Erweiterung und Wartung einer Cloud-basierten JEE-Architektur

Nils Sommer

16. September 2017

Zusammenfassung

Im Rahmen eines Praktikums in der Gruppe Sofware Design and Quality (SDQ) wird ein Evolutionsszenario des CoCoME-Projekts umgesetzt. Die Cloud-Variante der CoCoME-Anwendung wird analysiert und eine für das Evolutionsszenario erforderliche Microservice-Architektur entworfen. Diese wird durch Nutzung entsprechender JEE-APIs implementiert.

1 Einleitung

Seit einigen Jahren zeichnet sich eine Entwicklung im Entwurf von Webbasierten Architekturen ab, bei dem monolithisch aufgebaute Anwendungen durch viele kleine Dienste ersetzt werden. Diese Dienste werden Microservices genannt. Für deren Charakterisierung existieren verschiedene Definitionen. An dieser Stelle soll die Definition von Newman verwendet werden.

> "Microservices are small, autonomous services that work together."
> [6, S. 2]

Dies bedeutet, dass ein Microservice eine einzelne Aufgabe erledigt. Die Eingrenzung und der Umfang der Funktionalität eines Microservice richtet sich dabei nach dem Geschäftsfeld, das die Funktionalität vorgibt. Diese sollte möglichst klar abgrenzbar sein und nicht in Konflikt mit anderen Geschäftsbereichen stehen. Die Eigenschaft der Autonomie bedeutet, dass ein

Microservice als isolierte Einheit betrieben werden kann. Bestehen Beziehungen zwischen Microservices, die Interaktionen erfordern, müssen diese über zuvor definierte Netzwerkschnittstellen abgewickelt werden.

Im Gegensatz zu Anwendungen mit monolithischer Architektur werden Microservices einzeln in den Betrieb ausgeliefert (engl. deployment). Davon profitieren insbesondere Anwendungen besonderer Größe und Continuous Deployment (CD)-Pipelines, die eine hohe Änderungsfrequenz besitzen. Des Weiteren ergibt sich aus der Entkopplung der Teilsysteme durch Web-APIs eine Unabhängigkeit von einzelnen Technologien wie Programmiersprachen oder Frameworks. Damit können die für einzelne Microservices optimalen Technologien gewählt werden um bspw. Performance-Anforderungen zu erfüllen.

Auf der anderen Seite werden in Microservice-Architekturen jedoch auch Abstraktionen und Indirektionen eingeführt, die in einer monolithischen Architektur entfallen. Wird Funktionalität eines anderen Microservice aufgerufen, wird ein Netzwerkaufruf gestartet und auf beiden Seiten durch entsprechende Adapter an die jeweiligen Programmiersprachen gebunden. Deshalb ist eine für die Anwendung passende Zerlegung der Funktiontlität in Teilsysteme, bei der die Abhängigkeiten zwischen diesen minimiert werden, sehr wichtig. Dazu kommen die bei verteilten Systemen üblichen Herausforderungen, wie bspw. der Umgang mit dem Ausfall eines einzelnen Microservice. Dazu existieren zahlreiche Lösungsansätze aus dem Bereich der Container-Orchestrierung, die jedoch nicht Teil dieser Arbeit sind.

Im Folgenden wird zunächst die Aufgabenstellung der Praktikumsarbeit beschrieben. Anschließend wird die Funktionalität und die Architektur der Webanwendung CoCoME analysiert. Daraufhin erfolgt der Entwurf einer Microservice-Architektur auf Basis der zuvor analysierten Funktionalität. Des Weiteren wird die Implementierung der Microservices beschrieben. Darauf folgt ein Abschnitt zu Microservice-Tests und den in dieser Arbeit implementierten Testfällen. Abschließend folgt ein Fazit sowie ein Ausblick auf zukünftige Weiterentwicklungen.

2 Aufgabenstellung

In dieser Praktikumsarbeit werden zwei primäre Aufgabenstellungen bearbeitet.

Zuerst soll die Cloud-Variante des Supermarktinformationssystems Co-

CoME und dessen monolithische Architektur in voneinander getrennt umsetzbare Teilsysteme zerlegt werden. Dabei soll jeweils ein vertikaler Schnitt durch das System erfolgen. Dies bedeutet, dass jedes Teilsystem alle Ebenen der Schichtenarchitektur abdeckt. Als Abgrenzungskriterien wird anstatt der technischen Ebenen die Funktionalität betrachtet. Demzufolge setzt jedes Teilsystem die Funktionalität eines eigenen Geschäftsfelds um. Dies erhöht die Wiederverwendbarkeit der Teilsysteme und minimiert deren technische Abhängigkeit voneinander.

Die funktionsorientierte Zerlegung von CoCoME in mehrere Teilsysteme konstituiert eine Microservice-Architektur. Darauf aufbauend gilt es den technischen Funktionsumfang festzulegen und die Schnittstellen der Microservices zu entwerfen.

Die zweite Aufgabenstellung besteht in der Implementierung der zuvor entworfenen Microservices. Dabei sollen die gleichen Technologien verwendet werden, die auch in der Cloud-Variante von CoCoME verwendet wurden. Dies sind die Schnittstellen und Bibliotheken der Java Enterprise Edition (JEE), die vom Glassfish-Anwendungsserver implementiert und bereitgestellt werden.

3 Analyse der CoCoME-Cloud-Anwendung

Der in dieser Arbeit betrachtete Teil von CoCoME schließt die Cloud-Variante ein, die das Evolutionsszenario Platform Migration umsetzt, sowie den Service Adapter, der das gleichnamige Evolutionsszenario umsetzt. Nicht betrachtet wird der optionale Pickup Shop.

Um den Entwurf einer Microservice-Architektur zu ermöglichen, muss zunächst die Geschäftslogik der CoCoME-Cloud-Variante identifiziert und analysiert werden. Dies wurde in drei Schritten angegangen. Zunächst wurde die vorhandene Dokumentation des CoCoME-Projekts gesichtet und die darin beschriebenen Geschäftsprozesse, die die Anwendung bereitstellt, identifiziert. Anschließend wurde die Cloud-Anwendung in einer Testumgebung eingerichtet und verschiedene Funktionen über die Weboberfläche getestet. Abschließend wurde der Quellcode der Cloud-Anwendung sowie des Service Adapters analysiert um die darin implementierte Domänen- und Anwendungslogik im Detail zu verstehen.

Abbildung 1 zeigt einen Überblick über die Anwendungsfälle der CoCoME-Anwendung und der beteiligten Benutzerrollen. Dies zeigt die für ein Web-

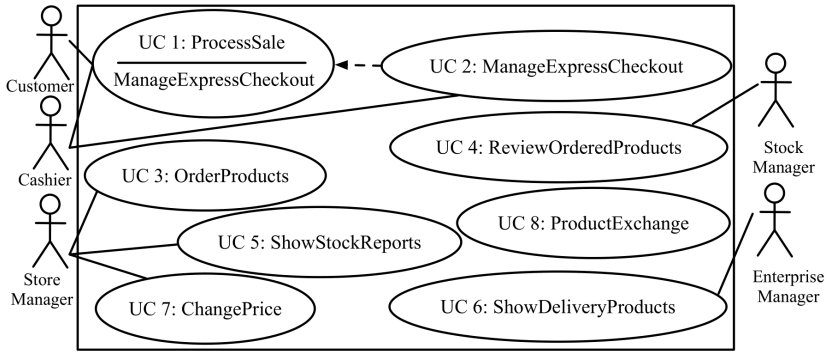

Abbildung 1: Use Cases der CoCoME-Anwendung [5]

basiertes Supermarktinformationssystem üblichen Vorgänge. Dazu gehören Aktivitäten zum Abwickeln von Einkäufen, die Pflege des Inventars von Produkten und die Generierung von Berichten über die Geschäftsvorgänge.

In der bestehenden Architektur der Cloud-basierten Variante von CoCoME mit Service Adapter fungiert dieser als zentraler Dienst zur Datenspeicherung und Datenabfrage. Jeder Teil der Anwendung, der auf die Datenbank zugreift, interagiert über ein Web-basiertes Protokoll und XML-Nachrichten mit dem Service Adapter. Der Service Adapter läuft als eigene Anwendung in einem Glassfish-Server und verwendet eine von Glassfish bereitgestellte relationale Datenbank.

4 Entwurf der Microservice-Architektur

Um die monolithische Architektur der Cloud-Variante von CoCoME in eine Microservice-Architektur zu transformieren, müssen geeignete Service-Kandidaten identifiziert werden. Einen Ansatz hierfür bildet das Domain-Driven Design nach Eric Evans [3]. Dies stellt ein umfangreiches Entwurfsverfahren dar, mit dem die Domäne einer Anwendung in einem Modell beschrieben wird. Durch die ubiquitäre Sprache in einem Domänenmodell und den damit beschriebenen Konzepten und Objekten entsteht eine allgemein verständliche Dokumentation der Anwendungsdomäne. Des Weiteren liefert ein Domänenmodell die Struktur der Domänenschicht in einer Layered Architecture und trägt somit zu einem sauberen Entwurf bei.

Im Folgenden sollen die wichtigsten Domänenobjekte des Informations-

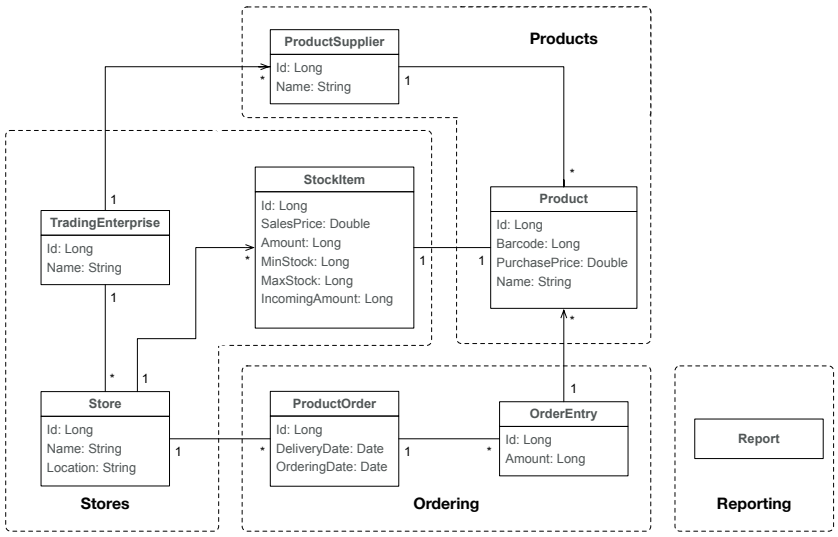

Abbildung 2: Bounded Contexts

systems betrachtet werden und durch das Entwurfsmuster des *Bounded Context* in voneinander abgrenzbare Teilbereiche unterteilt werden [3, S. 335]. Diese bilden einen guten Ausgangspunkt zur Implementierung einzelner Microservices [9]. In Abbildung 2 sind die wichtigsten Domänenobjekte des Informationssystems sowie der Assoziationen zwischen diesen in Form eines UML-Klassendiagramms abgebildet. In Form grau-gestrichelter Linien wurden die Grenzen der entworfenen Bounded Contexts eingezeichnet. Die vier Bereiche dienen der Unterteilung der Funktionalität zur Implementierung in separaten Diensten, die im Folgenden aufgelistet sind.

- Unternehmen und Filialen werden in einem **StoresService** verwaltet. Dieser verwaltet auch deren Inventar (StockItems).
- Produktinformationen werden in einem **ProductsService** gespeichert. Dazu gehören auch die jeweiligen Anbieter (ProductSupplier). Die Trennung von Produkten und Ladenfilialen (Stores) ermöglicht es, Produkte auch separat vom Angebot in diesen zu betrachten.
- Die Bestellungen der Kunden werden mit einem **OrdersService** aufgegeben und verwaltet. Diese Subdomäne im Entwurf von Filialen und Produkten zu entkoppeln ermöglicht es eine in Zukunft mögliche Anbindung an Kassensysteme oder Versandanbieter einfacher umzusetzen.

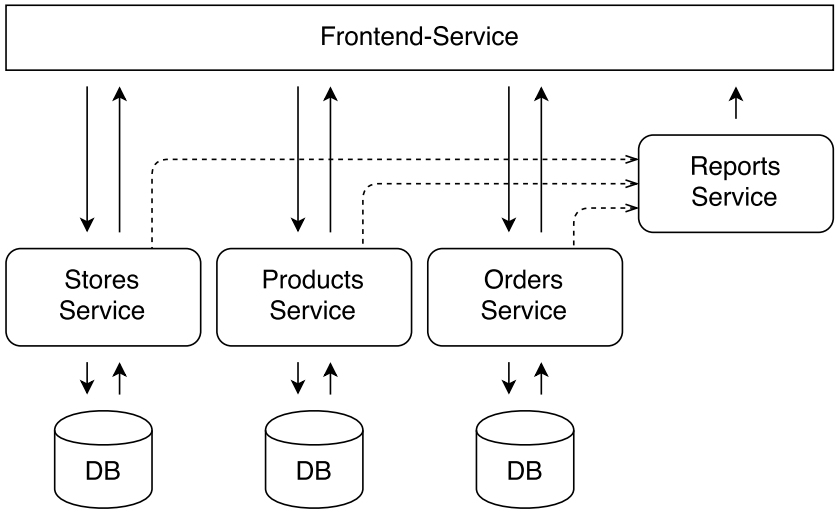

Abbildung 3: Microservice-Architektur des CoCoME E-Commerce Shops

- Berichte zur Auswertung und Archivierung der Geschäftstätigkeiten werden durch einen **ReportsService** generiert. Berichte können als Aggregation von Informationen anderer Dienste angesehen werden. Die Umsetzung als eigenständiger Dienst und der angestrebten losen Kopplung mit anderen Diensten erleichtert das Austauschen der datenbereitstellenden Dienste sowie der Erweiterung um weitere Datenquellen.

Abbildung 3 zeigt die Microservice-Architektur mit den zuvor genannten Diensten. Die Dienste zur Verwaltung von Ladenfilialen, Produkten und Bestellungen benötigen jeweils eine eigene Datenbank zur Speicherung der Daten. Dadurch können die Dienste autonom betrieben werden. Außerdem wird dadurch eine Abhängigkeit von einem einzelnen Endpunkt vermieden, wie dies in Form des Service Adapters in der aktuellen Cloud-Variante von CoCoME der Fall ist. Dies sollte aus Gründen der Skalierbarkeit und der Ausfallsicherheit vermieden werden. Der Dienst zur Generierung von Berichten benötigt keine eigene Datenbanken, da er lediglich die von den anderen Diensten bereitgestellten Daten aufbereitet. Die Informationsflüsse zwischen den Einheiten in der Architektur werden durch die Pfeile dargestellt. Da Microservices autonom betrieben können sollen, müssen u.U. erforderliche Interaktionen zwischen diesen über das Netzwerk und standardisierte Schnittstellen erfolgen [11].

Neben den domänenorientierten Microservices enthält die Microservice-

Architektur zudem noch einen Frontend-Service. Dieser wird benötigt, um Benutzern eine einheitliche grafische Oberfläche als Einstiegspunkt und Interaktionspartner bereitzustellen. Der Frontend-Service arbeitet rein anwendungsspezifisch. Dies bedeutet, dass zum einen die Benutzeroberfläche in Form dynamisch generierter HTML-Seiten bereitgestellt werden. Zum anderen delegiert der Frontend-Service die zur Bereitstellung einer Funktionalität, die durch Interaktion mit einer bestimmten Seite der Weboberfläche aufgerufen wird, benötigten Aktionen an die domänenorientierten Microservices weiter und übernimmt somit die Orchestrierung der Dienste innerhalb eines Geschäftsprozesses. Des Weiteren kann ein zentraler Frontend-Service Querschnittsaufgaben wie bspw. das Identity- und Access-Management (IAM) übernehmen.

In einem Entwurf ohne Frontend-Service müssten die domänenorientierten Microservices selbst die jeweils benötigten Oberflächenelemente bereitstellen. Dies würde zwar die bessere Trennung der Services als vertikale Schnitte durch die Anwendung bedeuten, würde jedoch auch signifikante Nachteile bei der Implementierung nach sich ziehen. Zum einen müssten Oberflächen, die Service-übergreifende Funktionalität benötigen, die Oberflächenkomponenten der Services komponieren, was ohne einen zentralen Frontend-Service zu einer engen Kopplung der Dienste führen würde. Zum würde dies Querschnittsfunktionen wie Authentifizierung und Autorisierung erschweren. Diese müssten in jedem einzelnen Microservice umgesetzt werden. Des Weiteren wäre eine Service-übergreifende Session durch Cookies aufgrund separater Hosts oder Ports nicht möglich.

5 Implementierung

Die Microservices wurden mit JEE-APIs implementiert. JEE stellt Schnittstellen für zahlreiche Grundfunktionalitäten von Webanwendungen bereit. In dieser Arbeit wurden vor allem die folgenden APIs verwendet.

- Die **Java Persistence API (JPA)** ist eine Schnittstelle zur Umsetzung von Persistenzfunktionen und der Abbildung von Java-Klassen auf die Strukturen relationaler Datenbanken. Diese Schnittstelle wurde in Store-, Product- und OrderService verwendet, um die Datenspeicherung und Datenabfrage zu implementieren.

- Die **JAX-RS** API ermöglicht die Implementierung von ressourcenorientierten Web-APIs nach dem REST-Architekturstil. Durch die API können URI-Pfadsegmente, HTTP-Operationen und Header-Felder auf Java-Klassen und Methoden abgebildet werden.

- **JAXB** liefert eine Schnittstelle zur Serialisierung und Deserialisierung von Java-Klassen zu und von XML- oder JSON-Dateien. Dadurch können Ressourcenrepräsentationen generiert bzw. ausgelesen werden.

Die Schnittstellen der Microservices sind als ressourcenorientierte Web-APIs implementiert. Diese folgen im wesentlichen dem REST-Architekturstil [4,7,10], vernachlässigen jedoch die Hypermedia-Eigenschaft von REST. Eine detaillierte Beschreibung der Web-API kann der Dokumentation des Repositories [8] entnommen werden.

Jeder Service enthält mehrere Bestandteile, die wie folgt strukturiert sind. Das Namenssegment `<name>` kann jeweils durch den Namen des betrachteten Microservice ersetzt werden.

- Das Maven-Projekt `<name>-service` dient als Überprojekt für die folgenden drei Unterprojekte. Durch Maven-Kommandos in diesem Projekt wird der Service an den Glassfish-Anwendungsserver ausgeliefert.

  - Das Maven-Projekt `<name>-service-ejb` enthält die Geschäftslogik des Microservice. Dort wird zudem die Datenpersistenzschicht implementiert.
  - Das Maven-Projekt `<name>-service-rest` enthält die Implementierung der ressourcenorientierten Web-API mit JAX-RS. Die benötigte Geschäftslogik wird aus dem EJB-Projekt geladen und auf Klassenebene durch *Contexts and Dependency Injection (CDI)* referenziert, sodass die Instanziierung und das Lebenszyklusmanagement der EJB-Klassen von der JEE-Umgebung verwaltet wird.
  - Das Maven-Projekt `<name>-service-ear` dient zur Paketierung des Dienstes im JEE-eigenen EAR-Format. Ein solches Anwendungspaket kann im Glassfish-Anwendungsserver betrieben werden.

- Das Maven-Projekt `<name>-client` enthält eine Bibliothek, die ein Java-API zur Konsumierung des Microservices bereitstellt. Somit kann

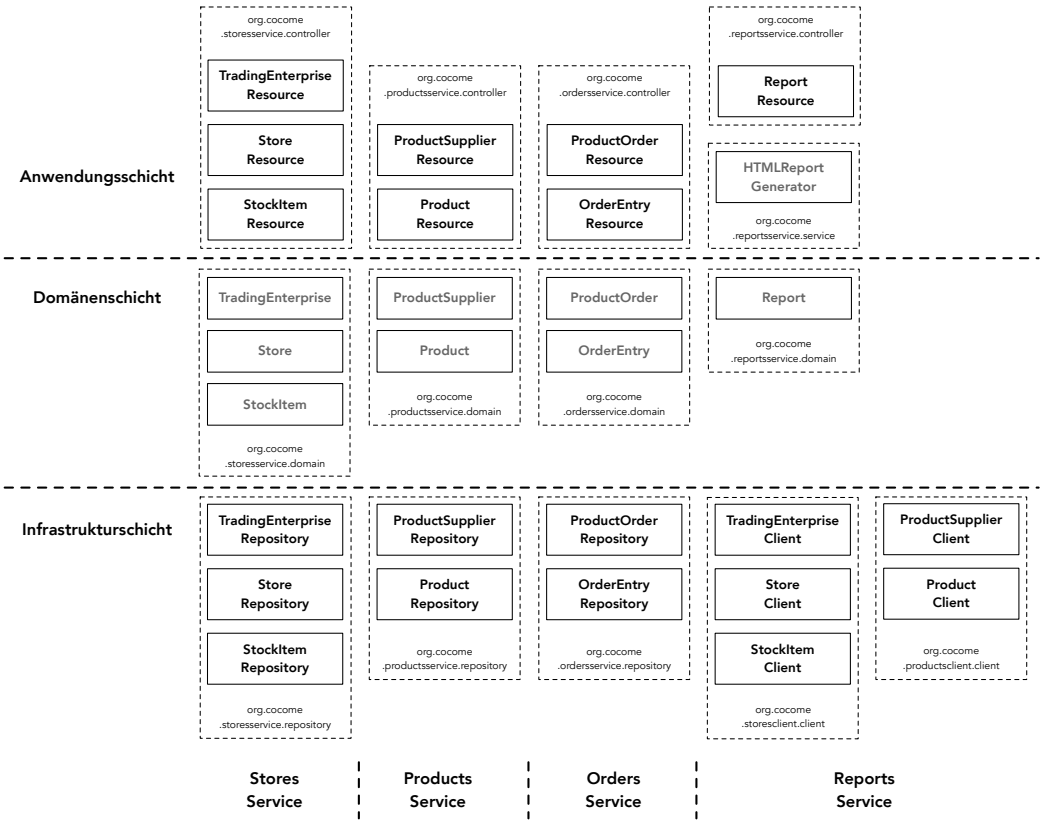

Abbildung 4: Komponentenüberblick

der entsprechende Microservice einfach von anderen Services aus verwendet werden.

- Das Verzeichnis `fixtures` enthält XML-Dateien mit Beispieldaten für den Service. In der Dokumentation, die im Repository beiliegt, wird beschrieben wie der Service anhand dieser Beispieldaten aufgerufen werden kann. Dazu wird das Terminal-Programm cURL verwendet.

- Das Maven-Projekt `<name>-api-tests` enthält die für den Microservice implementierten Testfälle. Dabei handelt es sich um sogenannte Vertragstests, die das Web-API des Microservice testen. Aufbau und Funktionsweise dieser Tests werden in Abschnitt 6 genauer beschrieben.

Die Komponenten auf Quellcode-Ebene der Microservices sind in Abbildung 4 dargestellt. Die Java-Packages, die in gestrichelten Rahmen darge-

stellt sind, werden nach den Schichten der Layered Architecture [3, S. 68] in Infrastruktur-, Domänen- und Anwendungsschicht gruppiert. Die in den Java-Packages enthaltenen Java-Klassen, deren Namen grau gedruckt ist, wurden aus der Cloud-Variante von CoCoME übernommen. Dies sind vor allem die Klassen der Domänenobjekte sowie die Funktionalität zur Generierung von Berichten in HTML-Form. Der Zugriff auf Datenbanken in der Infrastrukturschicht wurden nach dem Repository-Muster neu implementiert, weil dadurch der saubere Aufbau und die Struktur des Quellcodes signifikant gegenüber der Implementierung im Service Adapter der Cloud-Variante von CoCoME verbessert werden konnte. Die Resource-Klassen in der Anwendungsschicht, die die REST-Schnittstellen implementieren, wurden ebenfalls selbst implementiert, da diese zuvor nicht vorhanden waren. Ebenso wurden entsprechende Client-Bibliotheken implementiert, die in Abbildung 4 in der Infrastrukturschicht des Reports-Service zu finden sind. Die einzelnen Microservices sind im unteren Bildabschnitt markiert und stellen jeweils einen vertikalen Schnitt durch das System dar.

6 Microservice-Tests

Automatisierte Tests sind wichtig, um die Qualität der Software sicherzustellen. In einer Microservice-Architektur können Tests auf unterschiedlichen Ebenen der Architektur angesiedelt werden. Ein Überblick über die möglichen Testebenen in einer Microservice-Architektur ist in [1, 2] zu finden. Die möglichen Ebenen für Tests sind demnach die folgenden: (1) Unit-Tests testen eine einzelne Komponente in Isolation und werden i.d.R. für einzelne Klassen erstellt, (2) Integrationstests adressieren die Zusammenarbeit mehrerer Komponenten, (3) Komponententests testen größere Einheiten einer Software, (4) Vertragstests (engl. contract tests) oder auch API-Tests testen die öffentliche Schnittstelle eines Microservices und (5) Systemtests adressieren das Softwaresystem als Ganzes, indem die Benutzeroberfläche verwendet wird.

Für die Microservices in dieser Arbeit wurden Vertragstests implementiert. Die Testfälle interagieren demnach mit den ressourcenorientierten Web-APIs der Microservices. Die Assertions in den Testfällen überprüfen die von einem HTTP-Request zurück gelieferten Daten bzw. den Status-Code der Antworten. Die Testfälle wurden mit dem JUnit-Framework implementiert und verwenden die Client-Bibliotheken der Microservices zum Aufrufen

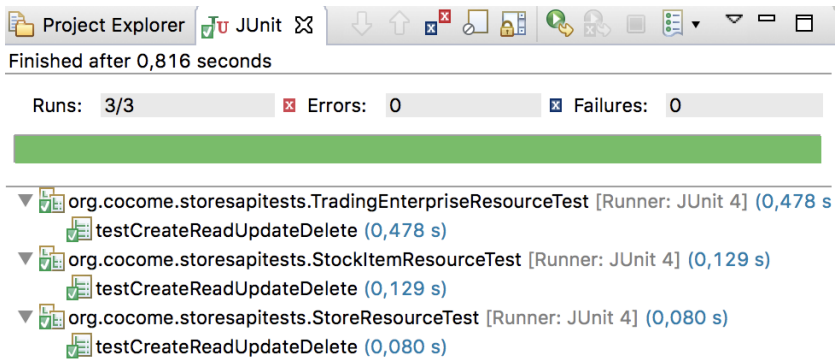

Abbildung 5: Testprotokoll der API-Tests der Stores-Microservice in Eclipse

der jeweiligen Web-APIs. Die Testfälle sind nach Microservice geordnet jeweils in einem eigenen Eclipse-Projekt umgesetzt. Dies hat den Vorteil, dass die Testfälle weiterverwendet werden können, auch wenn die Microservice-Implementierung durch eine andere ersetzt wird. Da die Vertragstests nur das Web-API benötigen, das von Programmiersprachen unabhängig ist und die Testfälle in einem separaten Eclipse-Projekt enthalten sind, ist die Unabhängigkeit der Tests von einer konkreten Service-Implementierung gegeben.

Die Testfälle können automatisiert durch den Aufruf von `mvn test` im Terminal aufgerufen werden oder durch die JUnit-Oberfläche in Eclipse. Diese ist in Abbildung 5 nach dem Aufruf der Testfälle des Stores-Service zu sehen und zeigt das Testprotokoll geordnet nach Ressourcen an. Die Vertragstests wurden für die Stores-, Products- und Orders-Services erstellt. Die Testfälle testen für jede Ressource der Web-API die Operationen Create Read Update Delete (CRUD). Da die Ausführungsreihenfolge von JUnit-Tests erst zur Laufzeit entschieden wird, müssen die Tests der CRUD-Operationen in einer Methode implementiert werden, um den Lebenszyklus der jeweiligen Ressource in der richtigen Reihenfolge zu durchlaufen.

7 Fazit und Ausblick

In dieser Arbeit wurde zunächst das Supermarktinformationssystem CoCoME analysiert und gemäß des Domain-Driven Design in Bounded Contexts unterteilt. Darauf aufbauend wurde eine Microservice-Architektur entworfen. Diese wurde mithilfe von JEE-Technologien implementiert. Die dabei umge-

setzten Microservices stellen ihre Funktionalität über ressourcenorientierte Web-APIs gemäß dem REST-Architekturstil bereit.

In dieser Arbeit nicht umgesetzt wurde der im Architektur-Entwurf beschriebene Frontend-Service. Dieser wird zur Bereitstellung der Benutzeroberfläche, bspw. durch Verwendung von Java Server Faces (JSF) benötigt. Zudem wurde die Benutzerverwaltung und Authentifizierung, die über den Frontend-Service abgewickelt werden kann, nicht betrachtet.

Um einen Frontend-Service zu implementieren, können die jeweiligen Client-Bibliotheken der Microservices verwendet werden, um die benötigten Aufrufe dieser umzusetzen. Des Weiteren wurde im Quellcode-Repository bereits die JEE-spezifische Projektstruktur für einen Frontend-Service erstellt. Somit besteht bereits die Grundlage für diese weiterführende Arbeit. Da die Client-Bibliotheken nicht explizit von den Microservice-Implementierungen abhängen, sondern lediglich deren Web-API konsumieren, besteht keine harte Abhängigkeit zwischen diesen. Dadurch entsteht die in Microservice-Architekturen angestrebte loose Kopplung zwischen Service-Bereitsteller und den Service-Konsumenten.

Literatur

[1] Cloves Carneiro and Tim Schmelmer. Testing with services. In *Microservices From Day One*, pages 127–150. Apress, 2016.

[2] Toby Clemenson. Testing strategies in a microservice architecture. `https://martinfowler.com/articles/microservice-testing/`, November 2014.

[3] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[4] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[5] Robert Heinrich, Kiana Rostami, and Ralf Reussner. The cocome platform for collaborative empirical research on information system evolution. 2016.

[6] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[7] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Sebastopol, CA, Mai 2007.

[8] Nils Sommer. Github repository – restful microservices for cocome. `https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest`, 2017.

[9] Roland H. Steinegger, Pascal Giessler, Benjamin Hippchen, and Sebastian Abeck. Overview of a domain-driven design approach to build microservice-based applications. In *The Third International Conference on Advances and Trends in Software Engineering*, pages 79–87, Venedig, Italien, April 2017.

[10] Stefan Tilkov, Martin Eigenbrodt, Silvia Schreier, and Oliver Wolf. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web (German Edition)*. dpunkt.verlag, 2015.

[11] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

Anhang

Im Folgenden wird der Inhalt der README-Dateien aus dem Quellcode-Repository aufgeführt.

RESTful Microservices for CoCoME

This repository contains JEE-based microservice implementations for the CoCoME application.

In the microservice architecture design, four sub domains have been identified to be implemented as seperate services.

- **Orders**: A service to make and track orders of customers.
- **Products**: A service to manage products.
- **Reports**: A service to generate reports.
- **Stores**: A service to manage enterprises and their stores.

The services are designed to use their own relational database each and follow a RESTful architecture style, exposed as an interface via HTTP. The implementations live in the corresponding subdirs of this repository which also contain instructions on how to setup and deploy each service.

General Structure of the Services

- Each service is built out of three maven projects. (1) `<name>-service-ejb` contains the domain models, the business logic and the persistence layer of the service. (2) `<name>-service-rest` contains the RESTful Web-API of the service. (3) `<name>service-ear` packages the service into a deployable archive.
- Each service has a parent maven project that holds the `*-ejb`, `*-rest` and `*-ear` projects.
- These parent maven projects offer deploying and undeploying of the service via maven.
- Each service contains a subdirectory `fixtures/` with example data XML files and instructions on how to use the service using cURL described in the corresponding `README.md`.
- Each service contains a separate Eclipse project containing contract tests in a subdirectory `<name>-api-tests`. These projects use the corresponding client library for the service to test.

Development Setup

To import this project into Eclipse, follow these steps.

- *File -> Import -> Maven -> Exising Maven Projects -> Click "Next"*
- *Root Directory -> Click "Browse" ->* Select the directory of this project
- *Click "Finish"*

Installation in a nutshell

Each microservice includes a README file with detailed deployment instructions. If you want to set up all of them, follow these steps:

1. Create the domains

```
asadmin create-domain --portbase 8800 storesmicroservice
```

```
asadmin create-domain --portbase 8900 productsmicroservice
asadmin create-domain --portbase 8700 ordersmicroservice
asadmin create-domain --portbase 8600 reportsmicroservice
```

2. Start the domains

```
asadmin start-domain storesmicroservice
asadmin start-domain productsmicroservice
asadmin start-domain ordersmicroservice
asadmin start-domain reportsmicroservice
```

3. Create the databases

```
asadmin create-jdbc-resource --connectionpoolid DerbyPool --host localhost --port
8848  jdbc/CoCoMEStoresServiceDB
asadmin create-jdbc-resource --connectionpoolid DerbyPool --host localhost --port
8948  jdbc/CoCoMEProductsServiceDB
asadmin create-jdbc-resource --connectionpoolid DerbyPool --host localhost --port
8748  jdbc/CoCoMEOrdersServiceDB
```

4. Start the databases

```
asadmin start-database
```

5. Build the client libraries

The client libraries need to be built first, because some of the services depend on them.

```
cd stores/stores-client     & mvn install
cd products/products-client & mvn install
cd orders/orders-client     & mvn install
cd reporst/reports-client   & mvn install
```

6. Build and deploy the services

The maven commands not only builds deployable packages but also deploys them to their corresponding glassfish domains.

```
cd stores/stores-service     & mvn -s settings.xml install
cd products/products-service & mvn -s settings.xml install
cd orders/orders-service     & mvn -s settings.xml install
cd reports/reports-service   & mvn -s settings.xml install
```

Stores Service

Deployment Setup

Prerequisites: Java8, Maven and Glassfish installed and ready to use.

Before the first deployment, the domain for the jee application (which is the stores microservice) has to be created.

```
asadmin create-domain --portbase 8800 storesmicroservice
```

After that, a JBCD resource has to be created.

```
asadmin create-jdbc-resource --connectionpoolid DerbyPool --host localhost --port
8848  jdbc/CoCoMEStoresServiceDB
```

Deploying into Glassfish

Start the database and domain first.

```
asadmin start-database & asadmin start-domain storesmicroservice
```

Then use maven to compile the sources, build the deployment package and deploy to glassfish.

```
mvn -s settings.xml install
```

Undeploying works using maven as well.

```
mvn -s settings.xml clean post-clean
```

RESTful API

URI Schema: `http://{hostname}:8880/storesmicroservice/rest` + resource path

Resource: Trading Enterprise

Path	HTTP Operation	Status Code	Response
/trading-enterprises	GET	200	XML representations of resource list
/trading-enterprises	POST	201	URI to resource in `Location` header
/trading-enterprises/{id}	GET	200	XML representation of resource
/trading-enterprises/{id}	PUT	204	Empty
/trading-enterprises/{id}	DELETE	204	Empty

Resource: Store

Path	HTTP Operation	Status Code	Response
/trading-enterprises/{id}/stores	GET	200	XML representations of resource list
/trading-enterprises/{id}/stores	POST	201	URI to resource in `Location` header
/stores/{id}	GET	200	XML representation of resource
/stores/{id}	PUT	204	Empty
/stores/{id}	DELETE	204	Empty

Resource: Stock Item

Path	HTTP Operation	Status Code	Response
/stores/{id}/stock-items	GET	200	XML representations of resource list
/stores/{id}/stock-items	POST	201	URI to resource in `Location` header
/stock-items/{id}	GET	200	XML representation of resource
/stock-items/{id}	PUT	204	Empty
/stock-items/{id}	DELETE	204	Empty

Working with test data

`fixtures/` contains XML files for the service's domain objects. You can use them as test data. For example, use cURL to save data using the service.

```
curl -X POST http://localhost:8880/storesmicroservice/rest/trading-enterprises -H
"Content-Type: application/xml" -d @fixtures/tradingenterprise/1.xml -v
```

By using the `-v` flag, you will see that the service answers with a *201 Created* status code and the URI to the newly created resource in the `Location` header.

Assuming the service was running and using the ID 1, fetch it using cURL like that:

```
curl -X GET http://localhost:8880/storesmicroservice/rest/trading-enterprises/1 -H
"Accept: application/xml"
```

Reports Service

Deployment Setup

Prerequisites: Java8, Maven and Glassfish installed and ready to use.

Before the first deployment, the domain for the jee application (which is the reports microservice) has to be created.

```
asadmin create-domain --portbase 8600 reportsmicroservice
```

Deploying into Glassfish

Start the database and domain first.

```
asadmin start-domain reportsmicroservice
```

Then use maven to compile the sources, build the deployment package and deploy to glassfish.

```
mvn -s settings.xml install
```

Undeploying works using maven as well.

```
mvn -s settings.xml clean post-clean
```

RESTful API

URI Schema: `http://{hostname}:8680/storesmicroservice/rest` + resource path

Resource: Report

Path	HTTP Operation	Status Code	Response	Parameters
/reports	POST	200	XML representation of report	`type=(enterprise-delivery|store-stock|enterprise-stock), id`

Products Service

Deployment Setup

Prerequisites: Java8, Maven and Glassfish installed and ready to use.

Before the first deployment, the domain for the jee application (which is the products microservice) has to be created.

```
asadmin create-domain --portbase 8900 productsmicroservice
```

After that, a JBCD resource has to be created.

```
asadmin create-jdbc-resource --connectionpoolid DerbyPool --host localhost --port
8948  jdbc/CoCoMEProductsServiceDB
```

Deploying into Glassfish

Start the database and domain first.

```
asadmin start-database & asadmin start-domain productsmicroservice
```

Then use maven to compile the sources, build the deployment package and deploy to glassfish.

```
mvn -s settings.xml install
```

Undeploying works using maven as well.

```
mvn -s settings.xml clean post-clean
```

RESTful API

URI Schema: `http://{hostname}:8980/productsmicroservice/rest` + resource path

Resource: Product Supplier

Path	HTTP Operation	Status Code	Response
/trading-enterprises/{id}/product-suppliers	GET	200	XML representations of resource list
/trading-enterprises/{id}/product-suppliers	POST	201	URI to resource in `Location` header

Path	HTTP Operation	Status Code	Response
/product-suppliers/{id}	GET	200	XML representation of resource
/product-suppliers/{id}	PUT	204	Empty
/product-suppliers/{id}	DELETE	204	Empty

Resource: Product

Path	HTTP Operation	Status Code	Response
/product-suppliers/{id}/products	GET	200	XML representations of resource list
/product-suppliers/{id}/products	POST	201	URI to resource in `Location` header
/products/{id}	GET	200	XML representation of resource
/products/{id}	PUT	204	Empty
/products/{id}	DELETE	204	Empty

Working with test data

`fixtures/` contains XML files for the service's domain objects. You can use them as test data. For example, use cURL to save data using the service. The following request presumes that there exists a trading enterprise with ID 1.

```
curl -X POST
http://localhost:8980/productsmicroservice/rest/trading-enterprises/1/product-suppl
iers -H "Content-Type: application/xml" -d @fixtures/productsupplier/1.xml -v
```

By using the `-v` flag, you will see that the service answers with a *201 Created* status code and the URI to the newly created resource in the `Location` header.

Assuming the service was running and using the ID 1, fetch it using cURL like that:

```
url -X GET http://localhost:8980/productsmicroservice/rest/product-suppliers/1 -H
"Accept: application/xml"
```

Ordering Service

Deployment Setup

Prerequisites: Java8, Maven and Glassfish installed and ready to use.

Before the first deployment, the domain for the jee application (which is the orders microservice) has to be created.

```
asadmin create-domain --portbase 8700 ordersmicroservice
```

After that, a JBCD resource has to be created.

```
asadmin create-jdbc-resource --connectionpoolid DerbyPool --host localhost --port
8748  jdbc/CoCoMEOrdersServiceDB
```

RESTful API

URI Schema: `http://{hostname}:8780/storesmicroservice/rest` + resource path

Resource: Product Order

Path	HTTP Operation	Status Code	Response
/stores/{id}/product-orders	GET	200	XML representations of resource list
/stores/{id}/product-orders	POST	201	URI to resource in `Location` header
/product-orders/{id}	GET	200	XML representation of resource
/product-orders/{id}	PUT	204	Empty
/product-orders/{id}	DELETE	204	Empty

Resource: Order Entry

Path	HTTP Operation	Status Code	Response
/product-orders/{id}/order-entries	GET	200	XML representations of resource list
/product-orders/{id}/order-entries	POST	201	URI to resource in `Location` header
/order-entries/{id}	GET	200	XML representation of resource
/order-entries/{id}	PUT	204	Empty
/order-entries/{id}	DELETE	204	Empty

Deploying into Glassfish

Start the database and domain first.

```
asadmin start-database & asadmin start-domain ordersmicroservice
```

Then use maven to compile the sources, build the deployment package and deploy to glassfish.

```
mvn -s settings.xml install
```

Undeploying works using maven as well.

```
mvn -s settings.xml clean post-clean
```

Frontend Service

This service acts as the interface between the microservices that implement the domain-oriented functionality and the user by prodiving a web-based user interface.

Note: This service is not implemented yet. Use this project structure for the implementation.

Deployment Setup

Prerequisites: Java8, Maven and Glassfish installed and ready to use.

Before the first deployment, the domain for the jee application (which is the frontend service) has to be created.

```
asadmin create-domain --portbase 8500 frontendservice
```

Deploying into Glassfish

Start the domain first.

```
asadmin start-domain frontendservice
```

Then use maven to compile the sources, build the deployment package and deploy to glassfish.

```
mvn -s settings.xml install
```

Undeploying works using maven as well.

```
mvn -s settings.xml clean post-clean
```