in der aktuellsten Version vorliegt und Verbesserungen und Fehlerbehebungen schnell ins Produktivsystem aufgenommen werden.
Dadurch kann beispielsweise die Zeit verringert werden in der das System sonst aufgrund von Wartungsarbeiten oder einem falschen Deployment nicht verfügbar ist.

1.3 Stakeholder

Funktion (Rolle)	Dozent
Relevanz	Hoch
Ziele/Interessen	• Überprüfung der Durchführbarkeit von Scrum im Hochschulumfeld • Erkenntnisse über Herausforderungen der Microservice-Architektur
Name	Herr Prof. Dr. Stefan Bente
Kontaktdaten	Raum 1506, Gebäude LC6 stefan.bente@th-koeln.de
Verfügbarkeit	nach Vereinbarung
Wissen	• Architektur von komplexen Systemen • Agile Vorgehensweisen & Teamprozesse

Funktion (Rolle)	Technischer Leiter
Relevanz	Hoch
Ziele/Interessen	• Erfahrungen über Eignung von Java EE7 für die Entwicklung von Microservices sammlen • Vergleichbarkeit von OASP und Java EE7 als Technologie Stack • Erfahrung mit der Umsetzung einer Continuous Integration und Delivery Pipeline für den entsprechenden Technologie Stack

Name	Axel Burghof
Kontaktdaten	axel.burghof@capgemini.com
Verfügbarkeit	nach Vereinbarung
Wissen	• Java EE7 als Plattform für Business Anwendungen

Funktion (Rolle)	Entwicklungsteam (Team 1)
Relevanz	Hoch
Ziele/Interessen	• Lauffähiges Gesamtprodukt • Einfache Nutzung der APIs von Team 2 • Verfügbarkeit der Anwendung
Name	Sebastian Domke
Kontaktdaten	sebastian.domke@th-koeln.de
Verfügbarkeit	Donnerstags von 10-17 Uhr
Wissen	• API und Besonderheiten der Team 1 Microservices • Kennt "Work in progress" und potentielle Änderungen an der API

2 Randbedingungen

2.1 Technische Randbedingungen

Randbedingung	Erläuterung
Software-Infrastruktur	• WildFly Java Application Server (auf Kundenvorschlag hin) • Jenkins als Build-System • GitLab (git) als Code Versionsverwaltung • PostgreSQL als Datenbanksystem, da bereits auf dem genutzten Server verfügbar
Verfügbarkeit der Laufzeitumgebung	• Betrieben in Hochschul-IT-Infrastruktur ○ Annähernd 24/7, ohne Garantie
Grafische Oberfläche	• Es soll lediglich eine einfache UI bereitgestellt werden, die die implementierten Funktionalitäten des Backends demonstriert
Bibliotheken, Frameworks und Komponenten	• Java EE7 für Backend
Programmiersprachen	• Backend: Aufgrund von Java EE7 als Vorgabe Java • Frontend: HTML, CSS und vermutlich Javascript
Referenzarchitektur	• Entwicklung nach dem Microservice-Entwurfsmuster
Technische Kommunikation	• Kommunikation zwischen Microservices und Clients geschieht über REST Schnittstellen, die Daten als JSONs über HTTP austauschen

2.2 Organisatorische Randbedingungen

Randbedingung	Erläuterung
Organisation und Struktur	
Eigenentwicklung oder externe Vergabe	Eigenentwicklung
Entwicklung als Produkt oder zur eigenen Nutzung?	Produkt dient zur Erprobung der Microservice Architektur mit Java EE7 mit einer DevOps Pipeline, um Erkenntnisse für Kundenprojekte zu gewinnen.

Randbedingung	Erläuterung
Ressourcen (Budget, Zeit, Personal)	
Zeitplan	• Beginn: 6.10.2016 • Ende Implementierung: 9.2.2017
Zeitplan und Funktionsumfang	Da der Zeitplan fest ist, kann es zu Einbußen bei der Funktionalität kommen.
Release-Plan	Jede 4 Wochen - Sprint-Review zur Bestätigung, ob Funktionalität Vorstellungen entspricht. Durch DevOps und geplante CI/CD Pipeline wird die Anwendung bei jeder Änderung ausgeliefert.

Team	• 3 Studenten
	  ○ Ein Mitglied mit Vorerfahrung mit Microservices bei Nutzung von OASP
	  ○ Ein Mitglied mit Erfahrungen in Webentwicklung
	  ○ Ein Mitglied mit guten Kenntnissen in mathematischer Programmierung
	• Donnerstags als Arbeitstag zum kollokierten Arbeiten

Organisatorische Standards	
Vorgehensmodell	Scrum
Entwicklungswerkzeuge	s. Technische Randbedingungen
Konfigurations- und Versionsverwaltung	GitLab
Abnahme- und Freigabeprozesse	Stetige Auslieferung durch Continuous Integration und Delivery

2.3 Konventionen

2.3.1 Definition of Done

Durch die Definition of Done wird festgelegt, ab wann bestimmte Aufgaben des Projekts als abgeschlossen gelten. Unter anderem definiert diese die Kriterien zur erfolgreichen Fertigstellung des Sprint Plannings und der Erstellung der User Stories eines Sprints. Zusätzlich gibt sie an, welche Kriterien ein Task erfüllen muss, um als abgeschlossen gewertet zu werden.

2.3.2 Coding Guidelines

Die Coding Guidelines definieren Style Conventions, die garantieren, dass alle Entwickler stets einheitlichen Programmcode entwickeln. Zusätzlich geben diese Guidelines an, die nicht die Syntax des Codes, sondern die Semantik und weitere Aspekte betreffen. So soll der Code

beispielsweise simpel gehalten werden und nach dem Test Driven Development Konzept (siehe Abschnitt Testbarkeit) vorgegangen werden.

2.3.3 Git

Bei der Integration von Änderungen in das System durch Git muss unter anderem darauf geachtet werden, dass der Programmcode fehlerfrei kompiliert und alle Tests (siehe Testbarkeit) erfolgreich durchlaufen. Zusätzlich sollte jeder Commit eine Beschreibung enthalten, die klar erkennen lässt, was geändert wurde.
All dies ist ebenfalls in den Coding Guidelines festgelegt.

Aufgrund der eingesetzten CI/CD-Pipeline, und der daraus resultierenden regelmäßigen Auslieferung der Anwendung, muss der Programmcode stets ausführbar sein. Daher wurde stets lediglich auf einem einzigen Git-Branch (master) gearbeitet.

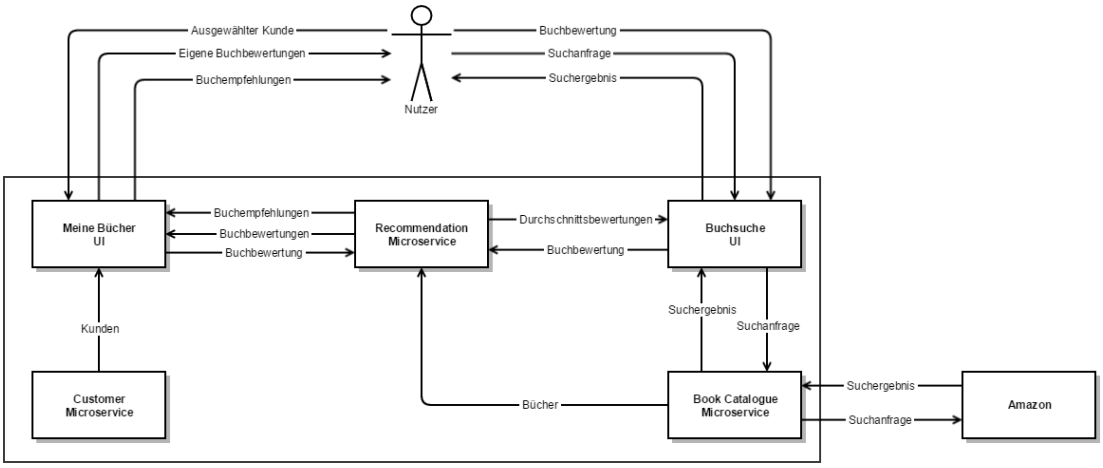
3 Kontextabgrenzung

Die folgenden Unterkapitel zeigen die Einbettung des entwickelten Systems in seine Umgebung und definiert dazu die fachlichen, sowie die technischen Schnittstellen des Systems.

3.1 Fachlicher Kontext

Bei dem entwickelten System handelt es sich um eine Microservice-Architektur, in der eine Anwendung aus mehreren eigenständigen Diensten besteht. Jeder einzelne Microservice stellt für sich ein Subsystem mit eigenen Schnittstellen dar, weshalb nicht das ganze System als Blackbox betrachtet wird, sondern jeder Microservice für sich als Blackbox angesehen werden kann. Nachbarsysteme, bzw. Kommunikationspartner, sind aufgrund dieser Architektur nicht nur Fremdsysteme, sondern auch die im System enthaltenen Microservices untereinander. Zusätzlich sind User Interfaces im System enthalten, die ebenfalls als Subsysteme gewertet werden können.

Das nachfolgende Diagramm zeigt alle im System enthaltenen Subsysteme und deren Kommunikation miteinander, sowie die Kommunikation mit Fremdsystemen.



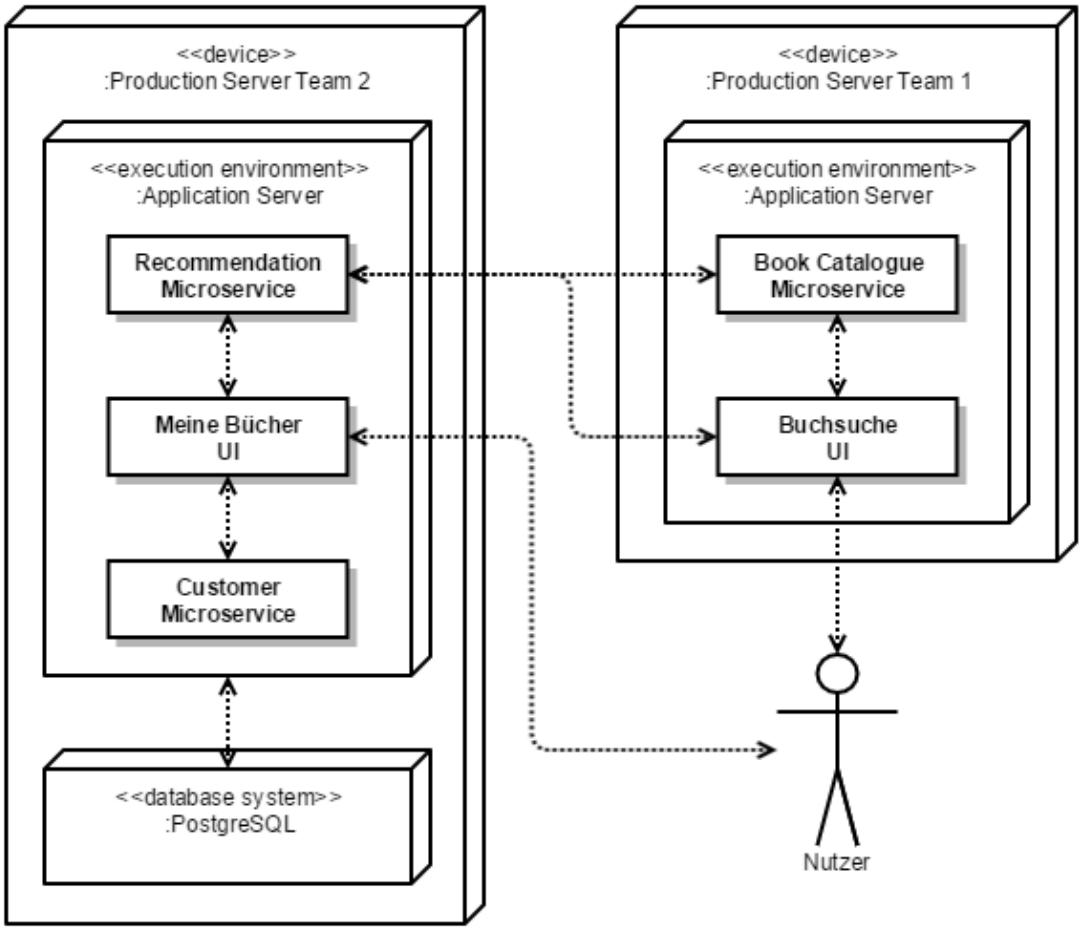
Die nachfolgende Tabelle listet die Kommunikation zwischen den verschiedenen Subsystemen auf und beschreibt dabei kurz die Eingabe-, sowie die Ausgabedaten.

Anmerkung: Interaktionen zwischen dem Book Catalogue Microservice, dem Buchsuche UI, Amazon und Nutzern werden an dieser Stelle nicht beschrieben, da die genannten Systeme von Team 1 entwickelt wurden.

Subsystem	Kommunikationspartner	Eingabe	Ausgabe
Recommendation Microservice	UI Meine Bücher	Buchbewertung: Wird von einem Nutzer eine bestehende Buchbewertung abgeändert, wird diese anschließend an den	Buchbewertungen: Alle von einem Kunden getätigten Buchbewertungen werden zur Darstellung an das Meine Bücher UI übermittelt. Buchempfehlungen: Bis zu drei Buchempfehlungen

Subsystem	Kommunikationspartner	Eingabe	Ausgabe
		Recommendation MS weitergeleitet.	(anhand der Bewertungen eines Kunden ermittelt) werden zur Darstellung an das Meine Bücher UI übermittelt.
	UI Buchsuche	Buchbewertung: Eine von einem Nutzer getätigte Bewertung eines bestimmten Buchs.	Durchschnittsbewertungen: Die Durchschnittsbewertungen aller von dem Buchsuche UI angefragten Büchern.
	Book Catalogue Microservice	Bücher: Buchdaten, die der Recommendation MS zur Weiterleitung von Bewertungen und Empfehlungen zu einem Nutzer an das Meine Bücher UI benötigt.	
Customer Microservice	UI Meine Bücher		Kunden: Alle im System registrierten Kunden, so dass sich ein Nutzer über das Meine Bücher UI einloggen kann.
UI Meine Bücher	Nutzer	Ausgewählter Kunde: Ein Nutzer kann einen im System registrierten Kunden aus einer Liste auswählen und sich damit einloggen.	Eigene Buchbewertungen: Alle von dem eingeloggten Kunden getätigten Buchbewertungen. Buchempfehlungen: Bis zu drei Buchempfehlungen, die anhand der Buchbewertungen des eingeloggten Kunden ermittelt wurden.

3.2 Technischer Kontext



Die Kommunikation zwischen den einzelnen Microservices findet über HTTP statt. Um die Einheitlichkeit der HTTP-Kommunikation zu gewährleisten, wird das REST-Paradigma verfolgt. Jeder Microservice stellt folglich REST API's bereit, die nach dem "Smart Endpoints and Dumb Pipes"-Prinzip selbstbeschreibende Nachrichten über die HTTP-Methoden "Get", "Post", "Patch" und "Delete" erwarten. Die Nachrichten an sich sind dabei einfach gehalten, die eigentliche Logik wird von den Endpunkten und nicht dem Kommunikationskanal bereitgestellt.

Müssen Datenobjekte zwischen den Microservices oder zu den Weboberflächen transferiert werden, werden diese als JSON-Objekte serialisiert und an entsprechender Stelle anschließend wieder deserialisiert und verarbeitet.

Die Dokumentation der REST API's der von Team 2 entwickelten Microservices wird automatisch mittels Swagger generiert (siehe Abschnitt API Dokumentation) und ist mit einem Webbrowser einsehbar. Sie listet alle verfügbaren Endpunkte eines Microservices, sowie erwartete Parameter oder JSON-Objekte, auf. Unter folgenden Weblinks sind die Dokumentationen der zwei entwickelten Microservices abrufbar:

Recommendation Microservice: http://fsygs15.gm.fh-koeln.de:8280/swagger/?url=http://fsygs15.gm.fh-koeln.de:8280/recommendation-ms/swagger.json

Customer Microservice: http://fsygs15.gm.fh-koeln.de:8280/swagger/?url=http://fsygs15.gm.fh-koeln.de:8280/customers-ms/swagger.json

Die nachfolgende Tabelle mappt die von den beiden Microservices bereitgestellten fachlichen Schnittstellen auf die technischen Schnittstellen und beschreibt diese kurz.

Subsystem	Fachliche Schnittstelle	Technische Schnittstelle	Beschreibung
Recommendation Microservice	Ausgabe: Durchschnittsbewertungen	HTTP GET: /books/ratings/bookIds=<x,y,z,...>	Gibt eine Liste von Durchschnittsbewertungen zurück. Durch den Parameter "bookIds" können bis zu 20 Ids von Büchern angegeben werden, von denen die Durschnittsbewertung benötigt wird.
	Eingabe: Buchbewertung	HTTP POST: /books/{bookId}/rate/{rating}	Erstellt oder überschreibt eine Buchbewertung. Der Parameter "bookId" gibt die Id des zu bewertenden Buchs, "rating" die Bewertung dieses an. Die Bewertung muss ein Wert zwischen 1 und 5 sein.
	Eingabe: Buchbewertung	HTTP DELETE: /ratings/{ratingId}	Löscht eine bestehende Buchbewertung mit der durch den Parameter "ratingId" spezifizierten Id.
	Ausgabe: Buchbewertungen	HTTP GET: /users/{userId}/books	Gibt eine Liste der Bewertungen eines Kunden zurück. Durch den Parameter "userId" wird der Kunde angegeben, von dem die Bewertungen erhalten werden sollen.
	Ausgabe: Buchempfehlungen	HTTP GET: /users/{userId}/recommendations	Gibt bis zu drei Bücher als Empfehlung für den durch den Parameter "userId" spezifizierten Kunden zurück.
Customer Microservice	Ausgabe: Kunden	HTTP GET: /customers	Gibt eine Liste aller im System enthaltenen Kunden zurück.
	Ausgabe: Kunde	HTTP GET: /customers/{customerId}	Gibt den Kunden mit der durch den Parameter "customerId" spezifizierten Id zurück.

4 Lösungsstrategie

4.1 Austauschbarkeit

Um von einem fremden Microservice bei der Entwicklung nicht abhängen zu müssen, wurden 2 Klassen, eins mit Zugriff auf fremdes Microservice (BooksRepositoryMicro), anderes mit Dummy Daten (BooksRepositoryStub), erstellt, die selbes Interface implementieren.

Um eins der Klassen durch Dependency Injection aufrufen zu können, muss das andere (BooksRepositoryMicro) mit Alternative-Annotation versehen werden.

Durch den Maven (s. Konfigurierbarkeit) wird während des Deploy-Vorgangs eins der Repositories durch den Austausch von **beans.xml** aktiv gesetzt.

4.2 Eingenständiges UI

Bei der Entwicklung von Microservices benötigt jedes eine UI Ansicht, falls die nötig ist. Um bei wachsendem Anzahl an Microservces ein einheitliches Design zu haben, wurde UI in ein eigenes Projekt ausgelagert.

Damit werden alle User-Interaktionen, sowie Darstellung der Informationen, zentral behandelt und die nötigen Microservices von UI abgerufen.

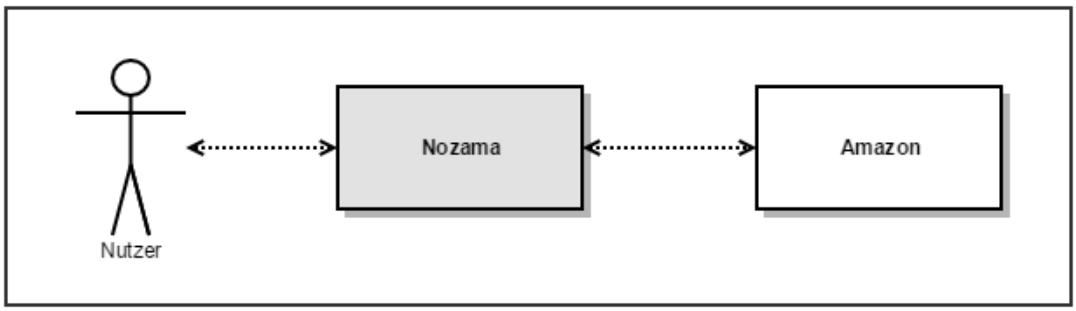
Dies erlaubt die Entwicklung an Microservices nur auf Lesen und Bearbeiten von Informationen zu begrenzen.

Die Microservices brauchen keine UI-Schnittstellen, wodurch die Menge an Übertragungsdaten sinkt, was für eine höhere Antwortzeit sorgt.

4.3 Entity-Control-Boundary Pattern

Um die Klassen innerhalb eines Microservices funktional aufzuteilen, wurde das Entity-Control-Boundary Pattern (ECB) eingesetzt. Mit ECB wird der Fokus auf die Trennung der Verantwortlichkeiten der Elemente gelegt. In Typische Muster, Strukturen und Abläufe wird dies näher erläutert.

5 Bausteinsicht

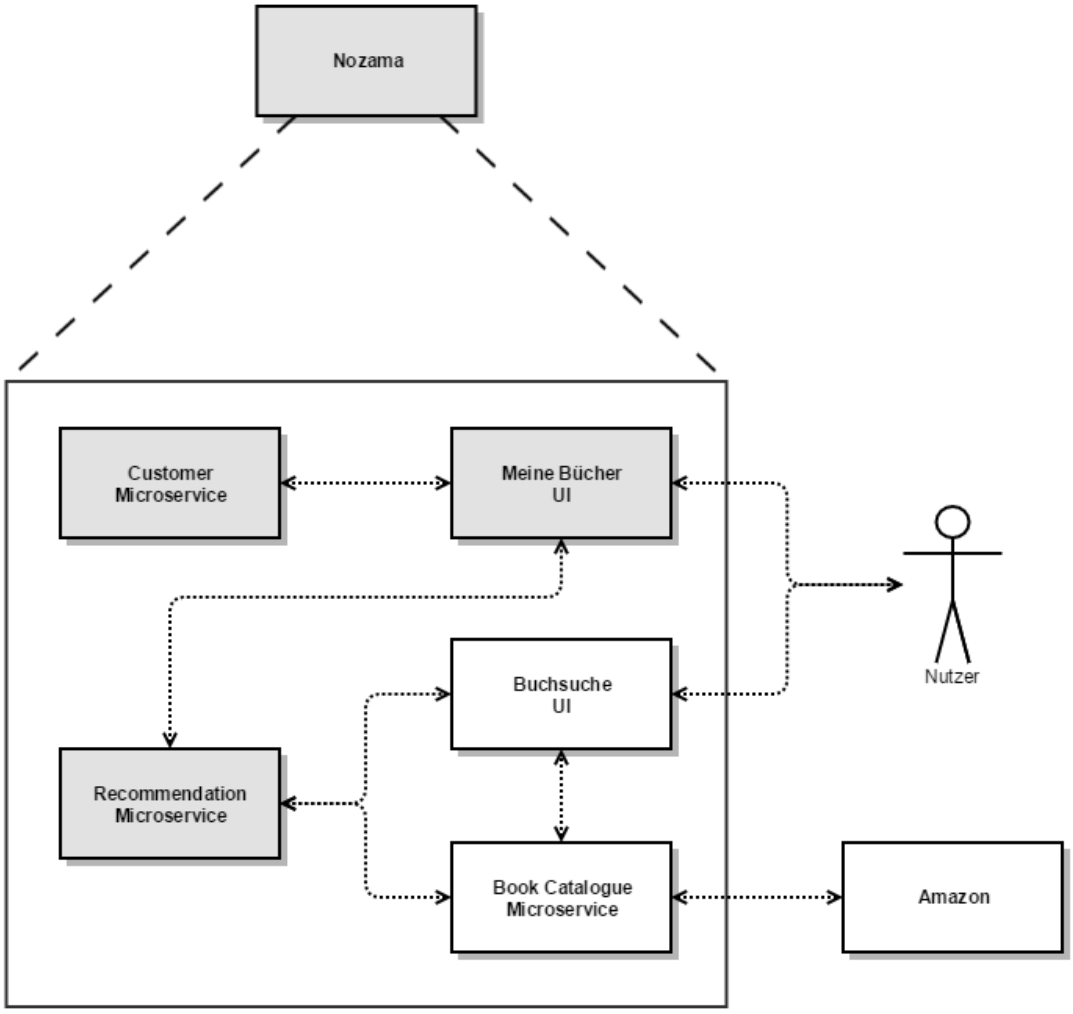


Die im obigen Diagramm dargestellte Kontextabgrenzung ist relativ übersichtlich. Das entwickelte System Nozama interagiert lediglich mit Nutzern und mit Amazon als Fremdsystem.

Die nächsten Abschnitte zeigen nach und nach die im System enthaltenen Komponenten und deren Interaktion miteinander, wobei jede Ebene die Whitebox-Beschreibung der zuvor beschriebenen Blackboxen enthält.

- Ebene 1: Whitebox-Darstellung des entwickelten Systems Nozama, welches die einzelnen Microservices und deren Interaktion miteinander darstellt.
- Ebene 2: Beschreibung der in den von Team 2 entwickelten Microservices (Customer Microservice, Recommendation Microservice) enthaltenen Komponenten.
- Ebene 3: Darstellung des Aufbaus der in Ebene 2 beschriebenen Komponenten.

5.1 Ebene 1



5.1.1 Übersicht

Vorgabe des Projektes war, eine Microservice-Architektur zu entwickeln. Folglich besteht die entwickelte Plattform Nozama aus mehreren eigenständigen Bausteinen, die im obigen Übersichtsdiagramm dargestellt sind.
Alle Komponenten in der Whitebox, die grau hinterlegt sind, wurden dabei von Team 2, die Komponenten mit weißem Hintergrund von Team 1 entwickelt. Neben den erstellten Microservices haben beide Teams jeweils eine Weboberfläche realisiert, die als User Interface zur Interaktion mit Nutzern dient.

Der von Team 1 entwickelte Book Catalogue Microservice interagiert außerdem mit dem Fremdsystem Amazon, was an dieser Stelle allerdings nicht weiter ausgeführt wird, da dies bereits Inhalt der Dokumentation von Team 1 ist.

Die Aufteilung des Systems in mehrere Microservices erfolgte größtenteils nach den in der Aufgabenstellung geforderten umzusetzenden Funktionalitäten. So entwickelte Team 1 einen Microservice zur Verwaltung eines Buchkatalogs und zur Buchsuche (Book Catalogue Microservice), während Team 2 eine Kundenverwaltung (Customer Microservice) und ein Bewertungs- und Empfehlungssystem (Recommendation Microservice) realisieren sollte. Der Grund dafür, dass das Bewertungs- und Empfehlungssystem in einen Microservice zusammengelegt wurde, kann in Abschnitt Teilung der Microservices nachgelesen werden.

Die Interaktion zwischen den verschiedenen Komponenten erfolgt mittels REST API's, die von den einzelnen Microservices angeboten werden (siehe Abschnitt Technischer Kontext).

Der nachfolgende Abschnitt enthält die Beschreibung der relevanten Blackboxen (Customer Microservice, Recommendation Microservice, Meine Bücher UI).

5.1.2 Beschreibung der Blackboxen

5.1.2.1 Customer Microservice

Der Customer Microservice dient der Verwaltung von Kunden. Die einzige Funktionalität dieses Microservices liegt jedoch lediglich im Abruf der gesamten Kundenliste. Das Erstellen des Kundenbestands erfolgte initial mit zufällig generierten Daten (siehe Abschnitt Migration). Laut Aufgabenstellung waren keine Funktionalitäten zum Erstellen (während des Betriebs), Ändern oder Löschen von Kunden erforderlich.

Die einzige bereitgestellte Schnittstelle ist somit nur der Abruf einer Liste von allen Kunden, die von der Meine Bücher UI benötigt wird, um Nutzern die Möglichkeit zu geben, sich als ein bestehender Kunde in das System einzuloggen.

5.1.2.2 Recommendation Microservice

Das Bewertungs- und Empfehlungssystem ist, wie bereits angemerkt, in einem Service zusammengefasst. Der Recommendation Microservice verwaltet die Buchbewertungen von allen Kunden und sucht anhand dieser bis zu drei Buchempfehlungen für einen Kunden, um sie dem Meine Bücher UI zur Darstellung zu übermitteln.

Über die REST Schnittstelle des Recommendation Microservice können Buchbewertungen angefragt, hinzugefügt, geändert und gelöscht werden. Zusätzlich können Buchempfehlungen für einen bestimmten Kunden angefragt werden.

Da die Buchdaten vom Book Catalogue Microservice verwaltet werden, stellt der Recommendation Microservice Anfragen an dessen REST API, um die Daten zu erhalten, die für die Darstellung der Buchbewertungen und -empfehlungen im Meine Bücher UI benötigt werden.

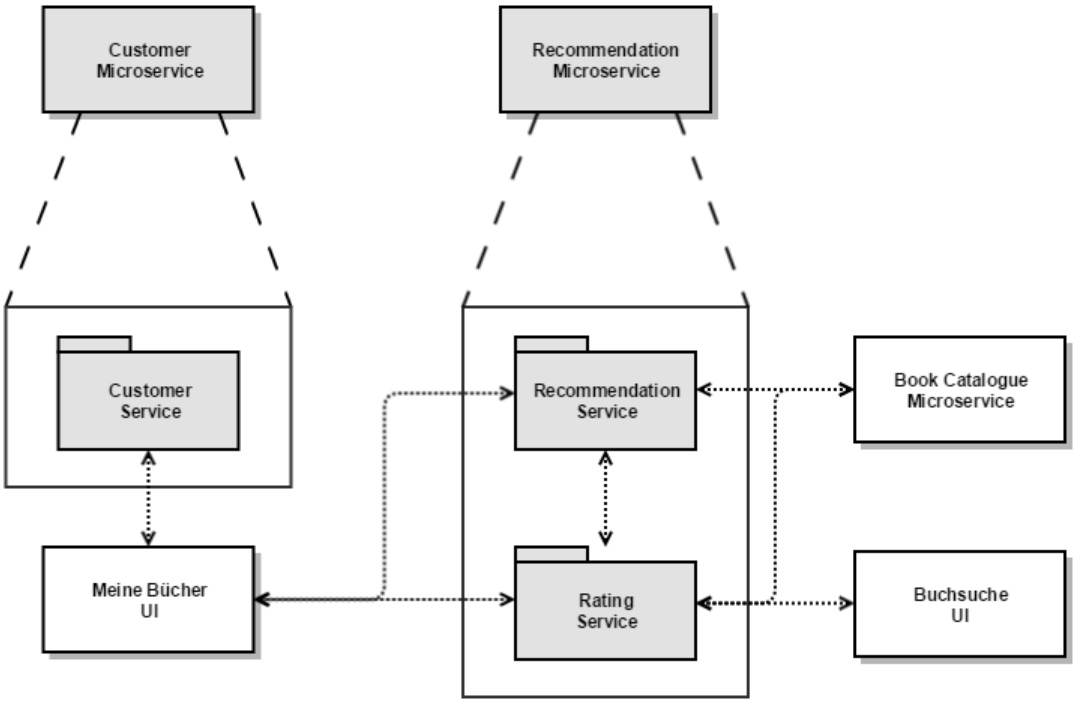
5.1.2.3 Meine Bücher UI

Nutzer können sich über das Meine Bücher UI einloggen, in dem sie einen bestehenden Kunden aus einer Liste auswählen. Ein eingeloggter Kunde kann sich ebenfalls wieder ausloggen.

Eingeloggte Kunden können über die Weboberfläche eine Liste der bereits bewerteten Bücher sehen und die eigenen Bewertungen ändern oder auch löschen. Gibt es für einen Kunden Buchempfehlungen, werden diese ebenfalls in der Weboberfläche angezeigt.

Das Meine Bücher UI greift auf die REST API des Customer Microservice zu, um eine Liste aller registrierter Kunden zu erhalten. Mittels der REST API des Recommendation Microservice können die Buchbewertungen eines Kunden, sowie mögliche Buchempfehlungen erhalten werden.

5.2 Ebene 2



Die Whitebox-Sicht der Komponenten, im obigen Diagramm dargestellt, ist überschaubar, da es sich um Microservices handelt, die der Unix-Philosophie folgen: "Do One Thing and Do It Well".

Da die Funktion der entwickelten Microservices bereits auf Ebene 1 der Bausteinsicht beschrieben wurde, wird an dieser Stelle lediglich kurz auf den Recommendation Microservice eingegangen, da er zwei Dienste vereint.
Aus diesem Grund besteht dieser, im Gegensatz zum Customer Microservice, aus zwei Komponenten: Dem Recommendation Service und dem Rating Service. Warum diese nicht in zwei Microservices aufgespalten wurden, wird in Abschnitt Teilung der Microservices erläutert.

5.2.1 Recommendation Service

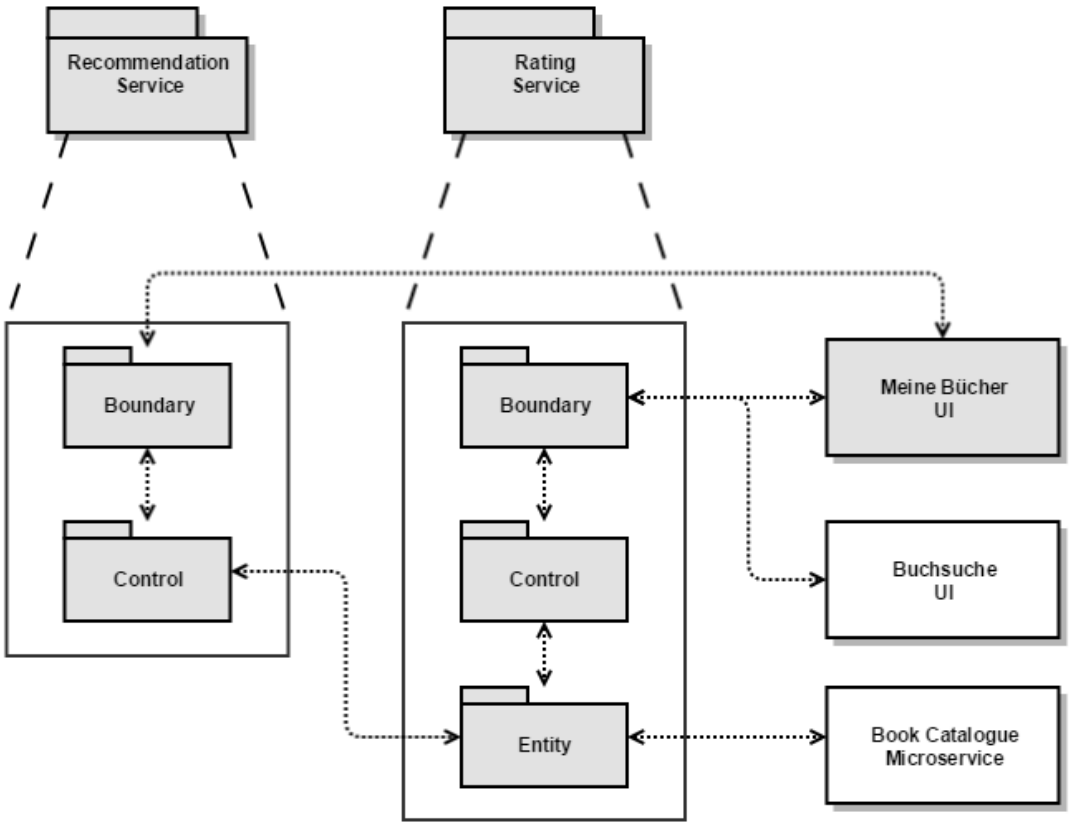
Dieser beinhaltet den Algorithmus zur Berechnung der Ähnlichkeiten von Kunden und zur Ermittlung von Buchempfehlungen anhand dieser Ähnlichkeitswerte (siehe Abschnitt Empfehlungssystem). Vom Rating Service erhält er die für die Berechnung benötigten Buchbewertungen.
Wurden Empfehlungen für einen Kunden gefunden, werden die dazugehörigen Buchdaten vom Book Catalogue Microservice angefragt und diese anschließend zur Anzeige für den Nutzer an das Meine Bücher UI gesendet.

5.2.2 Rating Service

Buchbewertungen werden durch den Rating Service verwaltet. Auf die Anfrage vom Buchsuche UI hin berechnet dieser die Durchschnittsbewertungen der angefragten Bücher und übermittelt diese anschließend in der Antwort.
Bewertet ein Kunde im Buchsuche UI ein Buch, so wird diese Bewertung an den Rating Service gesendet und von diesem persistent abgespeichert.

Wie beim Recommendation Service werden Buchdaten vom Book Catalogue Microservice angefragt, um diese anschließend an das Meine Bücher UI zur Anzeige der Buchbewertungen eines Kunden zu übermitteln.

5.3 Ebene 3



Das obige Diagramm stellt die Struktur der zwei Services des Recommendation Microservice dar. Da der Aufbau des Customer Microservice ähnlich ist, wird er an dieser Stelle nicht aufgeführt.

Jeder entwickelte Service ist nach dem Entity-Control-Boundary (ECB) Entwurfsmuster gegliedert, welches eine Variation des bekannten Model-View-Controller (MVC) Entwurfsmusters ist.
Die Boundary-Schicht der Services stellt dabei die Schnittstelle nach außen dar und beinhaltet somit die REST API. Im Gegensatz zum MVC Entwurfsmuster enthält die Control-Schicht die komplette Geschäftslogik. Die Entity-Schicht repräsentiert die Datenobjekte.
Mehr über die ECB-Struktur ist im Abschnitt Typische Muster, Strukturen und Abläufe zu finden.

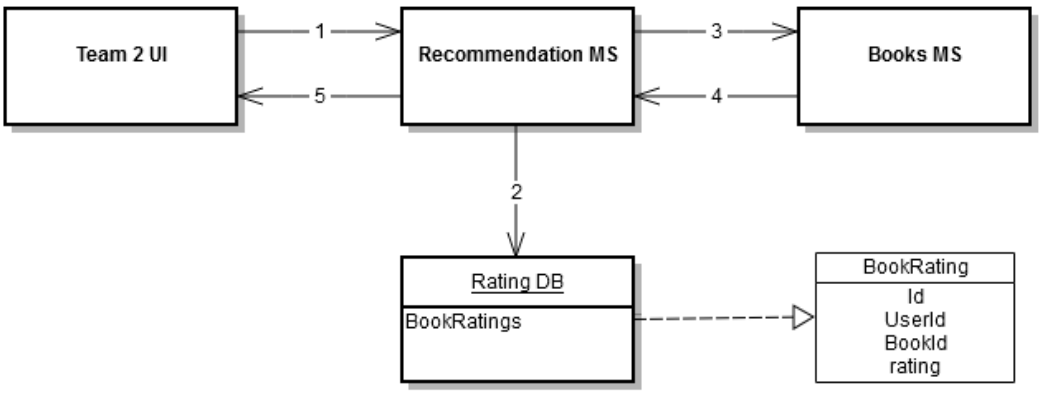
Werden über die REST API in der Boundary-Schicht Daten angefragt, wird diese Anfrage an den zugehörigen Controller weitergeleitet, der die benötigten Daten anschließend aus der Entity-Schicht abruft, aufbereitet und schließlich zur Antwort zurück an die Boundary-Schicht gibt.

Eine Besonderheit ist, dass Buchdaten vom Book Catalogue Microservice angefragt werden müssen, sofern diese benötigt werden, da dieser den Buchkatalog verwaltet. Dies erfolgt in der Entity-Schicht des Rating Service.
Doch nicht nur der Rating Service benötigt Buchdaten, sondern ebenfalls der Recommendation Service, was zu einer Interaktion zwischen diesen beiden Services führt.

6 Laufzeitsicht

In den folgenden Unterabschnitten werden die Interaktionen und Abläufe der Systeme näher beschrieben. Dadurch soll das Zusammenspiel der verschiedenen Komponenten verdeutlicht werden.

6.1 Laufzeitszenario 1: Meine Bücher



Sobald ein eingeloggter Nutzer auf die "Meine Bücher" Ansicht geht, werden ihm die von ihm bewerteten Bücher angezeigt.

Das passiert in 6 Schritten:

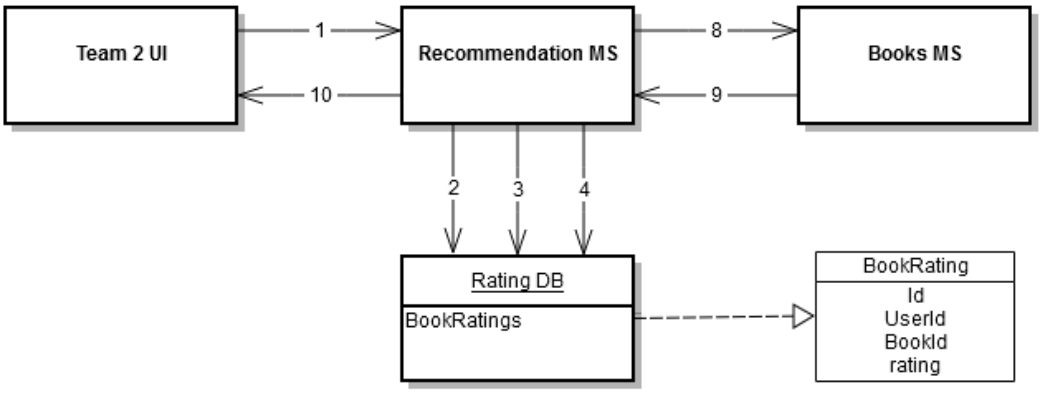
1. Der eingeloggte Nutzer wird ermittelt (s. Sessionbehandlung), und die Anfragen mit der *UserId* an den Recommendation Microservice verschickt.
2. Die API des Recomandation Microservices bekommt die Anfrage mit *UserId* und Seitennummer. Aus der Datenbank werden die zur angeforderten Seite gehörenden Bewertungen gelesen.
3. An den Book Microservice wird eine Anfrage mit den zu den Bewertungen gehörenden *BookIds* verschickt.
4. Die API des Book Microservices bekommt die Anfrage mit Array an *BookIds*. Aus der Datenbank werden die entsprechenden Bücher gelesen und als Antwort an den Recommendation Microservice verschickt.
5. Der Recommendation Microservice empfängt die Bücher vom Book MS, erstellt ein neues Objekt mit allen relevanten Daten, wie Buchinformationen und dessen Bewertung, und sendet es weiter an das UI.
6. Das UI bekommt die Bücher des Nutzers und stellt sie für den Nutzer dar.

Dabei gibt es einige Besonderheiten:

3. Wenn keine Bewertungen gefunden werden, wird sofort zu Schritt 5 übergegangen.

5. Erfolgt nach bestimmter Zeit keine Antwort oder kommt es zu einem Fehler bei der Anfrage an den Book MS, wird eine entsprechende Nachricht an die UI verschickt.

6.2 Laufzeitszenario 2: Empfehlungen



Auf "Meine Bücher" Ansicht werden dem Nutzer außer seiner Bücher noch 3 Bücher als Empfehlung angezeigt.

Die Empflehlungen werden folgend ermittelt:

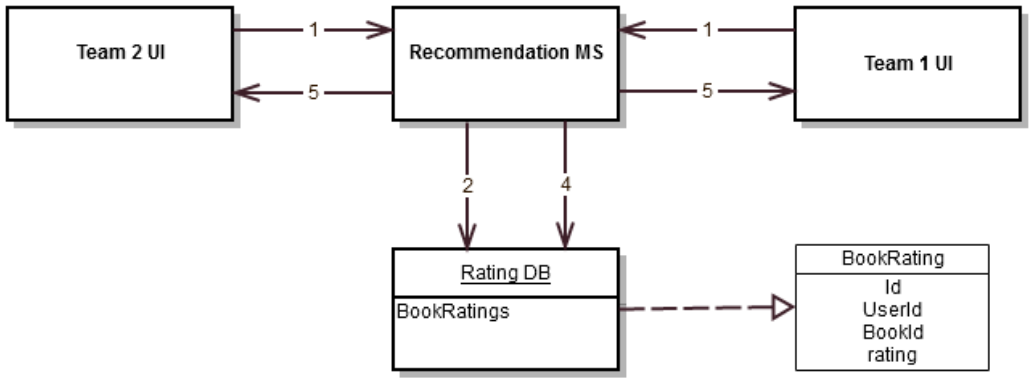
1. Das UserId des eingeloggten Nutzers wird an das Recommendation Microservice verschickt.
2. Aus dem Datenbank werden alle bewerteten Bücher des Nutzers anhand seiner Id ermittelt.
3. Aus dem Datenbank werden alle Nutzer, die mindestens eine Bewertung gemacht haben, ermittelt. Die Bewertungen des eingeloggten Nutzers werden dabei ignoriert.
4. Aus dem Datenbank werden für jeden Nutzer aus 3. die Bewertungen zu den Büchern aus 2. ermittelt.
5. Die Ähnlichkeit zwischen dem eingeloggten und den anderen Nutzern wird durch eine Ähnllichkeitsfunktion ermittelt.
6. Die Nutzer werden anhand deren Ähnlichkeit absteigend sortiert.
7. Von dem ähnlichsten Nutzer werden 3 am besten bewerteten Bücher ermittelt, die der eingeloggter Nutzer noch nicht bewertet hat.
8. Die Informationen zu den Büchern aus 7. werden am Book Microservice abgefragt.
9. Die API des Book Microservices bekommt die Anfrage mit Array an Buch ids. Aus dem Datenbank werden die entspechende Bücher gelesen und als Antwort an Recommendation Microservice verschickt.
10. Recommendation Microservice empfängt die Bücher vom Book MS, erstellt ein neues Objekt mit allen relevanten Daten, wie Buchinformationen und dessen Bewertung, und sendet es weiter an das UI.
11. Das UI bekommt die Bücher und stellt sie für dem Nutzer dar.

Besonderheit:

7a. Hat der ähnlichster Nutzer kene 3 Bücher, die ger eingeloggter Nutzer noch nicht vewertet hat, so wird der nächsttähnlichster Nutzer gelesen und die Empfehlung wird it seinen Büchern vervolllständigt.

So kann die dritte Empfehlung eine höhere Bewertung haben, als vorherge, da die von unterschiedlichen Nutzern stammen.

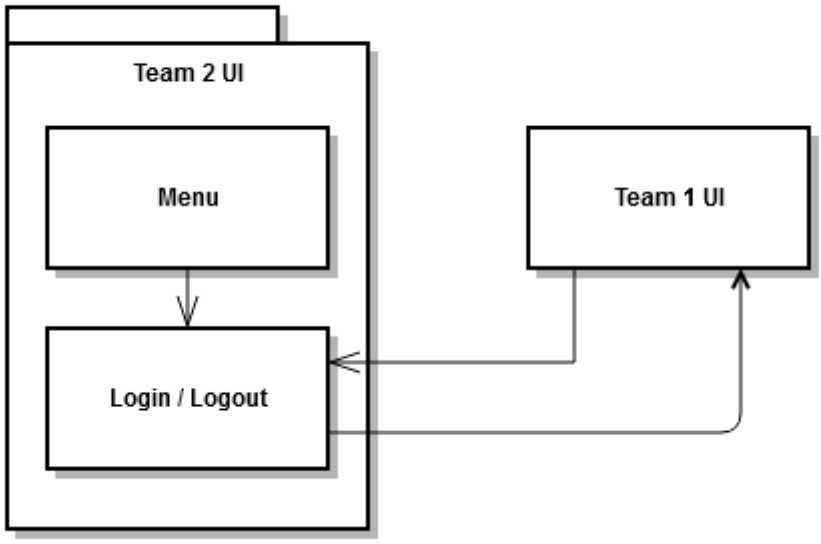
6.3 Laufzeitszenario 3: Bewertung



Ob eine Bewertung vom UI der Team 1 oder 2 kommt, verläuft die Buchbewertung für beide Startpunkte gleich. Der einziger Unterschied liegt nur in der UI, wo in Team 2 UI man nur eine Bewertung ändert kann und keine neue abgeben.

1. Das UI sendet Userdaten, BuchId und Buchbewertung an das Recommendation Microservce.
2. Das Microservice prüft nach validen Bewerung und wenn die Bewertung zwischen 1 und 5 liegt, wird die Bewertung im Datenbank gespeichert. Anderenfalls wird ein Fehler erzeugt und mit Statuscode 400 an das UI gesendet.
3. Vor der Speicherung wird vom Datenbank nach existierenden Bewertung anhand voon BuchId und UserId nachfragt. Existiert keins, wird ein neues Eintrag erzeugt. Existiert bereits eine Bewertung vom Nutzer zu diesem Buch, so wird die auf neues Wert korrigiert.
4. Der neue Durchschnittswert zu dem Buch wird vom Datenbank abgefragt
5. Ein neues Objekt mit BuchId und seinem neuen Durchschnittswert wird erzeugt und an UI gesendet.

6.4 Laufzeitszenario 4: Autorisierung

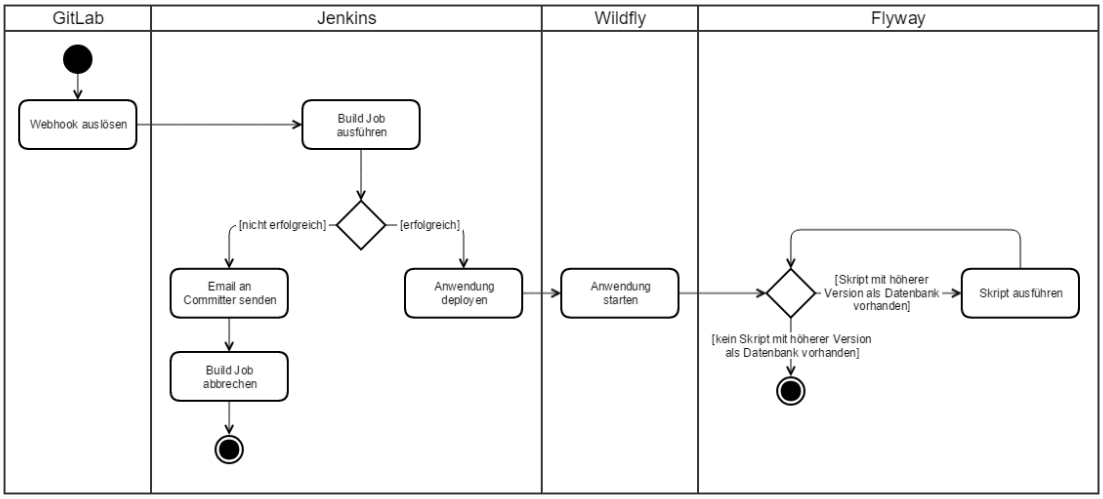


Die Autosierung wird in Form von Sessionbehandlung auf dem Team 2 UI umgesetzt.

Beim Login wird in Team 2 UI ein cookie des angemeldeten Nutzers gespeichert, der auf Team 1 UI als Übergabeparameter zu Authentifizierung dient.

Beim Logout wird das cookie auf Team 2 UI gelöscht und beim wechsel auf Team 2 UI kein Parameter mitgeschickt.

6.5 Laufzeitszenario 5: CI/CD-Pipeline



Das oben dargestellte Diagramm zeigt den Ablauf der CI/CD-Pipeline, angefangen mit einem ausgeführten Git-Push auf das Repository eines der entwickelten Systeme, anhand eines Aktivitätsdiagramms. Da die Vorgänge bereits im Abschnitt CI/CD-Pipeline beschrieben werden, wird an dieser Stelle nicht genauer auf diese eingegangen.

7 Verteilungssicht

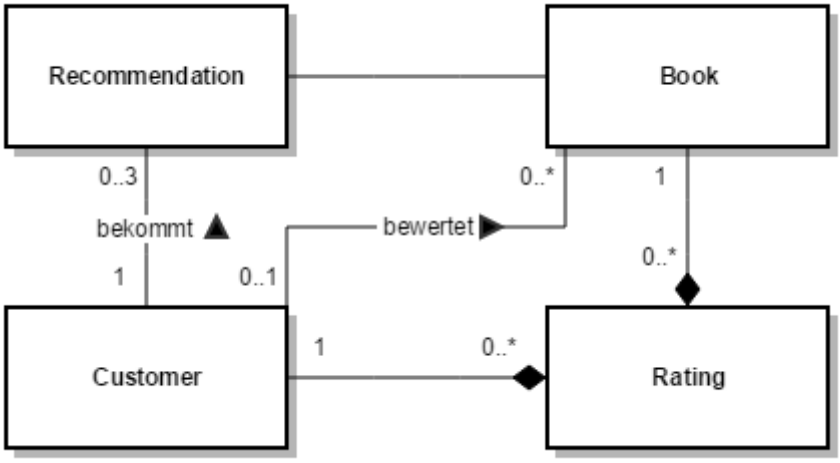
Im Rahmen des Projektes stand letztendlich nur eine Produktionsumgebung zur Verfügung auf die automatisiert deployed wurde. Dies ist dem Rahmen des Hochschulumfelds und der verfügbaren Infrastruktur geschuldet.

Da diese bereits ausreichend durch das Diagramm im Abschnitt Technischer Kontext beschrieben wurde, soll an dieser Stelle nicht weiter darauf eingegangen werden.

Die Verteilung bzw. das Deployment der Anwendung wird darüberhinaus ausführlicher im Abschnitt CI/CD-Pipeline beschrieben.

8 Querschnittliche Konzepte

8.1 Fachliches Datenmodell



Modell	Eigenschaften
Customer	• Nutzer des Systems • Vor- & Nachname • Email zur Nutzererkennung
Book	• Titel • Autor • Verlag • Erscheinungsjahr
Rating	• Jeder Kunde kann Bücher auf einer Skala zwischen 1 und 5 bewerten • Vom Kunden bewertete Bücher "gehören" dem Kunden
Recommendation	• Ein Kunde bekommt, solange er selber und ausreichend andere Nutzer Bücher bewertet haben, 3 Vorschläge von Büchern, die noch nicht "in seinem Besitz" sind. ○ Diese Bücher sollen aus dem Bestand anderer Nutzer angeboten werden, die ein ähnliches Bewertungsverhalten haben.

8.2 Typische Muster, Strukturen und Abläufe

8.2.1 Typische Muster und Strukturen

8.2.1.1 Microservices

Eine grundlegende Struktur der Anwendung ist bereits durch den Microservice-Architektur-Ansatz vorgegeben.
Somit wird die Anwendung bereits nach Funktionalitäten bzw. "Business Capabilites" getrennt. Jede dieser "Business Capabilities" wird durch einen eigenen Microservice abgebildet und stellt die Funktionen über eine REST Schnittstelle bereit.

Innerhalb eines Microservices wird ein Paket für jede "Business Capability" erstellt. Üblicherweise sollte dies nur eins pro Microservice sein. Im Falle des Teams ist der selbe Microservice für die Bücherbewertungen und das Bereitstellen der Empfehlungen zuständig und deckt somit 2 "Business Capabilities" ab. Mehr dazu im Abschnitt Teilung der Microservices.
Ein funktionales Paket wird nach dem Entity-Control-Boundary Pattern unterteilt. Dieses ist eine Variante des Model-View-Control Patterns:

ECB	MVC	Funktion	Innerhalb der Anwendung
Boundary	View	Stellt die Interaktionsschnittstelle bereit	REST Controller
Control	Control	Hier findet die Anwendungslogik statt	Klassen in Form von Services, die die Anwendungslogik und Validierungsregeln abbilden
Entity	Model	Verantwortlich für die Datenhaltung	Java Entities und Repositories

ECB wurde gewählt, da die Schichten-Bezeichnungen besser zu einem Microservice passen, welcher lediglich eine REST Schnittstelle anbietet und somit keine View, bzw. diese nur im übertragenen Sinne. Da in der Entity-Schicht, welche die Persistierungsschicht darstellt, Java Entities zum Einsatz kommen, um Datenbank Tabellen auf Objekte zu mappen, war die Bezeichnung ebenfalls treffender.
Ein weiterer Vorteil ist, dass durch die Bezeichnung die Layer in der Reihenfolge des Call Stacks in der Entwicklungsumgebung angezeigt werden, was eine hilfreiche Unterstützung für das mentale Modell eines Entwicklers darstellt.

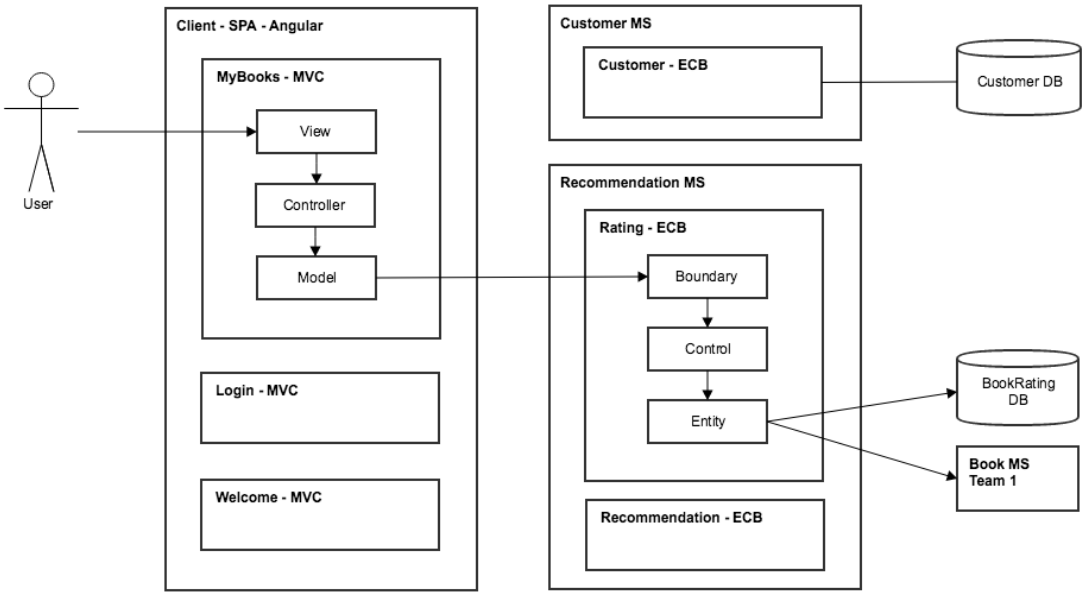
8.2.1.2 Frontend

Das Frontend ist im Vergleich als Monolith strukturiert und kommuniziert mit den verfügbaren Microservices.
Auch das Frontend ist nach Funktionalitäten in Ordner unterteilt. Darin wird allerdings das MVC Pattern genutzt. Mehr dazu in Benutzungsoberfläche.

8.2.2 Abläufe

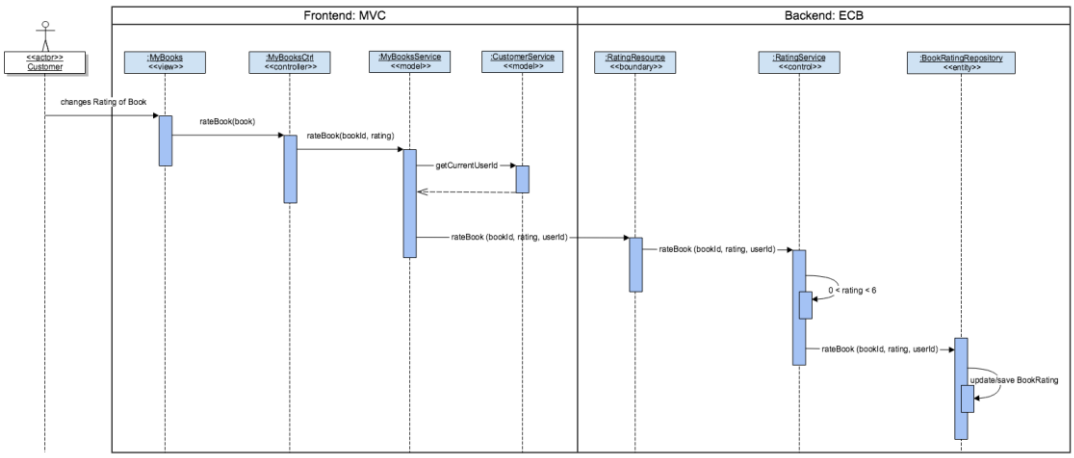
8.2.2.1 Typischer Ablauf durch die Struktur

Basierend auf der Struktur ergibt sich ein wiederkehrendes Muster, wie die Interaktion mit der Anwendung abläuft. Die Struktur, die in *MyBooks - MVC* oder *Rating - ECB* abgebildet ist, tritt genauso auch in den anderen Komponenten auf.



8.2.2.2 Beispiel eines typischen Ablaufs

Im Folgenden soll in Form eines Sequenzdiagramms der typische Ablauf innerhalb der Anwendung von der Interaktion des Nutzers mit dem Frontend bis zur Änderung in der Datenbank veranschaulicht werden. Als Beispiel wurde das Ändern eines *Ratings* gewählt. In diesem Diagramm wurde nur der Ablauf bis in die unterste Schicht abgebildet.



8.2.2.2.1 Anmerkungen

Im Beispiel eines typischen Ablaufs fallen drei Dinge auf, die sich aus der Trennung der Verantwortlichkeiten durch die Pattern ergeben:

1. Im Frontend dienen die Services als Models. Jeder Service hat eine Verantwortlichkeit. Somit muss der *MyBooksService* sich für eine Anfrage zunächst die aktuelle *UserId* vom dafür vorgesehenen *CustomerService* holen.
2. Die *RatingResource* Klasse stellt als REST Controller die Schnittstelle zur Anwendung bereit. An dieser Stelle werden die Informationen aus dem HTTP-Request entnommen und so aufbereitet, dass die Anwendungslogik damit umgehen kann.
3. In der Anwendungslogik werden die Regeln definiert und überprüft, sowie die Hauptlogik definiert. Dies ist in diesem Beispiel an der Überprüfung des *ratings* zu sehen.

8.3 Persistenz

Es werden mehrere Datenbanken verwendet, um der entwickelten Microservice-Architektur, die aus mehreren unabhängigen Services besteht, gerecht zu werden. So haben der Customer und Recommendation Microservice jeweils einen eigenen Datenbestand. Da die Services relativ klein gehalten sind, bestehen diese jeweils aus einer einzigen Datentabelle: Customers (Customer Microservice) oder BookRatings (Recommendation Microservice).

8.3.1 Customers

Id	Firstname	Lastname	Email
1	Max	Mustermann	max@mustermann.com

Kundendaten bestehen lediglich aus einer eindeutigen Id, sowie dem Vor- und Nachnamen und der E-Mail-Adresse des Kunden.

8.3.2 BookRatings

Id	UserId	BookId	Rating
1	1	abcdef	3

Buchbewertungen haben eine eindeutige Id und beinhalten die Id des Kunden, der das Buch bewertet hat, sowie die Id des bewerteten Buchs. Die Buch-Id ist dabei eine von Elasticsearch scheinbar willkürlich generierte Zeichenkette (siehe Team 1). Natürlich enthält eine Buchbewertung ebenfalls den eigentlichen Bewertungswert, welcher zwischen 1 und 5 liegen kann.

Zu beachten ist, dass lediglich in der Produktiv-Umgebung eine persistente Speicherung der Daten vorgenommen wird. In der lokalen Test-Umgebung wird mit einer In-Memory Datenbank (H2) gearbeitet, die nur zur Laufzeit der Anwendung besteht und somit alle Daten nach Beendigung dieser verwirft. In der Produktiv-Umgebung kommt das Datenbanksystem PostgreSQL zum Einsatz.

Durch die im Abschnitt Konfigurierbarkeit beschriebenen Maßnahmen erfolgt die Konfiguration der zu verwendenden Datenbank automatisch beim Deployment-Prozess. Zur Abbildung der Daten-Entitäten der Microservices (Customers, BookRatings) in relationale Datenbankobjekte wird die Java Persistence API (JPA) mit Hibernate als Implementierung verwendet. Das nachfolgende Codelisting zeigt die Konfigurationsdatei (persistence.xml) des Recommendation Microservice für die Produktiv-Umgebung.

Code Block 1 persistence.xml

```
<persistence ...>

    <persistence-unit name="recommendationDb" transaction-type="JTA">

<class>de.thkoeln.nozama.business.rating.entity.BookRating</class>

        <jta-data-source>java:jboss/datasources/recommendationDS</jta-
data-source>

        <properties>
            <property name="javax.persistence.jdbc.driver"
```

```
value="org.postgresql.Driver"/>
            <property name="javax.persistence.jdbc.url"
value="jdbc:postgresql://localhost:5432/gpmsteam2_recommendations"/>
            <property name="javax.persistence.jdbc.user"
value="gpmsteam2"/>
            <property name="javax.persistence.jdbc.password"
value="gp_ms_team_2"/>

            <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQL94Dialect"/>
            <property name="hibernate.show_sql" value="false"/>
        </properties>
    </persistence-unit>

</persistence>
```

Im Folgenden werden die wichtigsten Elemente der "persistence.xml" kurz erläutert.

persistence-unit: Definiert alle Entitätsklassen, die von einer Entity Manager Instanz verwaltet werden. Durch Setzen des "transaction-type" auf "JTA" (Java Transaction API) wird gewährleistet, dass nicht der Entwickler, sondern der Container für die Verwaltung des Entity Managers zuständig ist (siehe nachfolgenden Abschnitt).

class: Gibt die zu verwaltenden Java-Klassen (Customer, BookRating) an. Diese sind im Programmcode durch die @Entity Annotation gekennzeichnet und werden somit automatisch auf die entsprechende Datenbanktabelle abgebildet.

jta-data-source: Setzt den Datasource-Pfad zur Auffindung der Datenbank fest. Die Datasource muss im vorhinein im Application Server angelegt werden.

properties: Legt bestimmte Attribute zur Konfiguration der JPA fest, wie beispielsweise der zu verwendende Treiber oder die Verbindung zur Datenbank.

8.3.3 EJB

Ein EJB (Enterprise Java Beans)-Container dient unter anderem als Vermittlungsschicht zwischen Datenbanksystem und Anwendung. Viele Application Server, so auch der eingesetzte WildFly, stellen einen EJB-Container bereit.

Wie bereits erwähnt, wird die JTA Schnittstelle verwendet. Dies hat den Vorteil, dass der Entity Manager vom EJB-Container bereitgestellt wird und mittels einer Annotation (@PersistenceContext) während der Laufzeit der Anwendung injiziert werden kann. Dies sollte auch so umgesetzt werden, da die Erstellung des Entity Managers über die Entity Manager Factory dazu führen kann, dass verschiedene Instanzen dieses existieren, die jeweils mit einem eigenen Zwischenspeicher arbeiten, was letztendlich zu inkonsistenten Datenbeständen führen kann.

Zusätzlich werden alle Datenbanktransaktionen vom EJB-Container verwaltet, was dem Entwickler Arbeit abnimmt. Befinden sich folglich Änderungen im Zwischenspeicher des Systems, werden diese nach Abschluss der Transaktion automatisch in die Datenbank geschrieben.

Durch Annotation der Session Beans mit @Stateless wird gewährleistet, dass diese zustandslos sind, bzw. dass der Zustand der Beans, der während einer Transaktion besteht, nach Beendigung dieser nicht erhalten bleibt. Dies ermöglicht eine einfache Skalierung, was unter anderem auch der Vorteil der Verwendung der Microservice-Architektur ist.

8.3.4 Flyway

Das Datenmigrationstool Flyway wird im Projekt eingesetzt, um die zwei verwendeten Datenbanktabellen zu erstellen und initial mit Daten zu befüllen (siehe Abschnitt Migration). Dafür wurden, unter Einsatz der Java-Faker-Bibliothek, SQL-Skripte generiert, die 50 Kunden in die Customer-Tabelle und etwa 5000 Bewertungen in die BookRating-Tabelle einfügen.

8.4 Benutzungsoberfläche

Nach den Anforderungen sollte nur eine grundlegende Oberfläche bereitgestellt werden, um die Funktionalitäten, die durch die Microservices implementiert wurden, demonstrieren zu können. Dennoch sollen kurz die wichtigsten Eigenheiten beschrieben werden.

Das Frontend ist als Single Page Application realisiert worden. Dazu wurde das Javascript Framework AngularJs verwendet. Die Entscheidung dazu lag darin begründet, dass im Team das größte Know-How für ein Web-Frontend in dieser Technologie lag und sich so verstärkt auf die eigentliche Aufgabe in Form der Microservices fokussiert werden konnte.

Um sich möglichst wenig um das grundlegende Styling kümmern zu müssen, wurde das Komponenten Framework Angular Material eingesetzt. Dadurch konnten erste Erfahrungen im Umgang mit dem Flexbox Layout System gesammelt werden.

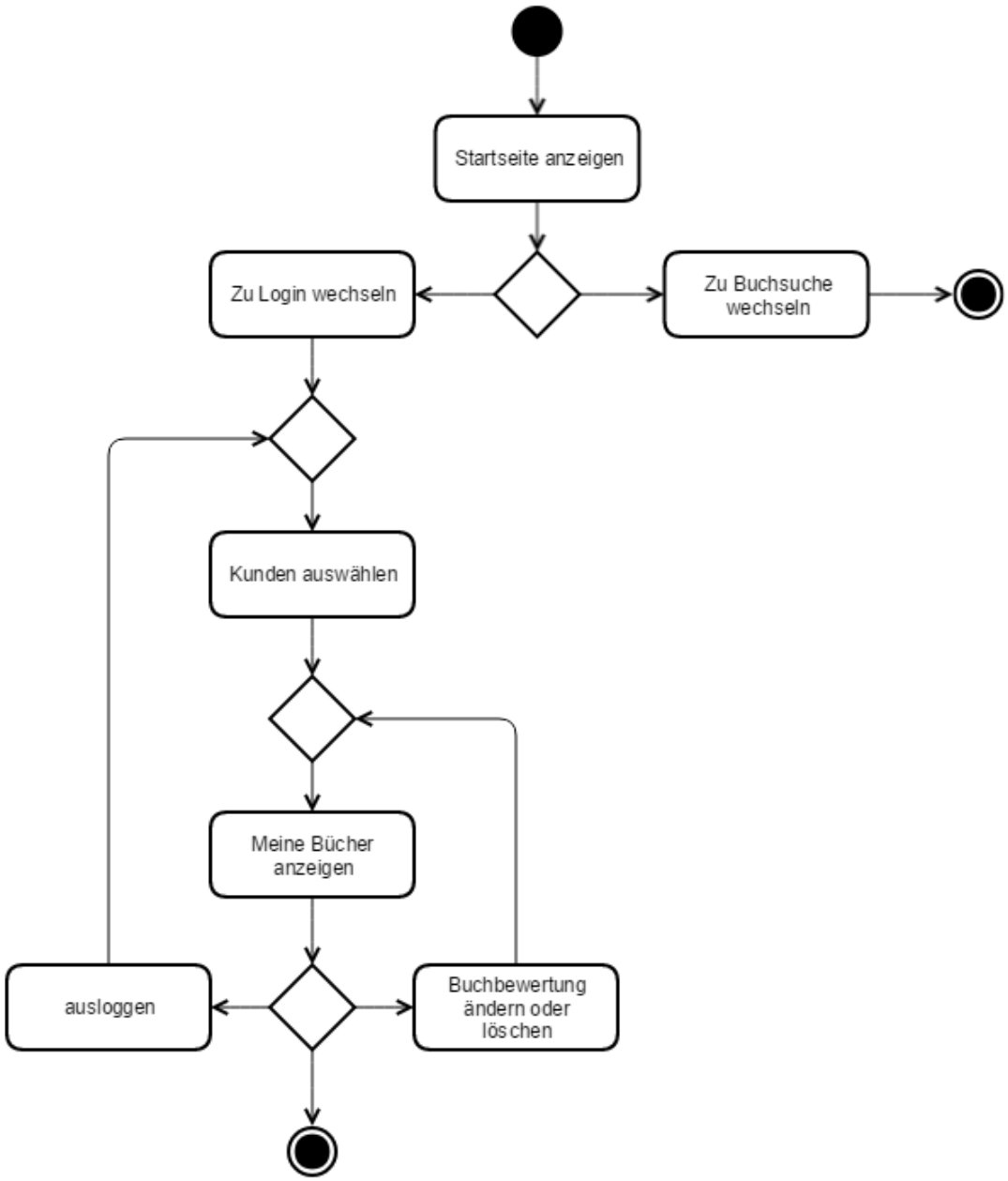
Die Anwendung ist, wie die Microservices, nach Funktionalitäten unterteilt:

- Login
- MyBooks
- Welcome

Innerhalb der drei Komponenten wird dem MVC-Pattern gefolgt.

Model	Angular Services werden genutzt, um mit Hilfe des *$resource* Services AJAX Anfragen an die REST APIs der Microservices zu stellen. Da es sich letztendlich um die grundlegenden Daten handelt, spiegelt es das Model in dem Pattern wieder.
View	HTML + CSS ergeben die Ansicht.
Controller	Angular Controller stellen Funktionalitäten bereit, die von der View benutzt werden.

8.5 Ablaufsteuerung



Das oben dargestellte Aktivitätsdiagramm veranschaulicht den Ablauf der Meine Bücher Weboberfläche.

Besucht ein Nutzer die Nozama Website zum ersten Mal, so gelangt er auf die Startseite. Über ein Navigationsmenü hat er dort die Möglichkeit, zur Login Seite oder zur vom Team 1 entwickelten Weboberfläche zu wechseln. Letzteres ist während jeder Ansicht möglich, ist aufgrund der Übersichtlichkeit nicht im Diagramm dargestellt.

Auf der Login Seite kann ein Nutzer einen bestehenden Kunden aus einer Liste auswählen und sich somit in das System einloggen. Dadurch wird ein Cookie gesetzt, der dafür sorgt, dass der Nutzer bei erneutem Aufruf der Nozama Seite nicht zur Startseite, sondern direkt zur Meine Bücher Ansicht gelangt (siehe Abschnitt Sessionbehandlung).

Die Meine Bücher Ansicht stellt die vom ausgewählten Kunden bewerteten Bücher, sowie eventuelle Buchempfehlungen dar. Der Nutzer hat dort die Möglichkeit, seine Buchbewertungen

zu löschen oder anzupassen.
Zusätzlich kann er sich wieder vom System abmelden, woraufhin er zur Login Ansicht gelangt.

8.6 Transaktionsbehandlung

Durch die Nutzung von Microservices hat jedes einzelne Microservice eine eigene Datenbank.

Im Normalfall sollte bei der Speicherung einer Buchbewertung sichergestellt werden, dass das entsprechende Buch oder der Nutzer existiert. Da dies zu weiteren Abhängigkeiten zwischen den Microservices führen würde und bei Entwicklung von Speicher-Routinen dazu noch keine API existierte, wurde darauf verzichtet.

Die vom Nutzer bewerteten Bücher oder Buch Bewertungen können nur angezeigt werden, wenn Recommendation und Book Microservices funktionieren und gültige Ergebnisse liefern, sonst wird ein Fehler an den Nutzer geleitet.

Für den Zugriff auf die Datenbank wurden EJBs genutzt, die mit Stateless-Annotationen versehen wurden um nur eine einzelne Instanz zu haben.

Alle erstellten Transaktionen sind atomar und benötigen daher keine Transaktionsbehandlung.

8.7 Sessionbehandlung

Sessions waren für die Aufgabenstellung keine Anforderung. Aus diesem Grund wurde dies zunächst nicht berücksichtigt und zu Beginn musste sich bei jedem Aufruf der Seite neu "eingeloggt" werden. Um für die Demos ein realistisches Szenario abzubilden, da zwischen den Webanwendungen der beiden Teams gewechselt wurde, hat sich das Team entschieden, eine rudimentäre Lösung zu implementieren, die es erlaubt, Anfragen über mehrere Seitenbesuche hinweg als die gleiche Person zu stellen, ohne sich jedes mal neu "einloggen" zu müssen.

Dazu speichert sich das Frontend nach dem "einloggen" die *CustomerId* in einem Cookie. Beim Laden der Anwendung versucht das Frontend zu Beginn, die *CustomerId* aus dem Cookie zu laden. Ist dieser nicht verfügbar, muss sich der Kunde "einloggen", andererseits wird der Nutzer auf die "Meine Bücher" Ansicht weitergeleitet.
Um die *CustomerId* auch der Frontend Anwendung von Team 1 zur Verfügung zu stellen, wird die *CustomerId* beim Navigieren per URL Parameter übergeben (solange dies über den entsprechenden Link in der Oberfläche erfolgt).
Wenn sich der Kunde über die Oberfläche von Team 1 einloggen möchte, passiert dies über einen Redirect zu dem Frontend von Team 2 mit einer Redirect URL. Dort wird wieder versucht, die *CustomerId* zu laden bzw. es wird sich "eingeloggt". Sobald eine *CustomerId* gesetzt ist, wird die übergebene URL genutzt, um den Nutzer wieder umzuleiten. Dabei wird, wie bereits beschrieben, die *CustomerId* als URL Parameter gesetzt und kann nun vom Frontend entsprechend ausgelesen werden.

Dies entspricht stark vereinfacht der gängigen Praxis. Der Login wird über eine zentrale Komponente gesteuert. Beim Einloggen wird ein Token erstellt und zurückgegeben, der von der Anwendung genutzt wird, um sich in Folge-Anfragen zu authentifizieren. In diesem Token sind die relevanten Informationen zur Identifikation enthalten, so dass jede Anfrage von einer beliebigen Instanz eines Microservices behandelt werden kann. In dem das Backend "stateless" gestaltet ist, muss sich nicht um eine komplexe Synchronisierung von Sitzungen zwischen den Systemen gekümmert werden, was eine wichtige Eigenschaft zum Skalieren eines Systems darstellt.

8.8 Validierung, Ausnahme- und Fehlerbehandlung

8.8.1 Validierung

Da die eigene UI keine freien Eingabemöglichkeiten bietet, können durch diese keine fehlerhaften Anfragen gesendet werden. In einer Microservice Architektur wird die Schnittstelle auch durch andere Teams benutzt und somit kann sich nicht darauf verlassen werden, valide Daten zu erhalten.

Alle festgelegten Business Regeln werden von den Anwendungen in der Control-Schicht (nach ECB) behandelt, in der die Anwendungslogik definiert ist (siehe Typische Muster, Strukturen und Abläufe).

In der Anwendung kommt es zu zwei Prüfungen von syntaktisch korrekten Parametern:

1. Die Wertung darf nur auf einer Skala von 1 bis 5 stattfinden.
2. Es dürfen nur maximal 20 Durchschnittsbewertungen abgefragt werden.

Im Falle von semantisch ungültigen Parametern wird eine *IllegalArgumentException* mit einer entsprechenden Fehlerbeschreibung geworfen.

Code Block 2 Beispiel einer Prüfung

```
if(rating < 1 || rating > 5) {
    throw new IllegalArgumentException("Rating has to be between 1 and
5");
}
```

Wie diese behandelt werden, wird im folgenden Abschnitt behandelt.

8.8.2 Ausnahme- und Fehlerbehandlung

Ist eine Exception nicht vom System zu behandeln, wird diese in Form von einer Beschreibung mit entsprechendem Fehlercode in der Response zurück gegeben.
Dazu wurden die in der Praxis gängigen HTTP Status Codes verwendet (vgl. http://www.restapitutorial.com/httpstatuscodes.html).

Code	Bezeichnung	Fall	Exception
400	Bad Request	• Wird bei ungültige Anfragen (Parametern oder Objekten) zurückgegeben	IllegalArgumentException
404	Not Found	• Wenn die angefragte Ressource nicht gefunden werden kann (z.B. GET */customers/51* und kein *Customer* mit der Id 51 existiert) • Kann auftreten, wenn eine URL nicht korrekt ist (z.b. GET */custoomer* ist kein existierender Endpunkt)	EntityNotFoundException
503	Service Unavailable	• Wird auf externe Dienste, wie andere Microservices, zugegriffen und diese sind nicht verfügbar	ProcessingException

Code	Bezeichnung	Fall	Exception
500	Internal Server Error	• Wird automatisch geworfen bei nicht behandelten Exceptions	-

Dazu werden Exceptions aufsteigen gelassen ("Exception bubbling") und schlussendlich in der Boundary-Schicht behandelt. Das Beispiel greift die geworfene Exception aus dem oberen Abschnitt über Validierung wieder auf.

Code Block 3 Handhabung einer IllegalArgumentException

```
@POST
@Path("books/{bookId}/rate/{rating}")
public Response rateBook(@PathParam("bookId") String bookId,
                         @PathParam("rating") int rating, UserDto userId)
{
    try {
        BookAverageRatingDto bookAverageRating =
ratingService.rateBook(userId.getUserId(), bookId, rating);
        return Response.status(201).entity(bookAverageRating).build();
    }
    catch(IllegalArgumentException e) {
        return Response.status(400).entity(new
ErrorDescription(e.getMessage())).build();
    }
}
```

Da jeder Request separat behandelt wird, kann das Gesamtsystem, trotz auftretender Exceptions, weitere Anfragen annehmen.
Das Excecption Handling in der Boundary-Schicht ist nach dem "Dont't repeat yourself"-Prinzip unschön implementiert (s. Risiken & technische Schulden).

8.9 Konfigurierbarkeit

8.9.1 Konfiguration des Projekts mittels Maven

Bei den entwickelten Microservices handelt es sich um Maven-Projekte. Das bedeutet, dass diese eine sogenannte Project Object Model-Datei (pom.xml) beinhalten, welche Informationen über das Projekt, Abhängigkeiten zu externen Bibliotheken und die Konfiguration des auszuführenden Maven Lifecycles enthält.

Das folgende Codelisting enthält eine vereinfachte Version der pom.xml-Datei des Recommendation Microservice. Zur besseren Übersicht wurden dort alle definierten Abhängigkeiten zu externen Bibliotheken entfernt. In den nachfolgenden Unterabschnitten wird jeweils kurz auf einzelne Aspekte dieses Codelistings eingegangen.

Code Block 4 Vereinfachte POM-Datei des Recommendation Microservice

```
<project ...>
    <modelVersion>4.0.0</modelVersion>
    <groupId>de.thkoeln</groupId>
    <artifactId>recommendation-ms</artifactId>
    <name>recommendation-ms</name>
    <version>0.1</version>
    <packaging>war</packaging>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <failOnMissingWebXml>false</failOnMissingWebXml>

        <app.name>recommendation-ms</app.name>
        <app.baseUrl>localhost:8080</app.baseUrl>

        <wildfly.username/>
        <wildfly.password/>
        <wildfly.hostname>localhost</wildfly.hostname>
        <wildfly.port>9990</wildfly.port>
        <wildfly.run.skip>true</wildfly.run.skip>
        <wildfly.java-opts/>

        <environment>dev</environment>
    </properties>

    <profiles>
        <profile>
            <id>env-local-run</id>

            <properties>
                <wildfly.run.skip>false</wildfly.run.skip>
                <wildfly.java-opts>-
agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
                </wildfly.java-opts>
            </properties>
        </profile>

        <profile>
            <id>env-local-redeploy</id>

            <properties>
                <wildfly.run.skip>true</wildfly.run.skip>
            </properties>
        </profile>

        <profile>
            <id>env-prod</id>
            <properties>
                <app.baseUrl>fsygs15.gm.fh-koeln.de:8280</app.baseUrl>
                <wildfly.username>team2management</wildfly.username>
                <wildfly.password>GPA_WS1617_Team2</wildfly.password>
                <wildfly.hostname>fsygs15.gm.fh-
koeln.de</wildfly.hostname>
                <wildfly.port>10190</wildfly.port>
                <wildfly.run.skip>true</wildfly.run.skip>

                <environment>prod</environment>
            </properties>
        </profile>
    </profiles>

    <build>
        <finalName>${app.name}</finalName>
```

```
        <plugins>
            <!-- Creates checkstyle-result.xml in target folder-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
                <version>2.16</version>
                <executions>
                    <execution>
                        <id>checkstyle</id>
                        <phase>verify</phase>
                        <goals>
                            <goal>check</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <!-- Creates a swagger.json in target with the concatenated
deployment domain-->
            <plugin>
                <groupId>com.sebastian-daschner</groupId>
                <artifactId>jaxrs-analyzer-maven-plugin</artifactId>
                <version>0.10</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>analyze-jaxrs</goal>
                        </goals>
                        <configuration>
                            <backend>swagger</backend>

<deployedDomain>${app.baseUrl}/${app.name}</deployedDomain>
                        </configuration>
                    </execution>
                </executions>
            </plugin>

            <!--Copy environment specific files-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-antrun-plugin</artifactId>
                <version>1.8</version>
                <executions>
                    <execution>
                        <id>copy environment specific files</id>
                        <phase>initialize</phase>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <configuration>
                            <tasks>
                                <echo>Copying beans.xml...</echo>
                                <copy
file="${basedir}/src/main/environment/${environment}/beans.xml"
tofile="${basedir}/src/main/webapp/WEB-INF/beans.xml" overwrite="true"/>
                                <echo>Copying persistence.xml...</echo>
                                <copy
file="${basedir}/src/main/environment/${environment}/persistence.xml"
tofile="${basedir}/src/main/resources/META-INF/persistence.xml"
                                    overwrite="true"/>
                            </tasks>
                        </configuration>
                    </execution>
                    <execution>
                        <id>copy swagger file</id>
```

```
                        <phase>prepare-package</phase>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <configuration>
                            <tasks>
                                <echo>Copying swagger.json...</echo>
                                <copy file="${basedir}/target/jaxrs-
analyzer/swagger.json" tofile="${basedir}/src/main/webapp/swagger.json"
overwrite="true"/>
                            </tasks>
                        </configuration>
                    </execution>
                </executions>
            </plugin>

            <!--Deployment on Wildfly-->
            <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-maven-plugin</artifactId>
                <version>1.1.0.Final</version>
                <configuration>
                    <username>${wildfly.username}</username>
                    <password>${wildfly.password}</password>
                    <hostname>${wildfly.hostname}</hostname>
                    <port>${wildfly.port}</port>
                    <name>${project.build.finalName}.war</name>
                    <java-opts>
                        <java-opt>${wildfly.java-opts}</java-opt>
                    </java-opts>
                </configuration>
                <executions>
                    <execution>
                        <id>Run Wildfly</id>
                        <phase>deploy</phase>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <configuration>
                            <skip>${wildfly.run.skip}</skip>
                        </configuration>
                    </execution>

                    <execution>
                        <id>Deploy on Wildfly</id>
                        <phase>deploy</phase>
                        <goals>
                            <goal>deploy-only</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

        </plugins>
    </build>

    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
                <version>2.16</version>
            </plugin>
        </plugins>
```

```
    </reporting>
</project>
```

8.9.1.1 Profile

Zur Konfiguration des Projekts unter verschiedenen Umgebungen werden Maven Profile verwendet. Es wurden dabei zwei Profile für die lokale Umgebung bei der Entwicklung und ein Profil für die Produktiv-Umgebung definiert, zu finden sind diese in den Zeilen 28 - 60 des Listings. Je nach Profil werden die in den Zeilen 10 - 26 definierten Properties auf bestimmte Werte gesetzt, welche Einfluss auf den Buildvorgang von Maven haben.

Nachfolgend werden die Profile kurz vorgestellt.

env-local-run: Um das Projekt lokal zu starten, muss das Profil "env-local-run" gesetzt werden. Durch dieses wird veranlasst, dass ein lokaler WildFly Server im Debug-Modus gestartet wird, so dass sich ein Debugger an den Prozess anhängen lässt, sofern dies gewünscht ist. Anschließend wird die Anwendung auf den gestarteten Server deployed.

env-local-redeploy: Statt einen lokalen WildFly Server zu starten, wird die Anwendung bei Verwendung dieses Profils auf einen laufenden Server deployed. Dies verhindert, dass der WildFly Server jedesmal neu gestartet werden muss.

env-prod: Die Verwendung dieses Profils konfiguriert das Projekt für das Deployment auf die Produktiv-Umgebung. Dafür werden die Zugangsdaten des WildFly Application Server durch Properties gesetzt. Es ist zu beachten, dass das Passwort des Servers dazu im Klartext in der POM-Datei steht, was bei einem "richtigen" Projekt nicht der Fall sein sollte.

8.9.1.2 Deployment

Zum Aufspielen der Anwendung auf einen WildFly Application Server wird das WildFly Maven Plugin verwendet. Dieses bietet die Möglichkeit, einen Application Server zu starten (und herunterzuladen, falls keiner vorhanden ist), zu stoppen, und Anwendungen auf einen Server zu deployen. Die Konfiguration dieses Plugins ist in den Zeilen 139 - 173 im Codelisting zu finden.

Durch Definition von "executions" wird die Ausführung von bestimmten "goals" (run, deploy) des Plugins in die "deploy"-Phase des Maven Lifecycles eingehängt. Durch die Property "wildfly.run.skip" kann durch die vorher beschriebenen Maven Profile festgelegt werden, ob ein Application Server gestartet werden soll.

8.9.1.3 Austausch von Dateien

Durch das Antrun Maven Plugin (Zeilen 101 - 136) werden bestimmte Dateien während der Ausführung des Maven Lifecycles in bestimmte Ordner der Projektstruktur kopiert. Dies betrifft die Dateien "beans.xml", "persistence.xml" und "swagger.json", die in den nachfolgenden Abschnitten beschrieben werden.

Die ersten beiden Dateien dienen dabei der Konfiguration des Projektes für verschiedene Umgebungen. Die Property "environment" wird dazu von dem ausgewählten Maven Profil gesetzt, um die für die Umgebung benötigten Konfigurationsdateien anhand des Dateipfads zu finden.

8.9.1.4 Checkstyle

Das Checkstyle Maven Plugin (Zeilen 67 - 80 & 178 - 186) führt eine Qualitätsanalyse des Programmcodes durch. Durch die "checkstyle.xml" im Projektordner wird dabei definiert, welche Prüfungen von Checkstyle durchlaufen werden sollen. Nach Fertigstellung der Analyse wird das Ergebnis zur späteren Einsicht in einer Datei ("checkstyle-result.xml") abgespeichert.

8.9.1.5 Generierung einer API Dokumentation

Bei Ausführung des Maven Lifecycles wird durch das "Jaxrs Analyzer" Maven Plugin (Zeilen 83-98) automatisch eine Swagger API Dokumentation (siehe Abschnitt API Dokumentation) generiert. Das Plugin erzeugt eine JSON-Datei, welche in den "webapp"-Ordner des Projektes kopiert wird und somit nach dem Deployment über einen Webbrowser aufgerufen werden kann.

8.9.2 Konfiguration von Jenkins

Im Projekt wurde Jenkins als Buildserver verwendet. Für die zwei entwickelten Microservices und die Weboberfläche ist jeweils ein Build Job in Jenkins vorhanden. Anstatt diese über die Weboberfläche zu konfigurieren, wurden diese allerdings über sogenannte Jenkinsfiles konfiguriert. Diese sind Skript-Dateien, die im Projektrepository liegen und eine beliebige Anzahl von Befehlen für Jenkins enthalten können. Der Vorteil, die Konfiguration der Build Jobs in Dateien auszulagern, liegt beispielsweise darin, dass diese somit durch Git versioniert sind und leicht für weitere Build Jobs verwendet werden können.

Im nachfolgenden Codelisting ist das Jenkinsfile für den Recommendation Microservice dargestellt.

Code Block 5 Vereinfachtes Jenkinsfile des Recommendation Microservice

```
node {
    def mvn = getMvnExePath()
    def mvnBaseArgs = "-f recommendation-ms/pom.xml -P env-prod"

    stage('Checkout') {
        git credentialsId: '23fc7e80-335b-4b69-a854-c90484488d77', url:
'https://fsygs15.gm.fh-koeln.de:8888/GPAWS1617TEAM2/recommendations-
ms.git'
    }
    stage('Build') {
        def version = version()

        if (version) {
            echo "Building version ${version}"
            currentBuild.displayName += " [v. ${version}]"
        }

        try {
            sh "${mvn} clean package -DskipTests ${mvnBaseArgs}"
        }
        catch (e) {
            currentBuild.result = 'FAILURE'
            sendMailToCommitter()
            throw e
        }
    }
    stage('Code Quality') {
        sh "${mvn} checkstyle:check ${mvnBaseArgs}"
    }
    stage('Unit Tests') {
        try {
            sh "${mvn} surefire:test ${mvnBaseArgs}"
        }
        catch (e) {
            currentBuild.result = 'FAILURE'
            sendMailToCommitter()
            throw e
        }
    }
    stage('Build & Test Results') {
        junit '**/target/surefire-reports/TEST-*.xml'

        archiveArtifacts artifacts: '**/target/*.war'
        archiveArtifacts artifacts: '**/target/checkstyle-result.xml'
    }
    stage('Deployment') {
```

```
        try {
            sh "${mvn} wildfly:deploy-only ${mvnBaseArgs}"
        }
        catch (e) {
            currentBuild.result = 'FAILURE'
            sendMailToCommitter()
            throw e
        }
    }
}

def version() {
    def matcher = readFile('recommendation-ms/pom.xml') =~
'<version>(.+)</version>'
    matcher ? matcher[0][1] : null
}

def getMvnExePath() {
    ...
}

def sendMailToCommitter() {
    def email

    email = sh(returnStdout: true, script: "git log -1 --
format='%ae'").trim()

    mail subject: "${env.JOB_NAME} (${env.BUILD_NUMBER}) failed",
            body: "It appears that ${env.BUILD_URL} is failing, somebody
should do something about that",
            to: "${email}",
            replyTo: "${email}",
            from: 'noreply@jenkins.de'
}
```

Wie sich z.B. in Zeile 5 erkennen lässt, können sogenannte "Stages" definiert werden, die die einzelnen Schritte der Jenkins-Pipeline enthalten. Beim entwickelten System sind 6 Stages definiert:

Checkout: Nachdem das Jenkinsfile von Jenkins aus dem Git-Repository geladen wurde, wird in der Checkout-Stage das restliche Projekt geladen.

Build: Die Versionsnummer des Projektes wird aus der POM-Datei ausgelesen und dem Namen des aktuellen Build Jobs hinzugefügt. Anschließend wird das Projekt mittels Maven gebaut, wobei die Tests vorerst ausgelassen werden.
Schlägt der Build-Prozess fehl, wird eine E-Mail an den Committer der Änderung gesendet und der Build Job abgebrochen.

Code Quality: Durch das Maven Checkstyle Plugin wird eine Qualitätsanalyse des Codes durchgeführt.

Unit Tests: Die Testausführung, die bei der vorherigen "Build"-Stage nicht durchgeführt wurde, wird in dieser Stage ausgeführt. Wie im Abschnitt Testbarkeit beschrieben, werden an dieser Stelle lediglich Unit Tests und keine Integration Tests ausgeführt.
Wieder gilt, dass der Vorgang bei Fehlschlag abgebrochen wird.

Build & Test Results: Die Ergebnisse der Checkstyle-Analyse und der Unit Tests werden zur späteren Einsicht persistent abgespeichert.

Deployment: Durch das Maven Wildfly Plugin wird die Anwendung auf den Produktiv-Server aufgespielt. Auch hier wird beim Fehlschlag abgebrochen und eine Mail an den letzten Committer gesendet.

Die E-Mail-Adresse des letzten Committers wird dabei mittels dem "log"-Befehl von Git bezogen (Zeile 68 des Codelistings).

8.9.3 Konfiguration der JPA

Die in den Microservices verwendete Java Persistence API wird jeweils durch eine "persistence.xml"-Datei konfiguriert (siehe Abschnitt Persistenz). Je nach Umgebung ist dabei eine bestimmte Datei zu verwenden. Die "persistence.xml" für die lokale Umgebung ist im folgenden Codelisting dargestellt.

Code Block 6 Persistence.xml für lokale Umgebung

```
<persistence ...>

    <persistence-unit name="recommendationDb" transaction-type="JTA">

<class>de.thkoeln.nozama.business.rating.entity.BookRating</class>

        <properties>
            <property name="javax.persistence.jdbc.driver"
value="org.h2.Driver"/>
            <property name="javax.persistence.jdbc.url"
value="jdbc:h2:mem:test;MODE=PostgreSQL;DB_CLOSE_DELAY=-1"/>
            <property name="javax.persistence.jdbc.user" value="sa"/>
            <property name="javax.persistence.jdbc.password" value=""/>

            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>

</persistence>
```

Bei der lokal verwendeten Datenbank handelt es sich um eine H2 In-Memory-Datenbank. Dies bedeutet, dass das Datenbankschema bei jedem Start der Anwendung neu erstellt wird und alle Daten nur während der Laufzeit bestehen, da sie im flüchtigen Speicher liegen.
H2 bietet die Möglichkeit, die Datenbank in einem Kompatibilitätsmodus laufen zu lassen. Durch Angabe von "MODE=PostgreSQL" (Zeile 11) wird der PostgreSQL Modus gewählt, damit sich die lokale In-Memory-Datenbank möglichst ähnlich zu der PostgreSQL Datenbank in der Produktiv-Umgebung verhält.
Der Parameter "hibernate.show_sql" (Zeile 15) bewirkt, sofern dieser auf "true" gesetzt ist, dass Hibernate während der Ausführung der Anwendung Logging-Ausgaben auf der Konsole ausführt. Dies ist vor allem in der lokalen Umgebung hilfreich, um die Datenbankabfragen zu überprüfen und Fehler- oder Sonderfälle zu finden.

Die Konfiguration für die Produktiv-Umgebung ist in der nachfolgenden "persistence.xml" zu sehen.

Code Block 7 Persistence.xml für Production-Umgebung

```
<persistence ...>

    <persistence-unit name="recommendationDb" transaction-type="JTA">
```

```
<class>de.thkoeln.nozama.business.rating.entity.BookRating</class>

        <jta-data-source>java:jboss/datasources/recommendationDS</jta-
data-source>

        <properties>
            <property name="javax.persistence.jdbc.driver"
value="org.postgresql.Driver"/>
            <property name="javax.persistence.jdbc.url"
value="jdbc:postgresql://localhost:5432/gpmsteam2_recommendations"/>
            <property name="javax.persistence.jdbc.user"
value="gpmsteam2"/>
            <property name="javax.persistence.jdbc.password"
value="gp_ms_team_2"/>

            <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQL94Dialect"/>
            <property name="hibernate.show_sql" value="false"/>
        </properties>
    </persistence-unit>

</persistence>
```

Anstatt einer In-Memory Datenbank wird dort die Verbindung zu der auf dem Produktiv-System befindlichen Datenbank konfiguriert. Wie auch bei der POM-Datei steht das Passwort der Datenbank dabei direkt in der Datei, was eigentlich verhindert werden sollte.

Durch Angabe des Parameters "jta-data-source" wird als Datenbankschnittstelle eine vorher im WildFly Application Server konfigurierte Datasource verwendet. In der lokalen Umgebung ist dies nicht notwendig, da ohne Angabe solch eines Parameters die Standard Datasource von H2 verwendet wird. In der Produktiv-Umgebung muss jedoch vorher einmalig eine JTA Datasource im Application Server erstellt werden.

8.9.4 Einsatz eines Stubs

Bei Unit Tests sollte nicht gegen andere Komponenten, geschweige denn gegen externe Services getestet werden. Deshalb wird in der lokalen Testumgebung statt gegen den Book Catalogue Microservice gegen einen Stub getestet, der bei Anfragen nach Büchern von der Java-Faker-Bibliothek generierte Daten zurückgibt (siehe Abschnitt Testbarkeit).

Doch auch bei Integration Tests kann die Stub-Implementierung verwendet werden. In der Tat sind die Microservice-Projekte sogar so konfiguriert, dass der Stub standardmäßig verwendet wird. Wird das Projekt durch Maven für die Produktiv-Umgebung gebaut, wird die "beans.xml"-Datei ausgetauscht. Wie im nachfolgenden Codelisting zu sehen, enthält diese nur den Eintrag "alternatives" (Zeilen 6 - 8), durch den im Code gekennzeichnete alternative Implementierungen verwendet werden. Werden diese Zeilen entfernt, würde die Stub Implementierung verwendet werden.

Code Block 8 Beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
```

```
       bean-discovery-mode="all">
    <alternatives>

<class>de.thkoeln.nozama.business.rating.entity.BooksRepositoryMicro</cla
ss>
    </alternatives>
</beans>
```

8.10 Migration

Zur eventuellen Datenmigration wird das Werkzeug Flyway verwendet. Da bisher keine Migration von Daten im Projekt notwendig gewesen ist, wird dieses Tool lediglich zur Erstellung von den benötigten Datenbanktabellen und der Befüllung dieser mit initialen Daten eingesetzt.

Dafür wurden SQL-Skripte erstellt, die von Flyway beim Start jedes Microservice ausgeführt werden, sofern sie noch nicht auf der aktuellen Datenbank ausgeführt wurden. Flyway nimmt dazu eine Versionierung der Datenbank anhand der generierten Tabelle "schema_version" vor. In dieser Tabelle wird jede Ausführung von SQL-Skripten durch Flyway dokumentiert. Die nachfolgende Tabelle zeigt den vereinfachten Inhalt der "schema_version"-Tabelle der Datenbank des Recommendation Microservice.

Version	Description	Script	Installed On	Success
1.0	Create BookRating	V1_0__Create_BookRating.sql	2017-02-08 10:46:21	True
1.1	Insert BookRatings	V1_1__Insert_BookRatings.sql	2017-02-08 10:46:22	True

Es lässt sich anhand dieser Daten folglich nachvollziehen, auf welcher Version sich die Datenbank befindet, welche Skripte wann ausgeführt wurden, um diese Version zu erreichen, und ob die Ausführung erfolgreich war.

Die Versionsnummern werden dabei aus dem Namen der ausgeführten SQL-Skripte bezogen. Die Benennung dieser erfolgt nach folgendem Schema:
"V<Versionsnummer>__<Skriptname>.sql". Der Skriptname ist dabei für Flyway irrelevant und dient nur der Verständlichkeit der Nutzer.

Das Skript "V1_0__Create_BookRating.sql" dient der Erstellung der BookRating-Tabelle, "V1_1__Insert_BookRatings.sql" befüllt diese Tabelle anschließend initial mit Daten. Diese Daten wurden zuvor mittels der Java-Faker-Bibliothek zufällig generiert und in ein SQL-Skript transformiert.
Nach Ausführung dieser Skripte beim Start des Microservices befindet sich die Datenbank auf Version 1.1, beim nächsten Start werden folglich keine Skripte mehr ausgeführt, es sei denn, es werden weitere Skripte zum Projekt hinzugefügt.

8.11 Testbarkeit

8.11.1 CI/CD Pipeline

Neben der Entwicklung des funktionellen Teils des Microservice-Systems wurde eine stabile Continuous Integration/Delivery-Pipeline realisiert. Diese Pipeline besteht aus mehreren Elementen, die garantieren, dass Änderungen am System regelmäßig automatisch getestet, integriert und ausgeliefert werden (siehe Abschnitt CI/CD-Pipeline). Zwei dieser Elemente

werden an dieser Stelle kurz beschrieben, da sie eine wichtige Rolle in Bezug auf die Testbarkeit des Systems spielen.

GitLab: Als Werkzeug zur Versionsverwaltung wurde GitLab verwendet. Jedes der entwickelten Microservices hat dabei ein eigenes Repository, womit diese jeweils für sich versioniert sind. Nach Fertigstellung einer neuen Funktionalität kann diese mittels Git in den jeweiligen Microservice integriert werden. Dank der Möglichkeit von GitLab, nach solch einem "Push" durch einen sogenannten Webhook beispielsweise einen "Build Job" eines Buildservers anzustoßen, kann eine Änderung anschließend automatisch in das Produktiv-System übernommen werden. In diesem Projekt wurde Jenkins, im Folgenden kurz vorgestellt, als Buildserver verwendet.

Jenkins: Damit Änderungen automatisch in das Produktiv-System integriert werden, wird Jenkins als Buildserver verwendet. Dieser wird durch den erwähnten Webhook von GitLab angestoßen und holt sich den aktuellen Stand des Projekts anschließend aus dem Git Repository. Mittels Maven wird das Projekt gebaut und alle vorhandenen Unit Tests ausgeführt. Nur wenn beide dieser Vorgänge ohne Fehler absolviert werden konnten, wird das Projekt in die Produktiv-Umgebung ausgeliefert.

8.11.2 Unit Tests

Die Funktionalität jeder einzelnen Komponente der Microservices wird mittels Unit Tests überprüft. Es wurde dabei nach dem Test Driven Development Konzept nach Kent Beck vorgegangen.
Dieses besteht aus folgenden Schritten:

1. Erstellung eines Tests, der einen bekannten Fehler überprüft oder vom vorhandenen Code nicht erfüllt wird.
2. Änderung des Programmcodes, so dass alle Tests erfüllt werden.
3. Refaktorisierung des Codes.
4. Wiederhole bis gewünschte Funktionalität erreicht.

Dieses Vorgehen gewährleistet, dass für jede neue Funktionalität oder entdeckten Fehlerfall Unit Tests vorhanden sind.

Mittels Bibliotheken wie AssertJ und Mockito wurde die Erstellung der Unit Tests erleichtert. Mockito ermöglicht beispielsweise, Fremdkomponenten der zu testenden Komponente durch Mock-Objekte zu ersetzen, wodurch diese tatsächlich vollkommen isoliert getestet werden können. Die Mock-Objekte dienen als Platzhalter für die eigentlichen Fremdkomponenten und implementieren deren Schnittstelle, wodurch alle Methodenaufrufe immer noch auf diesen aufgerufen werden können. Werden allerdings Rückgabewerte erwartet, müssen diese im Test vorher definiert werden.

Zusätzlich zur Verwendung von Mock-Objekten wurde ein "Stub"-Objekt verwendet, um Tests durchzuführen, die Daten von einem anderen Microservice benötigen. Dieses Objekt implementiert, ähnlich wie ein Mock, die Schnittstellen des eigentlichen Objekts, liefert allerdings realistische Daten bei Methodenaufrufen. Solch ein Stub wurde im Projekt verwendet, um Buchdaten zu erhalten, die eigentlich beim Book Catalogue Microservice angefragt werden müssten. Mithilfe der Java-Faker Bibliothek werden vom Stub-Objekt zufällige Bücher generiert, die von Tests verwendet werden können.

Wie bereits beschrieben, wird Jenkins als Buildserver eingesetzt. Mittels Maven werden alle vorhandenen Unit Tests automatisch von Jenkins ausgeführt, nachdem eine Änderung durch Git in das Projekt integriert wurde. Die Testergebnisse werden dabei von Jenkins gespeichert und können anschließend über die bereitgestellte Weboberfläche betrachtet werden.
Schlägt ein Test fehl, bricht Jenkins den Vorgang ab und sendet eine E-Mail an den "Committer" der fehlerhaften Änderung.

8.11.3 Integration Tests

Neben Unit Tests wurden ebenfalls Integration Tests erstellt, die das Zusammenwirken von mehreren Komponenten testen.
Dazu werden mittels eines Http-Clients Anfragen an die REST API des zu testenden Microservice gesendet und anschließend die korrekte Abarbeitung dieser Anfragen überprüft.

Diese Integration Tests werden allerdings nur in der lokalen Umgebung ausgeführt, da dort nicht mit Daten des Produktiv-Systems, sondern mit Testdaten gearbeitet werden kann.
Da solch eine Test-Umgebung nur lokal besteht, können diese Tests momentan nicht im Build-Vorgang von Jenkins ausgeführt werden. Dort fehlt die Test-Umgebung, die beispielsweise aus einer zweiten Datenbank oder einem weiteren Application Server bestehen könnte.

8.12 Buildmanagement

Für jedes von Team 2 entwickelte Subsystem, sprich Customer Microservice, Recommendation Microservice und Meine Bücher UI, wurde ein Repository in GitLab eingerichtet. Da diese Subsysteme kontinuierlich integriert und ausgeliefert wurden, wurde stets auf einem einzigen Git-Branch (master) gearbeitet.

Jegliche im Abschnitt Konfigurierbarkeit beschriebenen Konfigurationsdateien liegen dabei innerhalb der Projektstruktur der Subsysteme und damit ebenfalls im jeweiligen Git-Repository.

In dem nachfolgenden Abschnitt wird der Aufbau des Recommendation Microservices dargestellt und dabei beschrieben, an welchen Stellen sich die verschiedenen Konfigurationsdateien und Tests des Projekts befinden.

8.12.1 Struktur

Eine vereinfachte Struktur des Recommendation Microservice ist in der rechten Spalte dargestellt. Die nachfolgenden Abschnitte beschreiben diese kurz und erläutern die relevanten Aspekte.

8.12.1.1 Stammverzeichnis

Im Stammordner der Microservice-Projekte liegt jeweils eine "Project Object Model"-Datei (POM), da es sich um Maven-Projekte handelt. Diese wird zur Konfiguration des Maven Lifecycles und zur Angabe der Abhängigkeiten mit externen Bibliotheken verwendet.

Die ebenfalls in diesem Ordner liegende "checkstyle.xml" dient der Einrichtung der von Maven ausgeführten Qualitätsanalyse des Programmcodes.

Zusätzlich liegt dort das sogenannte Jenkinsfile, durch welches die Phasen des Jenkins Build Jobs des Microservices anhand eines Skripts eingerichtet werden können. Wie im Abschnitt Konfigurierbarkeit bereits beschrieben, liegt der Vorteil dabei darin, dass die Jenkinsfiles versioniert sind, da sie mit im Git-Repository liegen, und für potentielle weitere Build Jobs wiederverwendet werden können.

8.12.1.1.1 Src

Dieses Verzeichnis enthält die zwei Unterordner "main" und "test", die ähnlich zueinander aufgebaut sind. Im "main"-Order befindet sich die Implementierung des Microservices mit allen benötigten Konfigurationsdateien, im "test"-Verzeichnis die Unit Tests dieser Implementierung. Integration Tests befinden sich in einem separaten Projekt, da diese lediglich HTTP-Anfragen mittels einem HTTP-Client senden und damit unabhängig vom eigentlichen Microservice-Projekt sind (siehe Abschnitt Testbarkeit).

8.12.1.1.2 Main

Die für die Implementierung des Microservices relevanten Dateien liegen in diesem Verzeichnis. Im Folgenden wird die Struktur und die wichtigsten Dateien dieses kurz beschrieben.

Environment: Dateien, die vor dem Build-Vorgang von Maven ausgetauscht werden müssen, liegen im "environment"-Ordner. Diese sind dabei nach den zwei Umgebungen (lokal = "dev", Production = "prod") sortiert.

Nozama: Dieser Ordner enthält die eigentliche Implementierung des Microservices. Wie bereits in mehreren Abschnitten erläutert, besteht diese aus zwei Services: Rating Service und Recommendation Service. Die Implementierung dieser Services ist entsprechend in Ordner aufgeteilt, wobei jeder dieser Ordner die ECB-Struktur abbildet.
Die Klasse DbMigrator.java wird beim Start der Anwendung ausgeführt und veranlasst die Daten-Migration durch Flyway (siehe Abschnitt Migration).

Resources: Das Verzeichnis "migration" enthält die SQL-Skripte, die von Flyway beim Start der Anwendung ausgeführt werden (sofern die Datenbankversion nicht bereits auf diesem Stand ist). Im "META-INF"-Ordner liegt die "persistence.xml", die je nach gewählter Umgebung durch eine der Dateien im "environment"-Verzeichnis ausgetauscht wird.

Webapp: Die im Ordner "WEB-INF" liegende "beans.xml" wird ebenso wie die "persistence.xml"-Datei während des Build-Vorgangs ausgetauscht. Durch sie kann bestimmt werden, ob Buchdaten direkt vom Book Catalogue Microservice oder aus einem Stub bezogen werden sollen (siehe Abschnitte Testbarkeit und Konfigurierbarkeit).
Die "swagger.json" ist das Ergebnis der Generierung der API Dokumentation in der Swagger-Spezifikation durch ein Maven Plugin (siehe Abschnitt API Dokumentation).Recommendation Microservice

- src
    - main
        - environment

          dev
            - beans.xml
            - persistence.xml

          prod
            - beans.xml
            - persistence.xml
        - nozama

          rating
            - boundary
            - control
            - entity

          recommendation
            - boundary
            - control
            - entity

          DbMigrator.java
        - resources

          migration
            - V1_0__Create_BookRating.sql

- V1_1__Insert_BookRatings.sql

META-INF

- persistence.xml

- webapp

  WEB-INF

  - beans.xml

  swagger.json

-
  - test
    - ...
- checkstyle.xml
- Jenkinsfile
- pom.xml

8.13 CI/CD-Pipeline

Neben der Entwicklung der Microservice-Architektur war ebenfalls die Erstellung einer stabilen Continuous Integration und Delivery Pipeline Ziel des Projektes. Die nachfolgenden Abschnitte beschreiben die einzelnen Phasen der CI/CD-Pipeline von der Entwicklung bis zum Deployment.

8.13.1 Entwicklung

Bei der Entwicklung der Anwendung sind hinsichtlich der CI/CD-Pipeline zwei wesentliche Aspekte zu nennen: Die Projektkonfiguration durch Maven und die Erstellung von Unit Tests. Im Folgenden werden die relevanten Punkte dieser kurz beschrieben.

8.13.1.1 Maven

Wie bereits in Abschnitt Konfigurierbarkeit beschrieben, sind die entwickelten Microservices Maven-Projekte. Durch Einsatz mehrerer Plugins wird gewährleistet, dass die Projekte beim Durchlaufen des Maven Lifecycles automatisch für die gewählte Umgebung konfiguriert werden. So muss lediglich ein bestimmtes Maven Profil gewählt werden, um ein Projekt statt für die lokale Test-Umgebung für die Produktiv-Umgebung zu konfigurieren. Dieses Vorgehen ist der erste wichtige Schritt in Richtung Continuous Deployment, zieht allerdings auch Nachteile mit sich, wie in Abschnitt Risiken & technische Schulden beschrieben.

8.13.1.2 Tests

Zusätzlich zur Konfiguration der Projekte für verschiedene Umgebungen ist die Erstellung von Tests ebenfalls unabdingbar. Da während der Entwicklung nach dem Test Driven Development Prinzip (siehe Testbarkeit) vorgegangen wurde, sind für alle Komponenten Unit Tests vorhanden. Durch die später beschriebene automatische Ausführung dieser Tests wird garantiert, dass bei jeder durch einen Git-Push ins Git-Repository aufgenommenen Änderung die Funktionalität des restlichen Systems immer noch vorhanden ist.

8.13.2 Integration

Die Integration von Änderungen in das System erfolgt durch das Versionsverwaltungtool GitLab und den Build Server Jenkins.

8.13.2.1 Git

Ein verwendetes Werkzeug für die Continuous Integration ist GitLab. Jeder Entwickler hat seine Änderungen regelmäßig, mindestens einmal täglich, auf das entsprechende Git-Repository gepushed und diese somit in das System integriert. Da lediglich auf einem Branch gearbeitet wurde, wurde stets darauf geachtet, keinen unfertigen oder fehlerbehafteten Code in das System zu bringen.

Durch Integration des Programmcodes an sich ist der Integrations-Prozess jedoch noch nicht vollständig abgeschlossen. Damit eine Änderung als integriert gilt, muss das System erst erfolgreich gebaut und alle Tests durchlaufen werden. Dies geschieht durch den Build Server Jenkins. Dieser beinhaltet einen Build Job pro Subsystem, welcher jeweils durch einen Webhook von GitLab initiiert wird.

8.13.2.2 Jenkins

Die Schnittstelle zwischen Integration und Deployment ist Jenkins. Wie bereits beschrieben ist dort für jedes Subsystem ein Build Job vorhanden, welcher dieses baut, testet und deployed. Jeder dieser Build Jobs besteht aus sechs einzelnen sogenannten Stages, welche durch ein Jenkins-Pipeline-Skript definiert werden(siehe Abschnitt Konfigurierbarkeit). Die drei Phasen "Build", "Unit Tests" und "Deployment" sind dabei für die CI/CD-Pipeline besonders relevant.

Wie oben erwähnt, ist die Integration einer Änderung erst abgeschlossen, wenn das Projekt gebaut und die Tests durchlaufen wurden. Dies erfolgt von Jenkins in den Phasen "Build" und "Unit Tests". Schlägt der Build-Vorgang oder einer der Tests fehl, so wird der komplette Build Job abgebrochen und es wird eine E-Mail an den Entwickler, der die fehlerhafte Änderung in das System gebracht hat, gesendet.

Die Auswahl des entsprechenden Maven-Profils für die Produktiv-Umgebung beim Buid-Prozess sorgt dabei dafür, dass die Anwendung anschließend auf das Produktiv-System deployed werden kann.

8.13.3 Deployment

Die Stage "Deployment" im Build Job eines Projektes sorgt, wie der Name bereits vermuten lässt, für das Deploymet der Anwendung auf das Produktiv-System.

Dazu wird ein Maven Plugin verwendet, welches durch ein entsprechendes Profil so konfiguriert ist, dass die gebaute Anwendung auf den WildFly Application Server der Produktiv-Umgebung deployed wird. Es gibt dabei unter den Subsystemen keinen Unterschied, jedes der entwickelten Anwendungen wird auf den gleichen Application Server deployed.

8.13.3.1 Flyway

Durch das Datenmigrationstool Flyway besteht die Möglichkeit, automatisch Datenmigrationen in der Datenbank der Produktiv-Umgebung durchzuführen, sofern dies notwendig ist. Wie in den Abschnitten Konfigurierbarkeit und Migration beschrieben, können dazu SQL-Skripte zur Projektstruktur hinzugefügt werden, die beim Start der Anwendung automatisch ausgeführt werden, sollte die Version der Datenbank geringer sein als die Versionsnummer der Flyway-Skripte, welche dem Namen der Skripte entnommen werden kann.

8.14 API Dokumentation

Anstatt die API Dokumentation manuell zu erstellen und somit Information zu duplizieren, sowie potentiell Inkonsistenzen zwischen Code und Dokumentation zu haben, sollte dies automatisch geschehen.
Eine der bekanntesten Frameworks dazu ist Swagger. Dieses bietet ein umfangreiches Ökosystem an Werkzeugen und Generatoren um RESTful APIs zu designen, erstellen,

dokumentieren und konsumieren.
Dazu bietet Swagger eine Spezifikation zum Beschreiben der Endpunkte, mit Methoden, Parametern, Status Codes, Beschreibungen und mehr.
Diese Spezifikation kann beispielsweise im JSON Format erstellt werden.

Für das Team war das Ziel, die Dokumentation der API zu automatisieren, so dass die Teams die verschiedenen Endpunkte und die Interaktionsmöglichkeiten einsehen können:

- Welche HTTP-Methoden werden von welchem Endpunkt unterstützt?
- Welche Parameter werden akzeptiert?
- Welche JSON-Objekte werden erwartet?
- Wie sieht die Antwort aus?
- Welche Status-Codes sind zu erwarten?

Da eines der Projekt-Ziele war, möglichst ohne die Nutzung weiterer Bibliotheken oder Frameworks zu entwickeln (abgesehen von Java EE 7), sollte vermieden werden, die Spezifikation über Annotationen zu erstellen, auch wenn dies eine sehr feine Dokumentation, mit entsprechendem Mehraufwand, zulässt.

Code Block 9 Mögliche Nutzung von Swagger

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@ApiOperation(value = "Say Hello World", notes = "Anything Else?")
@ApiResponses(value = {@ApiResponse(code = 200, message = "OK"),
@ApiResponse(code = 500, message = "Something wrong in Server")})
public Response sayHello() {
 JsonObject value = Json.createObjectBuilder()
                      .add("message", "Hello World!")
                    .build();
 return Response.status(200).entity(value).build();
}
```

Statt dessen wurde der JAX-RS Analyzer (https://github.com/sdaschner/jaxrs-analyzer) über Maven eingebunden. Dieser analysiert den Bytecode der Anwendung und erstellt basierend auf den für JAX-RS genutzten Annotationen, Methoden Parametern und Rückgabewerten eine *swagger.json*.
Somit konnten die erstellten (REST-) Controller frei von zusätzlichen beschreibenden Metadaten gehalten werden.

Die *swagger.json* Dateien der Microservices können durch die Swagger UI ausgelesen werden und ermöglichen es, die API interaktiv zu erkunden. Neben einer einfachen Übersicht können so Anfragen an die API gestellt werden.
Anzumerken ist, dass solch eine interaktive API Dokumentation die Anfragen nicht gegen das Produktiv-System stellen sollte.

Microservice	swagger.json	Swagger UI
Customer MS	http://fsygs15.gm.fh-koeln.de:8280/customers-ms/swagger.json	Swagger UI: Customer MS
Recommendation MS	http://fsygs15.gm.fh-koeln.de:8280/recommendation-ms/swagger.json	Swagger UI: Recommendation MS

Die Swagger UI stellt eine separate Web-Anwendung dar. Diese gibt es bereits als *Java Web Archive* (*.WAR*) gepackt, so dass dieses lediglich deployed werden muss (siehe https://github.com/AdamBien/headlands/releases/tag/v0.0.2).

9 Entwurfsentscheidungen

9.1 Empfehlungssystem

Das umzusetztende System sollte die Möglichkeit bieten, einem Kunden bis zu drei Buchempfehlungen auszusprechen. Diese Aufgabe wird vom Recommendation Microservice, genauer gesagt dem Recommendation Service (siehe Abschnitt Bausteinsicht), übernommen.

Es wurde sich dafür entschieden, diese Empfehlungen anhand aller vorhandenen Buchbewertungen zu ermitteln.

9.1.1 Algorithmus

Die Berechnung der Ähnlichkeit zweier Kunden zueinander wird mithilfe der unten abgebildeten Kosinus-Ähnlichkeit durchgeführt.

$$sim(a,b) = cos(\theta) = \frac{a \bullet b}{||a|| * ||b||}$$

Die Variablen **a** und **b** stehen dabei für die Bewertungs-Vektoren zweier zu vergleichenden Kunden. Nehmen wir an, dass **a** der Kunde ist, für den Empfehlungen zu ermitteln sind und **b** ein zweiter Kunde, der mit **a** verglichen werden soll. Der Vektor des Kunden **a** enthält somit die Werte aller seiner Buchbewertungen. Damit die beiden Kunden verglichen werden können, muss Vektor **b** die gleiche Länge haben. Daher werden nur die Bücher in Vektor **b** mit einbezogen, die ebenfalls in Vektor **a** vorhanden sind. Ist eine Bewertung für ein Buch in Vektor **a** enthalten, das nicht von Kunde **b** bewertet wurde, so wird der Wert 0 übernommen.

Werden die Buchbewertungen beider Kunden nun mit der angegebenen Formel verglichen, so ergibt sich ein Wert zwischen 0 und 1, wobei 0 keine Ähnlichkeit und 1 eine komplette Übereinstimmung symbolisiert. Dieser Ergebniswert muss nun gespeichert werden und die Berechnung für alle anderen vorhandenen Kunden wiederholt werden.

Nachdem alle Kunden mit **a** verglichen wurden, kann der ähnlichste Kunde anhand der Ergebnisse ermittelt werden. Der Prozess ist jedoch noch nicht abgeschlossen, da das Ziel nicht ist, den ähnlichsten Nutzer zu finden, sondern Empfehlungen auszusprechen. Folglich werden die Bewertungen des ähnlichsten Nutzers, sortiert nach Höhe der Bewertung, betrachtet. Befindet sich darunter ein Buch, welches von Kunde **a** noch nicht bewertet wurde, so ist dies eine Empfehlung.

Konnten keine drei Empfehlungen vom ähnlichsten Kunden bezogen werden, so wird der nächstbeste Kunde betrachtet, bis entweder alle drei Empfehlungen vorhanden sind oder alle Kunden betrachtet wurden.

9.1.2 Alternativen

Anstatt die Buchbewertungen aller Kunden miteinander zu vergleichen und dadurch die Ähnlichkeiten dieser zu berechnen, können Empfehlungen auch anders ermittelt werden. So könnten beispielsweise die Produkte, sprich die Bücher, miteinander verglichen werden. Empfehlungen könnten dann durch den Vergleich bestimmter Merkmale, wie etwa das Genre oder der Autor von Büchern, ausgesprochen werden.

Im Szenario des Projektes ist dies allerdings nicht ohne weiteres möglich, zum einen da der Bücherkatalog z.B. nicht besonders viele Informationen über die Bücher beinhaltet, zum anderen wäre es mit Problemen verbunden, die benötigten Informationen zu beziehen, selbst wenn diese vorhanden wären.

Es gäbe dabei die Möglichkeiten, die benötigten Daten entweder aus dem Book Catalogue Microservice zu beziehen, und den Algorithmus somit zu verlangsamen oder fehleranfälliger zu machen oder diese direkt im Recommendation Microservice zu speichern, was leicht zu Dateninkonsistenz führen kann. Wie in Abschnitt Evaluation der verwendeten Technologien

angemerkt, ist dies durchaus ein komplexes Thema bei Microservice-Architekturen, welches bei diesem Projekt nicht behandelt wurde.

9.1.3 Probleme

Der Aspekt der Performance ist das größte Problem des eigentlich simplen Algorithmus. Bei dem momentanen Datenbestand von 50 Kunden und etwa 5000 Buchbewertungen ist die Berechnung aktuell noch ohne Wartezeit in Echtzeit durchführbar. Doch wird von einem größeren Datenbestand ausgegangen, so sollte eine Möglichkeit gefunden werden, das Empfehlungssystem zu optimieren.

Neben der Optimierung der benötigten Datenbankabfragen, der Aufbereitung und Filterung der Daten und der Berechnung der Ähnlichkeiten zwischen Nutzern wäre die Möglichkeit denkbar, die Berechnung nur zu bestimmten Zeiten oder nur nach Änderung einer Buchbewertung auszuführen und die Ähnlichkeitswerte der Kunden zueinander persistent in einer Datenbanktabelle abzuspeichern.

Wenn die Ähnlichkeiten der Kunden persistent vorliegen, kann auf Anfrage der Empfehlungen für einen Kunden schnell der ähnlichste Kunde ausgelesen werden. Anschließend müssen nur noch die drei Empfehlungen von diesem ermittelt werden.

9.2 Teilung der Microservices

Um die Empfehlungen zu erstellen braucht das Recommendation Microservice die Bewertungen von Nutzern.

Ohne die Bewertungen können keine Empfehlungen erstellt werden. Damit ist der Zusammenhangsart zwischen dem Rating MS und Recommendation MS aus der Sicht von Recommendation MS - **Gemeinsamer BC**, wobei aus Sicht von Rating MS **Separate Ways** vorliegt.

Bei getrennter Entwicklung von Rating MS und Recommendation MS würde man den Mehraufwand betreiben für die Entwicklung von Schnittstellen, die nur intern genutzt werden dürften.

Da Recommendation MS viele Daten vom Rating MS brauchen würde (s. Empfehlungen) und die Menge an Daten mit der Zeit wachsen würde, würde dies zu hohem Traffic zwischen den Beiden Microservices führen, was sich in Antwortzeiten wiederspiegeln würde.

Um den Traffic zu verringern könnte an in Recommendation MS eigenen Datenbank aufbauen, der im Endeffekt die daten aus Rating MS abbilden würde.

Aus diesen Gründen wurde entschieden beide Microservices zusammenzulegen, womit man den Schwerpunkt auf andere Aufgaben legen konnte und sich weniger um die Architektur kümmern müsste.

9.2.1 Projektstruktur

Ein Projekt kann funktional oder technisch getrennt werden.

Mit funktionaler Trennung werden die Abschnitte nach "Business Capability" getrennt (bspw. Bücher, Kunden, Bewertungen).

Bei technischer Trennung werden die Abschnitte nach technischer Zuständigkeit geteilt (bspw. Controller, Serivce, Data, Common).

Da im Projekt die Microservice Architektur gelebt wird, wird jedes Feature in eigenem Microservices entwickelt.

Damit ist eine technische Trennung obsolet.

Da Recommendation Microservice 2 Funktionalitäten abdeckt, wie im vorherigen Abschnitt beschrieben, werden beide gleichgestellt nach ECB aufgeteilt.

9.3 Dateninkonsistenz: Recommendation und Book Microservice

Wie in Evaluation der verwendeten Technologien angemerkt, wurde kein einheitlicher Umgang mit Dateninkonsistenzen definiert.
Dennoch wurde für den Fall, dass es zu Bewertungen kein Buch gibt ein Mechanismus implementiert:

Wenn bspw. basierend auf 10 Bewertungen die zugehörigen Bücher angefragt werden, aber nur 5 gefunden werden, wurde sich dazu entschieden, dass dem Nutzer lediglich die 5 Bücher und dazugehörige Bewertung angezeigt werden.
So kann der Dienst normal weiter genutzt werden kann. Es wurde sich gegen ein Platzhalter-Buch entschieden, da das Buch vermutlich dauerhaft nicht mehr verfügbar ist.

Dies hat ebenfalls Auswirkungen auf den Empfehlungs-Algorithmus, da dieser weiter die Buchbewertungen berücksichtigen würde. Kommt es zu dem Fall, dass ein nicht mehr existierendes Buch empfohlen wird, könnten nur 2 von 3 Büchern geladen werden und der Nutzer würde nur 2 Empfehlungen erhalten.

In einer fortgeführten Microservice Architektur, würde bspw. ein Event Bus oder Message Queue eingeführt. Der Book MS würde im Falle eines gelöschten Buches ein entsprechendes Event an den Event Bus senden. Alle Microservices für die dieses Event relevant ist könnten darauf reagieren. Im Falle des Recommendation MS könnten die Bewertungen zu dem gelöschten Buch entfernt werden.

10 Risiken & technische Schulden

Die Tabelle beschreibt Risiken die bereits aus aufgetretenen Problemen hervorgehen, sowie Aspekte der Anwendung, die das Team als nicht optimal empfindet.

Risiko/technische Schuld	Beschreibung	Mögliche Maßnahmen
Kein Change Management	Da kein Prozess installiert ist, wie mit Änderungen an der API umgegangen werden soll, besteht immer die Gefahr, dass bei mangelnder Absprache der Teams und durch die CI/CD Pipeline Änderungen bis ins Produktivsystem gelangen, die die Funktionsweise der anderen Microservices beeinflussen.	Einführen eines definierten Prozesses wie mit solchen Änderungen umgegangen wird. Wie werden solche Änderungen kommuniziert? Wie viel Zeit hat das andere Team um auf diese zu reagieren? Gibt es eine Migrationsphase? Wie können Änderungen gleichzeitig und ohne Downtime für den Nutzer deployed werden?
Keine Integrations-Tests in der CI/CD Pipeline	Unit Tests können nur eine gewisse Funktionsweise auf Teilebene gewährleisten. Für das Zusammenspiel der Komponenten werden daher Integrationstests genutzt. Zwar sind solche definiert worden, um Anfragen von der Boundary- bis zur Entitäts-Schicht zu behandeln. Zum jetzigen Zeitpunkt können diese nur lokal und manuell ausgeführt werden. Somit können nach wie vor Fehler durch die bspw. andere Umgebung (Server) auftreten.	Erstellung eines „Stages“ in der CI/CD Pipeline, in der die Anwendung in eine Testumgebung deployed wird, um die Integrationstest in dieser gegen Testdaten auszuführen. Erst nach dem erfolgreichen durchlaufen der Tests wird anschließend die Anwendung auf das Produktivsystem deployed. Eine andere Möglichkeit stellt der Austausch der Umgebungsrelevanten Daten dar, wie z.B. die *persistence.xml,* um anschließend den Traffic auf das ehemalige Test-System umzuleiten.
Maven-Skript für jeden Microservice	Zurzeit wird jeder Microservice durch ein eigenes Maven Skript erstellt. Wie in Konfigurierbarkeit	Es könnte eine POM-Vererbungshierarchie aufgebaut werden, so dass es ein Über-POM gibt, welches die Gemeinsamkeiten der Skripte vereint, so dass diese nur noch die individuellen Schritte definieren müssen. Hier müsste noch weiter die Machbarkeit evaluiert werden

Risiko/technische Schuld	Beschreibung	Mögliche Maßnahmen
	beschrieben, wird das Skript genutzt um entsprechend der Zielumgebung die Anwendung zu bauen. Für alle 3 Dienste des Teams gibt es Teile im Skript mit identische Konfigurationen, bis auf die Variablen zu Beginn. Somit gibt es einige duplizierte Abschnitte, was gegen das „Don't repeat yourself" Prinzip verstößt. Im Falle eines Bugs oder einer Änderung an einem Plugin in Form eines neuen Parameter, müsste in jedem Dienst das Skript entsprechend angepasst werden.	und eine Lösung für die Verwaltung der Über-POM und die Integration in andere Projekte gefunden werden.
Maven-Skript für jeden Microservice	Das Austauschen der *persistence.xml* oder *beans.xml* erfolgt über das Kopieren gesamter Ordnerinhalte in andere Ordner. Dies fühlt sich im Vergleich zu Lösungen anderer Frameworks, wie Spring mit einer Datei für jede Zielumgebung die entsprechend eines Parameters geladen wird, unnötig kompliziert und fehleranfällig an.	Evtl. Nutzung eines anderen Maven Plugins, dass Dateien umbenennen kann, so dass es eine *persistence.dev.xml* und *persistence.prod.xml* geben kann die nebeneinander liegen. Vor dem Bauen wird entsprechend der Zielumgebung die Version kopiert und als *persistence.xml* abgelegt.
Duplikationen in der Fehlerhandhabung in der Boundary-Schicht	Betrachtet man bspw. die *rating/boundary/RatingResource* Klasse im Recommendation MS, aus der Auszüge im Abschnitt	Es könnte ein Interceptor eingeführt werden, der jeden Aufruf eines Endpunktes mit einem try-catch einschließt. Somit kann die gesamte Fehlerbehandlung an eine Stelle verlegt werden. Die try-catchs könnten entfernt werden und somit wird bereits die Gefahr der Integer Status Codes beseitigt.

Risiko/technische Schuld	Beschreibung	Mögliche Maßnahmen

	Validierung, Ausnahme- und Fehlerbehandlung zu sehen sind, so fällt auf, dass sich die Fehlerbehandlung in allen Methoden gleicht.
	Dies verstößt gegen das "Dont't repeat yourself"-Prinzip.
	Des weiteren werden die HTTP-Status Codes manuell als Integer angegeben, wodurch es bspw. dazu kommt, dass eine *IllegalArgumentException* einmal ein "400 - Bad Request" und an anderer Stelle ein "404 - Not Found" zurück gibt.
	
	Durch diese Duplikation entstehen solche Fehler, bei Änderungen müssten alle Stellen angepasst werden und gleichzeitig verschlechtert sich durch die vielen try-catchs die Lesbarkeit.
		Dennoch könnten diese bspw. als Konstanzen oder in einem Enum definiert werden, so dass die Lesbarkeit erhöht wird.
		
		`//Vorher`
		`return`
		`Response.status(200).entity(...).build();`
		`return`
		`Response.status(201).entity(...).build();`
		`return`
		`Response.status(204).entity(...).build();`
		`return`
		`Response.status(400).entity(...).build();`
		`return`
		`Response.status(404).entity(...).build();`
		
		`//Nachher: 200er mit Konstanten oder 400er mit Code-Enum`
		`return`
		`Response.status(OK).entity(...).build();`
		`return`
		`Response.status(CREATED).entity(...).build();`
		`return`
		`Response.status(NO_CONTENT).entity(...).build();`
		`return`
		`Response.status(Code.BAD_REQUEST).entity(...).build();`
		`return`
		`Response.status(Code.NOT_FOUND).entity(...).build();`
		
		Des weiteren ist denkbar statische Hilfsmethoden anzubieten, die ein entsprechendes Response-Objekt erstellen:
		
		`public final class ResponseHelper {`
		
		`private static int OK = 200;`
		
		`public static Response Ok(Object entity){`
		`return`
		`Response.status(OK).entity(entity).build();`
		`}`
		`}`
		
		Mithilfe des Exception-Interceptors und dem ResponseHelper könnte ein Endpunkt nun wie folgt realisiert werden:
		
		`//Vorher`
		`@GET`
		`@Path("users/{userId}/books")`
		`public Response`
		`usersBooksWithRatings(@PathParam("userId")`

Risiko/technische Schuld	Beschreibung	Mögliche Maßnahmen

		long userId,
		
		@DefaultValue("1") @QueryParam("page") int page) {
		try {
		return Response.status(200).entity(ratingService.getUsersBooksWithRatings(userId, page)).build();
		}
		catch(IllegalArgumentException e) {
		return Response.status(400).entity(new ErrorDescription(e.getMessage())).build();
		}
		catch(ProcessingException e) {
		return Response.status(503).entity(new ErrorDescription(e.getMessage())).build();
		}
		catch(Exception e) {
		return Response.status(404).entity(new ErrorDescription(e.getMessage())).build();
		}
		}
		
		//Nachher
		@GET
		@Path("users/{userId}/books")
		public Response usersBooksWithRatings(@PathParam("userId") long userId,
		
		@DefaultValue("1") @QueryParam("page") int page) {
		
		return ResponseHelper.Ok(ratingService.getUsersBooksWithRatings(userId, page));
		
		}

11 Scrum im Hochschulumfeld

Ein weiteres Ziel des Projektes war es, dass die Studenten das Vorgehen nach Scrum erlernen und ihr zumeist theoretisches Wissen über die Methodik auch in der Praxis einsetzen. Ausgehend von den gesammelten Erfahrungen des Teams, sollen Rückschlüsse über die Eignung von Scrum für den Einsatz im Hochschulumfeld gezogen werden können.

Dazu werden im Folgenden die verschiedenen Phasen und Aspekte des Vorgehensmodells und wie diese für das Team funktioniert haben beschrieben.

11.1 Sprint Planung

Insgesamt wurden im Verlaufe des Projektes 4 Sprint Planungen durchgeführt. Die erste Planung fiel dem Team besonders schwer, da es noch keinerlei Erfahrungen mit der Technologie hatte und somit auch noch nicht bekannt war, was für Aspekte berücksichtigt werden müssen.

Es gab noch keine Grundstruktur für das Projekt und es war bspw. nicht genau abzusehen, wie viel Aufwand betrieben werden muss, um einfache Anfragen an die API zu behandeln.

Aus diesem Grund ist die Sprint Planung auch stark verwoben mit der Retrospektive, da in dieser betrachtet wird, was gut funktioniert hat und beibehalten werden sollte, während für Dinge die dem Team Probleme bereiten, konkrete Lösungen gefunden werden.
Durch die Erkenntnisse der ersten Retrospektive konnte das Team für den zweiten Sprint die Aufgabenbeschreibungen besser formulieren, um besser mit Erwartungshaltungen umzugehen und ein besseres paralleles arbeiten zu ermöglichen.

Durch das regelmäßige Wiederholen der Planungstätigkeit fiel es dem Team immer leichter, Probleme besser in seine Bestandteile aufzubrechen, wodurch gegen Ende gut die Machbarkeit von Aufgaben für einen Sprint eingeschätzt werden konnte.

Das kurzfristige Planen für lediglich 3 Arbeitstage setzte voraus, dass man sich für diese Zeit ein klares Ziel setzt und rückblickend hat dies geholfen, Aufgaben besser zu überblicken, da man fokussierter darauf hingearbeitet hat.

11.2 Sprint

Der Sprint selber wurde von einem dreiköpfigen Team durchgeführt. Dabei wurden die Rollen des Scrum Masters und Product Owners durch zwei der Teammitglieder eingenommen.
Dies ist bereits entgegen der von Scrum vorgeschlagenen Entwickler-Teamgröße von 6±3, da es nur ein einziges "einfaches" Teammitglied gab.

Die Problematik liegt der Ansicht des Teams darin, dass es sich bei der Teamgröße von 3 Mitgliedern ausreichend selbst regulieren kann und keine explizite Rolleneinteilung benötigt wird. Die Rollen wurden aus diesem Grund eher als Belastung wahrgenommen, da so eine Verpflichtung für Tätigkeiten ins Projekt gebracht wurde, die im Rahmen des Projektes schwer einzuhalten sind.

Im Vergleich konnte die Scrum Master Rolle besser wahrgenommen werden, als die des Product Owners. Der Scrum Master hat so etwa verstärkt auf die Einhaltung der Rahmenbedingungen geachtet, auch wenn allen Mitgliedern diese gleichermaßen bewusst waren.

Hingegen hat der Product Owner lediglich verstärkt auf die Pflege der Product und Sprint Backlog Items geachtet. Die Rolle des Bindeglieds zu den Stakeholdern und als bestimmende Instanz in Fragen des Backlogs konnte nicht zufriedenstellend eingenommen werden, da das gesamte Team den gleichen Zugang zu den Stakeholdern und Anforderungen hatte. Ebenso sind die Teammitglieder gleichgestellt und daher fällt es schwer in Entscheidungsfragen während der Planung plötzlich auf Funktionalitäten zu bestehen.

Diese Doppelbesetzung ist besonders schwierig für den Product Owner, da die meisten Konflikte bzgl. "Was kommt in den nächsten Sprint?" zwischen diesem und den Entwicklern entstehen. Aber wie soll sich der Product Owner verhalten, wenn er die meiste Zeit als normales Teammitglied fungiert?

11.3 Daily Scrum

Der Daily Scrum ist im Falle des einzelnen Arbeitstages in der Woche eher als Weekly Scrum zu bezeichnen. In diesem wurden die erledigten oder noch offenen Aufgaben aus der letzten Woche besprochen.

Eine wichtige Erkenntnis die Teams aus den Daily Scrums gewinnen sollen ist, an welchen Stellen es zurzeit Probleme gibt und wie diese den erfolgreichen Abschluss des Sprints beeinträchtigen können. Somit wird sich regelmäßig einen Überblick über den Fortschritt verschafft.

Durch die längere Pause zwischen jedem Treffen, fiel es schwer, aufgetretene Probleme vollständig wiederzugeben und somit konnten die anderen Teammitglieder schlechter auf diese eingehen.

Dennoch haben die Daily Scrums dabei geholfen, den jeweiligen Tag besser zu strukturieren. In dem Teammitglieder ihre Ziele für den Tag festlegen, fällt es leichter, die Arbeit zu synchronisieren und im Falle von Überschneidungen diese gezielt aufzulösen.

11.4 Scrum of Scrums

Diese fanden immer am ersten und letzten Arbeitstag des Sprints statt. Sie sollen zur Koordinierung der Aufgaben der beiden Teams dienen.
Die Sitzungen wurden von den Teams dazu genutzt, die anstehenden Aufgaben und erledigten Aufgaben kurz vorzustellen. Dadurch dass die Teams bereits in den Review Meetings einen Ausblick auf den folgenden Sprint gaben, hielt sich der Nutzen des ersten Scrum of Scrums in Grenzen, da zu diesem Zeitpunkt kaum neue Erkenntnisse vorhanden waren.

Das Meeting am letzten Arbeitstag hingegen schien zu spät stattzufinden, um gegebenenfalls auf Veränderungen einzugehen, oder um auf Probleme hinzuweisen, die sich durch geplante Änderungen noch ergeben werden.

Was sich im Nachhinein als problematisch herausstellte war, dass die Teams sehr fokussiert auf ihre Aufgaben waren und die Scrum of Scrums nicht als Gelegenheit wahr nahmen, das gemeinsame Produkt zu besprechen. Die Teams hätten viel fokussierter Aufgaben vorstellen müssen, die potentiell das andere Team betreffen können.

Davon ausgehend hätte ein gemeinsamer Prozess geschaffen werden können, der einen Rahmen für die Zusammenarbeit beider Teams darstellt.

11.5 Sprint Review

Das Ziel von Sprint Reviews ist den Stakeholdern das Software-Inkrement zu präsentieren. Auf Basis des Feedbacks und den Vorstellungen, wie der Wert der Anwendung als nächstes gesteigert werden kann, findet die nächste Planung statt.

Durch die stark vordefinierte Aufgabenstellung mit den vorgegeben Funktionalitäten, kommt dieser Aspekt kaum vor. Darüberhinaus stellen Aufgabe und die umgesetzte Funktionalität nur Mittel zum Zweck dar, um sich mit dem Architektur-Ansatz und den Technologien zu beschäftigen. Aus diesem Grund war es schwierig für das Team, trotzdem eine Situation wie vor einem Kunden zu simulieren.

Dies wurde zusätzlich dadurch verstärkt, dass Herr Bente als Dozent und Kunde eine Doppelrolle einnimmt. Dadurch hat er bereits ein anderes Interesse als ein üblicher Kunde und

dem Team fiel es nicht immer leicht, Themen für das Review auszuwählen, da dieses einerseits dem Kunden, aber auch dem Dozenten dienen sollte.

11.6 Retrospektive

Nach Abschluss eines Sprints und dem Abhalten des Sprint Reviews, hat das Team reflektiert, was am Prozess in den letzten Arbeitstagen gut funktioniert hat und was verbesserungswürdig ist.
Dadurch konnte, wie bereits im Abschnitt über die Sprint Planung erwähnt, die Zusammenarbeit im Team stark verbessert werden.

Es wurden häufigere Review-Sitzungen eingeführt, in denen die Mitglieder sich gegenseitig Lösungen von Aufgaben präsentierten. Dadurch erhielten alle Mitglieder einen besseren Überblick über die Gesamtanwendung und konnten sich besser in Besprechungen einbringen. Durch das Wissen über andere Aspekte in der Anwendung konnten die Mitglieder sich besser in die Planung einbringen. Somit konnten Aufgaben besser beschrieben werden, was dabei half Abhängigkeiten zwischen Aufgaben aufzudecken und zu beseitigen.
Somit konnten die Reibungen innerhalb des Teams aufgrund schlechter Aufgabeneinteilung weitestgehend beseitigt werden, was sich in einer sehr positiven zweiten Sprint Retrospektive zeigte.

Schwierigkeiten bereitet die wirkliche Einhaltung der aufgestellten Regeln und das bewusste Ausüben von geforderten Tätigkeiten, da diese gegen die Gewohnheit gehen. Das Bewusstmachen von Prozessverlusten und möglichen Lösungen für diese, stellte für das Team eine wichtige Tätigkeit dar, um erfolgreich im Team zusammen zu arbeiten.

Innerhalb eines Projektes auf die Arbeitsweise zu schauen, um noch während der Durchführung einer Teamarbeit Verbesserungen an der Zusammenarbeit durchzuführen, war bis dahin etwas neues für eine Teamarbeit im Hochschulumfeld.
Das Team hat somit seine Effektivität, wie auch die soziale Struktur, aktiv verbessern können, was positiv zum Gesamtergebnis des Moduls beigetragen hat.

11.7 Abschluss

Es lässt sich festhalten, dass Scrum auch im Hochschulumfeld funktioniert, auch wenn manche Aspekte aufgesetzter und gezwungener anmuten als andere. Teilweise ist dies der kleinen Teamgröße geschuldet, da es dem Team noch möglich ist, sich selbst zu strukturieren. Das Hauptproblem liegt aber in der zu geringen gemeinsamen Arbeitszeit bzw. den viele Pausen. Dennoch hilft das Vorgehen, feste Strukturen in die, sonst meist eher ungeordnete, Teamarbeit zu bringen. Dennoch überwiegen die Vorteile und die Möglichkeit für die Studenten die ersten praktischen Erfahrungen mit einem in der Praxis üblichen Vorgehen zu sammeln.

12 Evaluation der verwendeten Technologien

Als Abschluss der Dokumentation sollen nochmal auf zwei der Ziele dieses Guided Projects eingegangen werden.
Es sollten Erkenntnisse über den Microservice Architektur-Ansatz gewonnen werden und eine Technologie-Benchmarking von Java EE7 und OASP durchgeführt werden.

12.1 Microservices

Durch eine für die Zeit umfangreiche Aufgabenstellung hat das Team einen ersten funktionierenden Prototypen erstellen können der eine Microservice-Grundstruktur aufweist, mit mehreren Diensten die sich auf jeweils eine "Business Capability" konzentrieren. Viele Herausforderungen in der Praxis die sich aus diesem Ansatz ergeben konnten jedoch nicht im Rahmen des Projektes behandelt werden.

In diesem Abschnitt sollen Aspekte gennant werden, die für zukünftige Projekte relevant sein könnten und welche Probleme es mit dem Architekturansatz im Hochschulumfeld gibt. Als Abschluss sollen, die konkreteren Lernerfolge des Projektes für das Team vorgestellt werden.

12.1.1 Ungelöste Herausforderungen

12.1.1.1 UI-Integration

Im Rahmen des Projektes wurden von beiden Teams einfache unabhängige Singe Page Applications mit AngularJs erstellt. Die Entwicklung der UI sollte auch keine große Bedeutung zu kommen, da die Funktionen der Dienste im Hintergrund im Fokus standen.

Dennoch ist eines der größeren Probleme im Microservice Umfeld die Erstellung eines User Interfaces. Hat man ein Frontend, dass mit den verschiedenen Microservices kommuniziert? Oder entwickeln die verschiedenen Teams Komponenten, die dann zusammengeführt werden müssen.

In Variante eins entsteht so eine monolithische Frontend-Anwendung in denen alle Teams an einer Anwendung entwickeln und man so die gleichen Probleme erzeugt, die Microservices letztendlich lösen sollen.

Die zweite Variante hingegen folgt der Microservices-Idee, so dass jedes Team unabhängig bleibt. Für solch ein UI-Composition-Pattern hätte aber entsprechend Arbeit in einen weiteren Dienst investiert werden müssen, der sich um die Komposition einer einheitlichen Anwendung, bestehend aus vielen unabhängigen Komponenten kümmert. Eine Herausforderung ist dabei der Umgang mit Bibliotheken und das erstellen eines konsistenten Stylings über alle Komponenten hinweg.

12.1.1.2 Logging & Monitoring

Ein wichtiger Aspekt von Microservices ist, dass jedes Team unabhängig seinen Dienst entwickelt und verbessert. Dazu wird eine entsprechende Strategie benötigt, mit der beispielsweise das Nutzungsverhalten analysiert werden kann, um basierend darauf weitere Verbesserungen vorzunehmen ("develop, deploy, evaluate, improve"-Schleife).

Dies könnte im Rahmen des Projektes bspw. das Bewertungsverhalten der Nutzer sein. Werden Bewertungen gemacht? Wenn nicht müssten UI-Anpassungen gemacht werden, damit dies vielleicht leichter fällt. Anschließend wird wieder betrachtet, ob die Änderungen den erwarteten Erfolg gebracht haben.

In diesem Zusammenhang müsste ein System geschaffen werden, in dem wichtige Daten der verschiedenen Dienste migriert werden, damit auch Abhängigkeiten erkannt werden können.

Im Rahmen dieser Verbesserungs-Schleife sind auch Konzepte wie A/B-Testing und deren Umsetzung interessant.

12.1.1.3 Dateninkonsistenz

Da jeder Microservice seine eigene Datenbank besitzt und die Daten im für den Dienst benötigten Format hält, kann es neben nicht vermeidbaren Beziehungen auch Duplikate geben. Dies impliziert, dass von einer "eventual consistency" auszugehen ist.
So gibt es neue Fälle die man behandeln muss, da zB. in Beziehung stehende Objekte nicht über Kaskadierung ebenfalls gelöscht werden können. Eine weitere Möglichkeit die den Entwicklern genommen wird, ist es einfach Transaktionen auszuführen, wodurch mehrere Änderungen nur als Gesamtes stattfinden können. Werden mehrere Microservices genutzt, muss ein Lösung gefunden werden, wie zB damit umgegangen wird, sollte die Anfrage bei einem Microservice fehlschlagen.

12.1.1.4 Rapid provisioning & deployment

Einen wichtigen Vorteil den Microservices bringen sollen ist die Skalierbarkeit der Anwendung. Dazu wird, am besten voll-automatisiert, entsprechend der Last die Anwendung auf weiter Server deployed. Diese müssen dazu zunächst verfügbar gemacht werde, ins Cluster aufgenommen werden und wenn vorhanden dem Load-Balancer hinzugefügt werden. Dies war durch die bereitgestellte Infrastruktur und den fehlenden Einsatzzweck für die Anwendung nicht realisierbar.

Das Team hat in diesem Fall nur Erfahrung mit dem automatisierten Deployment nach Änderungen sammeln können.

12.1.1.5 Versionierung & Change Management

Die Teams haben zwar in den Pfad Ihrer Anwendungen eine Versionsnummer aufgenommen, aber diese letztendlich nicht genutzt. Aus diesem Grund wurden auch keine Erfahrungen gesammelt, wie man im Falle von Versionssprüngen durch funktionale Änderungen vorgeht.

Hier hätte ein Prozess installiert werden müssen, in dem sich die beiden Teams auf ein Vorgehen einigen, wie mit Änderungen umgegangen werden soll.

Wie werden Änderungen kommuniziert? Wie lange hat man Zeit, um den eigenen Dienst entsprechend anzupassen? Bei einem Versionssprung: Wie lange wird die aktuell genutzte Version noch unterstützt?

12.1.1.6 Sicherheit

Ebenfalls für das Projekt, aber nicht für die Praxis, irrelevant ist die Umsetzung von Sicherheitskonzepten:

Wie integriert man in verschiedene unabhängige Dienste ein einheitliches Sicherheitskonzept mit unterschiedlichen Rollen?
Wie können sich Dienste untereinander authentifizieren (Machine-to-machine)?
Werden Mechanismen zur Prüfung der Datenintegrität benötigt?

Dies sind nur einige Beispiele von Fragen, die eine umfassende Microservice-Architektur beantworten muss.

12.1.2 Probleme

Das Problem bei der Realisierung einer Microservice-Architektur im Hochschulumfeld, stellt die fehlende Möglichkeit dar, direkt mit der Infrastruktur zu arbeiten.
Der Grundgedanke von DevOps ist: "You build it, you run it". Leider war dies im Projekt somit nicht der Fall, auch wenn die Betreuer einen nach bester Möglichkeit unterstützt haben.

So konnten beispielsweise keine Erfahrungen mit dem Betrieb der Anwendung gesammelt werden und Aspekte wie die vollständige Automatisierung der Infrastruktur durch Skripte, war so nicht möglich.

12.1.3 Gesammelte Erfahrungen

Das Team konnte durch das Projekt erste Erfahrungen mit der Erstellung von polyglotten Systemen sammeln. Für die Teilnehmer war es neu, ein Gesamtsystem bestehend aus mehreren unabhängigen Systemen mit unterschiedlichen Technologie-Stacks zu entwickeln.

Wie im vorherigen Abschnitt angemerkt, wurde die Thematik der Microservices nur recht oberflächlich betrachtet, da es in der kürze des Projektes nur möglich war eine Grundstruktur für die Entwicklung eine Microservices mit Java EE7 zu schaffen, die automatisch durch Jenkins deployed wird.

Nimmt man die benötigten Grundvorraussetzungen nach Martin Fowler (https://www.martinfowler.com/bliki/MicroservicePrerequisites.html) hinzu, fehlen 2 von 3 der Vorraussetzungen, die es für ihn und einige Experten erst ermöglichen eine wirkliche Microservice-Architektur umzusetzen.

Für zukünftige Projekte wäre es daher wünschenswert, wenn sich fokussierter mit Teilbereichen beschäftigt wird, mit dem Ziel eine praxisnahe Lösung zu schaffen, um einen wirklichen Eindruck über die Komplexität und die tatsächlichen Herausforderungen gewinnen zu können.

Wie sich gezeigt hat herrscht ein ungleicher Wissenstand über die Thematik. Zwar hat das Projekt sich grundlegend damit beschäftigt, aber es ist fraglich, ob allen auf eine ähnliche Art die Herausforderungen bewusst sind. Daher wären vereinzelte Workshops, mit über das Projekt hinaus gehenden Information zu der Thematik vorstellbar, die dabei helfen ein breiteres Verständnis bei allen Teilnehmern zu schaffen.

Welchen Aspekt das Projekt gut simulieren konnte, war dass Teams unabhängig von einander Funktionalitäten umsetzen konnten, ohne sich zunächst in die Quere zu kommen. Spannend wurde es sobald es um die Integration der Dienste ging und erste Abhängigkeiten herrschten. Da dies auch zu einem zwischenzeitlichen Konflikt zwischen den Teams führte wurde verdeutlicht, dass es sich bei Microservicen nicht nur um ein Entwurfsmuster handelt, sondern auch die organisatorischen Strukturen entsprechend ausgebildet sein müssen.
Ohne einen entsprechenden Prozess und Austausch zwischen den Teams, kann es ebenso zu Problemen bei der Entwicklung kommen, wie es der Fall in einer Monolithischen Anwendung der Fall ist.

Dennoch konnten einige Vorteile solch einer Architektur verdeutlicht werden:
Hohe Entkopplung des Systems. Einfache Erweiterbarkeit. Es ist einfacher Probleme isoliert zu betrachten. Änderungen in einem kleinen System haben meist weniger Einfluss auf andere Komponenten.

12.2 Java EE7 vs. OASP

Das Projekt sollte dem Projektpartner Capgemini dazu dienen, weitere Erkenntnisse über die technologische Eignung der eingesetzten Technologien für den Microservice-Ansatz zu gewinnen.
Als Entwickler von OASP ist darüberhinaus interessant, ob die Architektur und Struktur die vorgegeben wird für eine Microservice Architektur geeignet ist.

An dieser Stelle können nur allgemeine Eindrücke festgehalten werden, da das Team nur mit Java EE 7 gearbeitet hat und keinerlei praktische Erfahrung mit OASP gesammelt wurde. Die Eindrücke über OASP stammen aus der ersten kurzen Einführung, sowie den Anmerkungen des anderen Teams in den Sprint Reviews.

12.2.1 Eindrücke Java EE 7

Java EE 7 hat sich als unkompliziertes Framework zum erstellen von Webservices herausgestellt. Die Herausforderung zu Beginn bestand darin eine grundlegende Struktur für das Projekt zu definieren (s. Typische Muster, Strukturen und Abläufe) nach der die Anwendung aufgeteilt wird.

Darüberhinaus war zu Beginn nicht direkt offensichtlich wie die Java Persistence API (JPA) funktioniert, da es im Falle der JPA sich lediglich um einen Standard handelt für den Java EE 7 keine Referenzimplementation mit bringt.

Das Team hätte allerdings beim Aspekt der Austauschbarkeit ein einfacheres System erwartet. So lässt sich zwar per Context and Dependcy Injection (CDI) und der *beans.xml* eine Konfiguration bestimmen, welche Klassen und Alternativ-Implementation wie Stubs oder Mocks genutzt werden sollen, allerdings gibt es keinen einfachen Weg die *beans.xml* für spezielle Konfiguration zu nutzen.

So kann im Vergleich in Spring eine *properties.dev* und *properties.prod* Datei definiert werden, die entsprechend einem beim Start übergebenen Parameters genutzt wird.
Im Falle von Java EE 7 hat das Team ebenfalls zwei *beans.xml* definiert, diese mussten allerdings durch Maven in das entsprechende Verzeichnis kopiert werden, bevor die Anwendung gebaut wird (vgl. Konfigurierbarkeit).

12.2.2 Vergleich zu OASP

OASP welches mitunter Spring nutzt, löst bereits solche typischen Herausforderungen für den Entwickler. Das Problem, dass das Team im Vergleich sieht, ist die zu Beginn größere Hürde das Zusammenspiel der unterschiedlichen Komponenten zu verstehen.
Es werden unterschiedliche Bibliotheken mit ihren spezifischen Eigenheiten genutzt und es ist eine spezielle Struktur vorgesehen an die man sich halten sollte.
Da diese Entscheidungen von dem Team bei der Entwicklung mit Java EE 7 selbst getroffen werden müssen, entsteht hier zwar ein zusätzlicher Aufwand, aber dafür werden diese Entscheidungen bewusst getroffen und somit ist das Verständnis ein besseres, da keine fremden Muster verstanden werden müssen.
Ebenso konnten durch dieses Bewusstsein Probleme leichter zurückverfolgt werden. Als Beispiel ist der bereits integrierte Logger zu nennen, der im OASP Team von Beginn an falsch konfiguriert war. Herauszufinden welche der unterschiedlichen vorkonfigurierten Komponenten im Projekt den Fehler verursacht ist so eine größere Herausforderungen. Zwar hat das Java EE 7 Team keinen Logger integriert, aber hätte es beim hinzufügen des Loggers Probleme gegeben, so dass die Anwendung nicht mehr gebaut werden kann, dann ist der Rückschluss, dass es am gerade hinzugefügten Logger liegt, um einiges einfacher.

Alles in allem würde fühlt sich Java EE 7 für das Team, basierend auf den genannten Erkenntnissen, im Vergleich als Bibliothek und OASP als Framework an.
OASP löst viele Probleme für den Entwickler und gibt einen stärkeren Rahmen vor dem der Entwickler folgt. Somit kann sich stärker auf die eigentliche Entwicklung konzentriert werden.
Mit Java EE 7 hat man als Entwickler im Vergleich mehr Freiheiten und kann die Anwendung selber strukturieren.
Aus diesem Grund spielen Anforderungen wie "Time-to-market" oder auch wie generisch das Problem ist bei der Auswahl ein Rolle. Dennoch müssten die beiden Technologien anhand genauer Eigenschaften verglichen werden, um genaue Aussagen über ihre Eignung treffen zu können, was im Rahmen des Projektes nicht möglich war.

13 Glossary

A

- BookMär 23, 2017 (13:12)
- ...

1 TermB

- Context and Dependency InjectionMär 24, 2017 (14:27)
- Continuous DeliveryMär 24, 2017 (14:32)
- ...

4 TermsC

- Definition of DoneOkt 10, 2016 (22:23)
- ...

1 TermD

- Entity-Control-BoundaryMär 23, 2017 (13:20)
- ...

1 TermE F G H I J K L M N O P Q

- RatingMär 23, 2017 (13:15)
- ...

1 TermR S T U V W X Y Z+ Add a new letter

13.1B

(B) **1** term Add Add termABCDEFGHIJKLMNOPQRSTUVWXYZ |

13.1.1 Book Created Mär 23, 2017 (13:12)

Synonym(s):

- Buch

Bezeichnung in der Anwendung für das Buch

13.2C

(C) **4** terms Add Add termABCDEFGHIJKLMNOPQRSTUVWXYZ |

13.2.1 Continuous Integration Created Feb 12, 2017 (20:45)

Abbreviation(s):

- CI

Automatisiertes und kontinuierliches bauen und integrieren der Komponenten durch einen entsprechenden Server wie Jenkins.

13.2.2 Customer Created Mär 23, 2017 (13:04)

Synonym(s):

- Kunde,
- Nutzer

Bezeichnung in der Anwendung für den Kunden oder Nutzer

13.2.3 Context and Dependency Injection Created Mär 24, 2017 (14:27)

Abbreviation(s):

- CDI

Teil des Java EE Frameworks. Handhabt Dependency Injection.

13.2.4 Continuous Delivery Created Mär 24, 2017 (14:32)

Abbreviation(s):

- CD

Automatisierts deployen der, durch Continuous Integration erstellten, Anwendung auf das Produktiv-System.

13.3D

(D) **1** term Add Add termABCDEFGHIJKLMNOPQRSTUVWXYZ |

13.3.1 Definition of Done Created Okt 10, 2016 (22:23)

- DoD

Abbreviation(s):

- DoD

Eine Checkliste um Qualität und vollständigen Abschluss/Bearbeitung einer Aktivität oder eines Artefakts zu überprüfen.

13.4E

(E) **1** term Add Add termABCDEFGHIJKLMNOPQRSTUVWXYZ |

13.4.1 Entity-Control-Boundary Created Mär 23, 2017 (13:20)

Abbreviation(s):

- ECB

Entity-Control-Boundary Pattern zur Strukturierung der Anwendung. Es stellt eine Variante des geläufigeren Model-View-Controller Pattern (MVC) dar.

13.5R

(R) **1** term Add Add termABCDEFGHIJKLMNOPQRSTUVWXYZ |

13.5.1 Rating Created Mär 23, 2017 (13:15)

Synonym(s):

- Bewertung,
- Buchbewertung,
- BookRating

Als Sammelbegriff für die Synonyme

14 Definition of Done

Hier wird beschrieben, wann die verschiedenen Arten von Aufgaben abgeschlossen sind.

14.1 Sprint Planning

- Sind ausreichend User Stories für einen Sprint vorhanden?
- Wurden ausreichend Tasks zur parallelen Bearbeitung erstellt?
- Sind die Tasks in unter einem Arbeitstag zu bewältigen?
- Wurde eine gemeinsame Schätzung des Aufwands gefunden?
- Wurden die bestehenden Impediments in der Planung berücksichtigt?
- Wurde das gewonnene Feedback aus der Retrospektive berücksichtigt?
- Versteht jedes Mitglied, was das Team im Sprint erreichen möchte?

14.2 User Story

- Wurde die US nach dem Template (Wer möchte was warum) klar formuliert?
- Ist die US in einem Sprint umsetzbar?
    - Kann man die US schätzen?
- Wurde die US gegengelesen?
- Wurde die US mit dem Kunden abgestimmt?

14.3 Task

- Wurde der “fertige” Task von einem anderen Mitglied gereviewed?
- Wurde eine Task ausreichend getestet?
    - Lokal, baut und besitzt Tests
- Ist der Task ins System integriert und deployed laut DevOps?
    - In Git, lauffähig, absolviert Test-Suite und in “production”
- Wurden die Coding Guidelines eingehalten?

15 Coding Guidelines

15.1 Style Conventions

15.1.1 Allgemein

Die maximale Anzahl von Zeichen in einer Zeile beträgt 180.

Methoden sollten so kurz gehalten werden, dass sie auf den Bildschirm passen und kein Scrollen erforderlich ist.

15.1.2 Leerzeichen

Keine Leerzeichen beim Methodenaufruf zwischen Methodennamen und Übergabeparametern.
Keine Leerzeichen nach öffnenden oder vor schließenden Klammern.
Keine Leerzeichen bei generischen Typen.
Keine Leerzeichen nach if-, for-, switch-Schlüsselwort.

Gut:

```
method(a);
b[2];
List<T> list = new ArrayList<T>();
if()
for()
switch()
```

Schlecht:

```
method ( a );
b [ 2 ];
List <T> = new ArrayList <T> ();
if ()
for ()
switch ()
```

Keine Leerzeichen nach Methodennamen bei der Methodendeklaration.

Gut:

```
void method()
```

Schlecht:

```
void method ()
```

Leerzeichen werden zur besseren Lesbarkeit von Ausdrücken verwendet.

Gut:

```
if(a + 5 > method(do() + 4))
```

Schlecht:

```
if(a+5>method(do()+4))
```

15.1.3 Klammern

Öffnende Klammern am Anfang eines Code Blocks werden in der gleichen Zeile gesetzt.

Gut:

```
if(a) {
 do();
}
```

Schlecht:

```
if(a)
{
 do();
}
```

Code Blöcke werden immer mit Klammern umschlossen, egal wie kurz sie sind.

Gut:

```
if(a) {
 if(b) {
    do();
 }
 else {
 }
}
```

Schlecht:

```
if(a)
 if(b)
    do();
```

15.1.4 Einrückung

Jeder neue Code Block wird eingerückt.

```
for(int i = 0; i < 10; i++) {
 switch(i) {
    case 0:
       doA();
 }

 if(i == 5) {
    doA();
 }
 else {
    doB;
 }
}
```

15.1.5 Methodenparameter

Methodenparameter werden mit Leerzeichen getrennt.
Zur besseren Lesbarkeit werden Methodenparameter, die über die festgelegte Zeilenlänge hinausgehen, über mehrere Zeilen formatiert.

Gut:

```
method(blabla, blubblub, blibli,
    bloblo, blabblab);
```

Schlecht:

```
method(blabla, blubblub, blibli, bloblo, blabblab);

method(blabla, blubblub, blibli,
bloblo, blabblab);
```

15.1.6 Mehrzeilige Kommentare

Über Klassen und Methoden:

```
/*
* Bla
```

```
* Blub
*/
```

Innerhalb von Methoden:

```
// Bla
// Blub
```

15.1.7 this

"this" wird nur verwendet, wenn ein Methodenparameter den gleichen Namen wie eine Member-Variable hat.

```
class Foo {
 int value;
 int number;

 void bla(int value, int value2) {
    this.value = value;
    number = value2;
 }

 void blub()
 {
    bla(1, 2);
 }
}
```

15.1.8 Benennungen

15.1.8.1 Klassen

Klassennamen sind Substantive (Was symbolisiert die Klasse?) und werden großgeschrieben.

15.1.8.2 Methoden

Methodennamen werden kleingeschrieben. Werden mehrere Wörter aneinandergehängt, so wird der erste Buchstabe jeden Wortes großgeschrieben.
Methoden sollten als Verben (Was tut die Methode?) benannt werden.

Namen von Methoden, die einen boolean-Wert zurückgeben und der Validierung von Variablen dienen, starten mit "is".

```
boolean isValid(int value)
```

15.1.8.3 Variablen

Variablennamen müssen aussagekräftig sein.
Die Aneinanderreihung von Wörtern erfolgt wie bei Methodennamen.
Keine Verwendung von "_" oder anderen Sonderzeichen in Variablennamen, Ausnahme sind Konstanten ("_" zur Trennung erlaubt).

```
class Foo {
 public static final int NUMBER_PI = 3;
 private int number;
 public String text;

 void setAge(int newNumber) {
    age = newNumber;
 }
}
```

15.1.8.4 Enums

```
public enum Day {
 MONDAY = 0,
 TUESDAY,
 WEDNESDAY,
    THURSDAY,
 FRIDAY,
 SATURDAY,
 SUNDAY
}
```

15.1.8.5 Test-Klassen

[Klassenname]Test

15.1.8.6 Tests-Methoden

Tests werden nach dem folgenden Schema benannt:

Should_ExpectedBehaviour_When_StateUnderTest

Beispiele:

Should_ThrowException_When_AgeLessThan18

Should_FailToWithdrawMoney_ForInvalidAccount

15.2 Guidelines

15.2.1 "Keep the code simple"

Generell gilt, dass Code so einfach wie möglich gehalten werden sollte. Die Antwort auf die Frage "Ist der Code auch für einen fremden Programmierer verständlich?" sollte immer "Ja" lauten.

Eine Methode, um Code verständlicher zu gestalten ist, diese in mehrere kleinere Teile zu zerlegen (Refaktorisierung).
Zusätzlich sollte darauf geachtet werden, unnötige Verschachtelungen zu verhindern.

15.2.2 Kommentare

Innerhalb von Methoden sollten lediglich Sachverhalte kommentiert werden, die nicht auf den ersten Blick durchschaubar sind. Es sollte vorher allerdings überprüft werden, ob diese Sachverhalte nicht z.B. durch bessere Variablenbenennung verständlich gemacht werden können.

Kommentare sollen auf Englisch verfasst werden.

Für alle Schnittstellen nach außen und derer Methoden sollen Javadoc-Kommentare erstellt werden.

15.2.3 Tests

Es wird nach dem Test First-Konzept vorgegangen. Dies bedeutet, dass Unit-Tests geschrieben werden müssen, bevor die eigentliche Funktionalität implementiert wird.

Laut Kent Beck wird in folgenden Schritten vorgegangen:

1. Schreiben einen Test, der auf einen bekannten Fehler überprüft und vom vorhandenen Programmcode nicht erfüllt wird.
2. Ändere den Programmcode so ab, dass er alle Tests erfüllt.
3. Refaktorisiere den Code (keine funktionellen Änderungen).
4. Fange wieder bei Schritt 1 an.

15.3 Versionskontrolle

Vor einem commit sollte überprüft werden, dass

- keine offenen Todo's im Code vorhanden sind
- kein auskommentierter Code vorhanden ist
- der Code fehlerfrei kompiliert
- alle überflüssigen imports entfernt sind

Es sollten regelmäßig (pro funktioneller Änderung) commits durchgeführt werden.

Ein commit sollte eine Beschreibung enthalten, die folgende Dinge enthält:

- Grund für Änderung

- (Ggfs.) Neue Funktionalität
- (Ggfs.) Bug-Fix

16 Bearbeitungsmatrix

Edit Document

Abschnitt	FD	AN	MS	Sum	Status	Kommentar
Einführung und Ziele						
Aufgabenstellung	70,00%		30,00%	100,00%	Done	
Qualitätsziele	100,00%			100,00%	Done	
Stakeholder	100,00%			100,00%	Done	
Randbedingungen						
Technische Randbedingungen	100,00%			100,00%	Done	
Organisatorische Randbedingungen	70,00%	30,00%		100,00%	Done	
Konventionen			100,00%	100,00%	Done	
Kontextabgrenzung						
Fachlicher Kontext			100,00%	100,00%	Done	
Technischer Kontext			100,00%	100,00%	Done	
Lösungsstrategie		100,00%		100,00%	Review	
Bausteinsicht						
Ebene 1			100,00%	100,00%	Done	
Ebene 2			100,00%	100,00%	Done	
Ebene 3			100,00%	100,00%	Done	
Laufzeitsicht						
Laufzeitszenario 1: Meine Bücher		100,00%		100,00%	Review	
Laufzeitszenario 2: Empfehlungen		100,00%		100,00%	Review	
Laufzeitszenario 3: Bewertung		100,00%		100,00%	Review	
Laufzeitszenario 4: Autorisierung		100,00%		100,00%	Review	
Laufzeitszenario 5: Deploy Vorgang			100,00%	100,00%	Done	

Verteilungssicht	100,00%			100,00%	Done	
Querschnittliche Konzepte						
Fachliche Strukturen und Modelle	100,00%			100,00%	Done	
Typische Muster, Strukturen und Abläufe	100,00%			100,00%	Done	
Persistenz		30,00%	70,00%	100,00%	Done	
Benutzungsoberfläche	100,00%			100,00%	Done	
Ablaufsteuerung		30,00%	70,00%	100,00%	Done	
Transaktionsbehandlung		100,00%		100,00%	Done	
Sessionbehandlung	100,00%			100,00%	Done	
Validierung, Ausnahme- und Fehlerbehandlung	50,00%	50,00%		100,00%	Done	
Konfigurierbarkeit			100,00%	100,00%	Done	
Migration			100,00%	100,00%	Done	
Testbarkeit			100,00%	100,00%	Done	
Buildmanagement			100,00%	100,00%	Done	
CI/CD-Pipeline			100,00%	100,00%	Done	
API Dokumentation	100,00%			100,00%	Done	
Entwurfsentscheidungen						
Teilung der Microservices		100,00%		100,00%	Review	
Umgang mit Dateninkonsistenz	100,00%			100,00%	Review	
Empfehlungssystem			100,00%	100,00%	Done	
Risiken & technische Schulden	100,00%			100,00%	Review	
Scrum im Hochschulumfeld	100,00%			100,00%	Done	
Evaluation der verwendeten Technologien	100,00%			100,00%	Review	
Glossar	-	-	-			
Individueller Beitrag	14,90	8,40	14,70	38,00		
Individueller Beitrag in %	39,21%	22,11%	38,68%			
Gesamtanzahl				38		

Beitrag zum Gesamtdokument in %	39,21%	22,11%	38,68%	100,00%