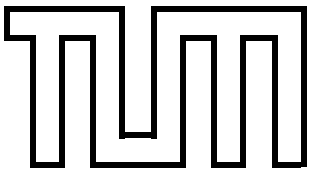


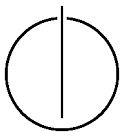
FAKULTÄT FÜR INFORMATIK

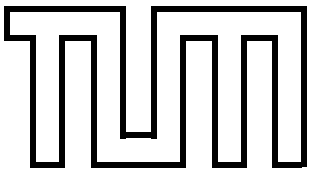
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Konzeption und prototypische Realisierung einer einfachen, hochgradig skalierbaren Multi-Mandanten-Architektur auf Basis der Google Cloud Services

Sebastian Wenninger





FAKULTÄT FÜR INFORMATIK

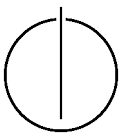
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Design and prototypical implementation of a simple, highly-scalable multi-tenant architecture based on Google Cloud Services

Konzeption und prototypische Realisierung einer einfachen, hochgradig skalierbaren Multi-Mandanten-Architektur auf Basis der Google Cloud Services

Autor:	Sebastian Wenninger
Themensteller:	Prof. Dr. Florian Matthes
Betreuer:	Prof. Dr. Florian Matthes
Abgabedatum:	15. September 2011



Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 14. September 2011 Sebastian Wenninger

Zusammenfassung

Cloud Computing ist ein schnell an Bedeutung gewinnendes, vergleichsweise neues Konzept in der IT-Branche, bei dem Dienstleistungen (sowohl von Hard-, als auch von Software) über ein Netzwerk zur Nutzung angeboten werden. Dadurch ist es Entwicklern möglich, Anwendungen zu erstellen, die weitestgehend unabhängig von der individuellen Konfiguration eines Nutzers auf einer Vielzahl von Endgeräten lauffähig und durch die Anbindung an das Internet weltweit abrufbar sind. Dafür existieren mittlerweile mehrere Plattformen, wie zum Beispiel die Google App-Engine (GAE), die Entwickler Applikationen auf Google's Infrastruktur ausführen lässt. Insbesondere für Unternehmen, die bestehende Projekte in die Cloud auslagern, oder neue Dienste in der Cloud anbieten wollen, sind die Möglichkeiten und Einschränkungen, die diese Plattformen mit sich bringen, von großem Interesse.
Im Rahmen dieser Arbeit wird eine einfache, Multi-Mandanten-fähige eCommerce-Architektur mit Hilfe der Google App-Engine entwickelt, die über einen Webservice den Zugriff auf ihre Funktionen gestattet. Dafür werden zuerst in einem grundlegenden Kapitel Cloud-Computing und Mehr-Mandantenfähigkeit betrachtet. Die Beschreibung der Architektur erfolgt danach nach dem „Bottom-Up" - Prinzip, von der Persistenzebene über die Serviceschicht bis zu den darüberliegenden Komponenten der Präsentationsschicht. Dabei wird jeweils auf die zugehörigen Funktionen der Google-App-Engine, Schwierigkeiten und mögliche Alternativen bei der Implementierung eingegangen.
Den Schluss der Arbeit bilden ein Fazit sowie ein Ausblick auf die Zukunft der entwickelten Applikation.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Seit noch nicht allzu langer Zeit ist der Begriff Cloud Computing in der IT-Welt zu einem vieldiskutiertem Thema geworden.
Das Angebot von IT-Dienstleistungen als Service unter einem "pay-as-you-go" Bezahlmodell hat sich für viele kleine und mittlere Firmen zu einem attraktiven Angebot entwickelt. Anstatt selbst in teure Hard-/Software investieren zu müssen, können diese Kapazitäten zu variablen Kosten von Anbietern wie Microsoft, Google oder Amazon aus der Cloud bezogen werden. Das Unternehmen bleibt flexibel und kann gleichzeitig sparen. Dasselbe gilt natürlich auch für die Personalkosten, die beim Betrieb und der Wartung der eigenen Infrastruktur entstehen würden.
Welche Schritte notwendig sind, um Software für die Cloud zu entwickeln, welche Technologien existieren, und welche Probleme dabei auftreten können, ist auf den ersten Blick nicht sichtbar, für viele Unternehmen aber von großem Interesse. So auch für die Firma commercetools GmbH[1]. Die commercetools GmbH wurde 2006 gegründet und bietet seinen Kunden eine Software-as-a-Service eCommerce-Lösung an. Kunden wie Red Bull, Burton und Brita betreiben internationale Shop-Lösungen auf der geclusterten Technologie. Nun ist der mobile Markt auch im eCommerce stark am Wachsen. Der Verkauf von Produkten über Webseiten ist dafür allerdings keine adäquate Lösung. Aus diesem Grund wurde bei commercetools bereits eine native iPhone-Applikation entwickelt, mit der mitunter Produktkataloge angezeigt und Bestellungen angelegt werden können. Allerdings wurde dafür ein Backend benötigt, das die in der Applikation entstehenden Daten verwaltet. Dieses Backend soll in der Cloud laufen, um die bestehende Infrastruktur von commercetools nicht zusätzlich zu belasten, und auch die Anbindung anderer Frontends ermöglichen.
Daraus ergab sich die Forderung nach einer Architektur auf der Google App Engine, die ihre Dienste über einen Webservice zur Nutzung anbietet.

[1] `http://www.commercetools.de/`

1.2 Ziele der Arbeit

Im Rahmen dieser Arbeit soll auf Basis der Google Cloud-Services der Prototyp einer Multi-mandantenfähigen Architektur zur Verwaltung der Daten aus verschiedenen Shop-Frontends erstellt werden. Diese Frontends sollen über einen Webservice an ct.Box angebunden werden können. Diese Architektur wird im Folgenden mit ct.Box bezeichnet. Dabei sollen die einzelnen Schritte des Architekturentwurfs dokumentiert und die verwendeten Technologien evaluiert werden, um einen möglichst verständlichen Leitfaden zur Entwicklung einer Cloud-Architektur anzufertigen.
Da die Architektur primär für kleinere Händler mit reduzierten Anforderungen gedacht ist, die sich mittels weniger Schritte einen einfachen Shop aufsetzen wollen, soll sie später einmal in der Lage sein, bis zu 1000 Shops parallel zu betreiben. Der in dieser Arbeit entwickelte Prototyp muss diese Anforderung allerdings noch nicht erfüllen.
Nachdem es sich bei der Entwicklung dieser Arbeit um einen Prototypen handelt, ist die Performance von ct.Box zwar am Ende interessant, um den noch notwendigen Aufwand für die Optimierung der Applikation einzuschätzen, allerdings kein Kriterium für den Erfolg der Arbeit.

1.3 Gliederung

Nachfolgend wird im zweiten Kapitel dieser Arbeit der Begriff Cloud Computing näher erläutert, um ein Grundverständnis über die Funktionsweise der von Google angebotenen Services zu schaffen und einige Begrifflichkeiten zu klären. Ebenso werden das Modell der Multi-Mandanten-Fähigkeit (Multi Tenancy) und andere grundlegende Konzepte eingeführt, die beim Entwurf der Architektur eine wichtige Rolle spielen. Das dritte Kapitel beschreibt die Implementierung von ct.Box, jeweils mit eigenen Abschnitten für die einzelnen Schichten der Architektur. So beschreibt Kapitel 4.1 die Implementierung der Persistenzschicht, Kapitel 4.2 die Serviceschicht und Kapitel 4.3 die Präsentationsschicht von ct.Box. Anschließend daran findet sich ein Kapitel über die Sicherheitsmaßnahmen, die zur Absicherung von ct.Box und seiner Komponenten getroffen wurden.
Kapitel 5 beinhaltet das Fazit der Arbeit, mit der Evaluierung der entstandenen Architektur. Im abschließenden Kapitel wird schließlich die Arbeit noch einmal zusammengefasst und ein Ausblick auf die Zukunft von ct.Box und der Google App Engine gegeben.

2 Grundlagen

Beim Cloud Computing werden externe Infrastrukturen zum Speichern und Bearbeiten von Informationen genutzt. Der Anwender arbeitet also nicht mehr lokal, sondern über das Netzwerk auf seinen Daten oder mit Anwendungen. Das kann sogar soweit gehen, dass sich nicht mehr feststellen lässt, wo genau sich die eigenen Daten befinden. Der Begriff Cloud ("Wolke") soll genau diese Undurchsichtigkeit ausdrücken. Eine einheitliche Definition von Cloud Computing ist allerdings nur schwer zu finden. Das National Institute of Standards and Technology (NIST) der US-Regierung beschreibt es folgendermaßen:
"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [MG11]
In dieser Definition werden mehrere essentielle Eigenschaften des Cloud Computings beschrieben. Cloud Computing stellt keine neue Technologie dar, sondern ein neues Konzept, bei dem Dienstleistungen in Echtzeit über das Internet angeboten und nutzungsgerecht abgerechnet werden. Dabei können diese Dienste von einer Vielzahl von Endgeräten, sei es ein PC, PDA oder ein modernes Smartphone, über standardisierte Schnittstellen wie Webservices oder Browser weitestgehend ortsunabhängig aufgerufen werden.
Anwendungen in der Cloud sind hoch skalierbar. Da eine Cloud in der Regel aus vielen (bis zu mehreren tausend) Rechnern besteht, können Spitzenlasten durch das Hinzuschalten neuer Ressourcen dynamisch und bestenfalls automatisch abgefangen werden. Ebenso übernimmt der Cloud-Anbieter die Reduzierung der Rechenleistung, falls diese gerade nicht mehr benötigt wird. Die Fähigkeit von Cloud-Computing, Leistung nach Bedarf zu- und abzuschalten, wird als Elastizität bezeichnet, und als eine der essentiellen Eigenschaften von Clouds angesehen.[KE11] Durch den redundanten Betrieb mehrerer Rechenzentren können Cloud-Anbieter weiterhin eine hohe Verfügbarkeit ihrer Services garantieren. Eine Cloud bietet aber selbstverständlich nicht nur Vorteile.
Da die Standardisierung von Cloud-Computing im Moment noch nicht sehr weit fortgeschritten ist, tritt bei der Auswahl eines Cloud-Anbieters ein sogenannter "Lock-in Effekt" auf. Es ist danach nicht mehr ohne erheblichen Auf-

wand möglich, seine Daten und Anwendungen zu einem anderen Anbieter zu migrieren. Nachdem die Daten in der Cloud nicht mehr unter der Kontrolle ihrer eigentlichen Besitzer liegen, sehen viele Nutzer besonders bei der Sicherheit ihrer Informationen in der Cloud ein kritisches Problem. Wichtig ist hier vor allem, dass die Anbieter durch Maßnahmen wie Service-Level-Agreements Vertrauen bei den Anwendern schaffen. Es ist davon auszugehen, dass Cloud-Anbieter mindestens ebenso viel Aufwand zum Schutz der Daten ihrer Nutzer betreiben, wie es ein Unternehmen selbst würde, denn Datenverlust jedweder Art würde für sie einen enormen Image-, und damit auch Umsatz-Verlust bedeuten.

2.1 Klassifikation von Cloud-Computing

Die Cloud ist kein homogener Begriff. Zuerst lassen sich hinsichtlich dem Besitzer ,beziehungsweise Betreiber einer Cloud die sogenannten Private, Public, Hybrid und Community-Clouds unterscheiden. (Abbildung 2.1)

Public Cloud Die wohl am häufigsten vertretene Art des Cloud Computing sind die sogenannten Public Clouds. Eine Public Cloud zeichnet sich dadurch aus, dass die von ihr angebotenen Services der breiten Öffentlichkeit zugänglich gemacht werden. Die Daten verschiedener Anwender liegen bei diesem Modell vollständig beim Anbieter der Cloud, was bei vielen Firmen Bedenken wegen der Datensicherheit aufkommen lässt. Bezahlt werden bei diesem Modell nur die Ressourcen, die auch wirklich verbraucht werden (“pay-as-you-go”), es gibt allerdings auch kostenlose Angebote (z.B. Google Docs). [FE10]

Private Cloud Mit dem Begriff Private Cloud bezeichnet man Cloud Computing innerhalb eines private Netzwerks, wie zum Beispiel innerhalb eines Firmennetzwerks. Im Gegensatz zu den Public Clouds hat hier der Anwender die volle Kontrolle über seine Daten und ist natürlich auch selbst für die Sicherheit der eigenen Cloud zuständig. Hier wird also der Vorteil der flexibleren Kosten einer Public Cloud zu Gunsten größerer Kontrolle über die eigenen Daten aufgegeben. [FE10]

Hybrid Cloud Eine Hybrid Cloud kombiniert eine oder auch mehrere Public und Private Clouds. Hierbei können die Vorteile beider Modelle (kostengünstige Ressourcen und die Kontrolle über kritische Daten) genutzt werden. Die Schwierigkeit besteht allerdings darin, eine sinnvolle Aufteilung der Ressourcen zwischen Public und Private Clouds zu finden und Anwendungen übergreifend auf ihnen laufen zu lassen.[FE10]

Community Cloud Als letztes Modell wäre noch die Community Cloud zu nennen. Eine Community Cloud unterscheidet sich insofern von einer Public Cloud, dass ihre Services nicht für die gesamte Öffentlichkeit, sondern nur für einen eingeschränkten Nutzerkreis zur Verfügung stehen, der sich die Kosten für den Betrieb der Cloud teilt. Meistens schließen sich Institutionen mit denselben Anforderungen an die Cloud zusammen und kombinieren ihre Private Clouds zu einer Community Cloud, die sie gemeinsam nutzen um Kosten zu sparen. [MG11]

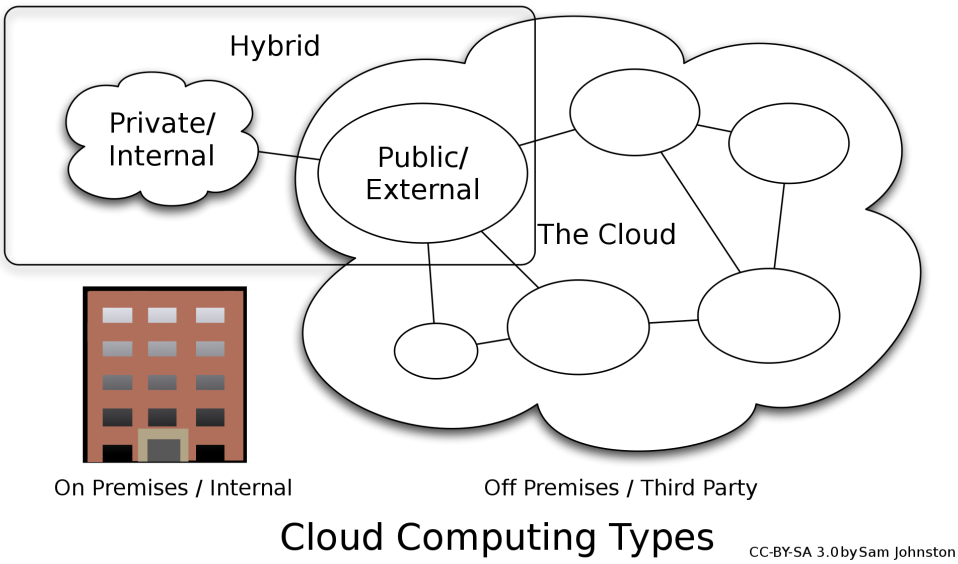


Abbildung 2.1: Cloud Computing Typen [Wikb]

Die Google App-Engine ist dabei den Public Clouds zuzuordnen, da sie ihre Dienste öffentlich anbietet. Das bedeutet für ct.Box, dass es zusammen mit Applikationen vieler anderer Entwickler auf der Infrastruktur von Google laufen wird.

2.2 Cloud-Services

Nicht nur die Typen einer Cloud können sehr unterschiedlich sein, sondern auch die von einer Cloud angebotenen Services. Man spricht je nach Art der angebotenen Dienstleistung von Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), oder Infrastructure-as-a-Service (IaaS).

Infrastructure-as-a-Service Beim IaaS werden direkt fundamentale Komponenten einer Rechnerinfrastruktur, wie zum Beispiel Speicher, CPU und Netzwerkkomponenten, virtualisiert als Service angeboten, auf denen der Anwender dann beliebige Software und sogar Betriebssysteme laufen lassen kann. Als Nutzer kann man sich bei hier komplett auf die Entwicklung der eigenen Anwendungen konzentrieren, da Betrieb und die Wartung der genutzten Infrastruktur vom IaaS-Provider übernommen werden. Als Beispiele für IaaS-Anwendungen wären Amazons EC2 [1] oder Rackspace[2]zu nennen. [MG11]

Platform-as-a-Service Wie der Name schon andeutet, bietet PaaS eine komplette Entwicklungsumgebung als Service zur Nutzung an. Nutzer dieser Services können mit den von den Providern unterstützten Werkzeugen Software entwickeln und auf deren Infrastruktur laufen lassen. Entwickler interagieren nur über die angebotenen API's mit der Umgebung, und haben keinen Zugriff auf die darunterliegende Infrastruktur, inklusive dem Betriebssystem. Die Google App-Engine [3], Windows Azure [4] und Force.com [5] sind Beispiele hierfür. [MG11]

Software-as-a-Service Hier werden dem Verbraucher konkrete Software-Dienstleistungen angeboten. Diese Software kann über verschieden Endgeräte ("Thin-Clients") genutzt werden und stellt meistens Standardsoftware wie E-Mail- oder CRM-Anwendungen dar. Die komplette Infrastruktur ist für den Nutzer transparent, die einzige Interaktion findet mit der Applikation selbst statt. Da die Software vom Anbieter verwaltet wird, werden zum Beispiel Updates selbstständig zentral eingepflegt und sind somit sofort für alle Nutzer verfügbar. Ein Beispiel für eine SaaS-Anwendung ist Google Apps [6]. Hier können Standard-Business Anwendungen wie E-Mail und Textverarbeitung im Browser verwendet werden. [MG11]

Während die App-Engine also PaaS-Dienste anbietet, stellt ct.Box selbst eine SaaS-Anwendung dar. Wie in der Definition gegeben, lassen sich die Services von ct.Box über verschiedene Endgeräte abrufen, wobei die internen Vorgänge für die Nutzer nicht sichtbar sind.

[1] http://aws.amazon.com/de/ec2/
[2] http://www.rackspace.com/
[3] http://code.google.com/intl/de-DE/appengine/
[4] http://www.microsoft.com/windowsazure/
[5] http://www.salesforce.com/platform/
[6] http://www.google.com/apps/intl/de/business/index.html

2.3 Multi-Tenancy

Multi-Tenancy (zu deutsch Multi-Mandantenfähigkeit) ist ein weiteres neues Konzept, das im Zusammenhang mit Cloud-Computing, insbesondere mit Software-as-a-Service, immer mehr an Bedeutung gewinnt. Multi-Tenacy bedeutet dabei im Grunde, dass nicht mehr für jeden Tenant (ein Nutzer der Software, z.B. eine Firma, eine Firmenabteilung, etc.) eine eigene separate Infrastruktur zur Verfügung gestellt wird, sondern sich die Nutzer die Ressourcen teilen. Die unterschiedlichen Nutzer arbeiten also auf einer einzigen Applikationsinstanz auf ihren Daten, bemerken dies aber nicht. Für den Nutzer sieht es so aus, als ob er auf einer Single-Tenant Architektur arbeiten würde, auf der ihm alle Ressourcen alleinig zur Verfügung stehen. Die Vorteile einer Multi-Tenant-Architektur liegen dabei vor allem in der hohen Skalierbarkeit. Außerdem können enorme Kosten gespart werden, wenn nicht mehr jeder Nutzer eine eigene Infrastruktur benötigt. Da nur eine Instanz der Applikation für alle Nutzern eingesetzt wird, ist die Software leichter zu warten und Updates können gleichzeitig eingespielt werden, was die Software sicherer macht.
Um einen reibungsfreien Betrieb einer solchen Architektur zu gewährleisten, muss sichergestellt sein, dass Nutzerdaten nur für den dazugehörigen Tenant sichtbar sind. Die Isolation der Daten kann dabei auf mehreren Wegen erreicht werden, und je nach Art der Realisierung spricht man von unterschiedlichen Ebenen der Multi-Mandantenfähigkeit.

Shared Machine Beim Shared Machine Ansatz teilen sich die Mandanten eine Applikationsinstanz und die komplette Rechenleistung. Für jeden Tenant wird jedoch eine eigene Datenbankinstanz angelegt, wodurch es möglich ist, auch für jeden Tenant ein eigenes Datenbankschema anzulegen, beziehungsweise ein bereits vorhandenes Schema zu erweitern. Durch das Speichern von Metadaten können die Datenbanken den Nutzern zugeordnet werden. Das Verhindern von Fremdzugriffen auf eine Tenant-Datenbank kann dabei durch die Sicherheitsmechanismen des Datenbanksystems übernommen werden. Im Falle eines Datenverlusts bietet diese Lösung weiterhin den geringsten Aufwand beim Wiederherstellen der Daten. Da in jeder Datenbank nur die Informationen eines Kunden gespeichert sind, muss auch nur ein System wiederhergestellt werden, die anderen Systeme bleiben davon unbeeinträchtigt. Die Kosten sind bei diesem Ansatz jedoch am höchsten, da die Anzahl der maximal möglichen gleichzeitig lauffähigen Datenbanken die Anzahl der Mandanten beschränkt und somit mehr Hardware benötigt wird, um hoch zu skalieren. Mehr Hardware bedeutet als Folge natürlich auch höhere Wartungskosten, weswegen dieser Ansatz meistens dann eingesetzt wird, wenn die erhöhte

Erweiterbarkeit einer einzelnen Datenbank pro Mandant benötigt wird, oder die strikte der Trennung der Datenbanksystem aus sicherheitsrelevanten Gründen notwendig ist. [CCW06]

Shared Process Beim Konzept der Shared Processes geht man einen Schritt weiter. Die Tenants teilen sich nicht nur die Hardware und Applikation, sondern auch eine Datenbankinstanz. Die Schemata der einzelnen Mandanten sind aber immer noch unterschiedlich, und können auch immer noch von ihnen erweitert werden. Das Wiederherstellen verlorener Daten gestaltet sich hingegen schwieriger.
Ein einfaches Zurücksetzen der gesamten Datenbank würde die Daten aller Mandanten überschreiben, weswegen häufig eine separate Backup-Datenbank erstellt wird, von der aus nur die beschädigten Tabellen ersetzt werden. Im allgemeinen ist der Shared Process Ansatz ein Kompromiss zwischen Skalierbarkeit und Sicherheit, zu geringeren Kosten als beim Shared Machine Modell, da die gleiche Anzahl an Servern eine größere Anzahl von Mandanten bedienen kann. [CCW06]

Shared Table Das dritte Modell sind die Shared-Tables. Wie der Name schon vermuten lässt, benutzen hier die Mandanten ein gemeinsames Schema innerhalb einer Datenbank. Dafür enthält jede Tabelle eine Spalte, die einen eindeutigen, genau einem Tenant zugeordneten Schlüssel beinhaltet. Dadurch kann bei sehr geringem Kostenaufwand besonders hoch skaliert werden. Es wird also Isolation aufgegeben, um an Leistung zu gewinnen. Nachdem die Datensätze unterschiedlicher Mandanten in einer Tabelle stehen, muss aber schon bei der Konzeption der Applikation besonderes auf die Integrität der Daten geachtet werden. Außerdem sind nutzerspezifische Erweiterungen der Schemata hier nicht mehr möglich, da sich alle Tenants das selbe Schema teilen. Nachdem im Fehlerfall einzelne Reihen ersetzt werden müssen, ist der Recovery-Aufwand bei den Shared Tables recht hoch, insbesondere falls eine Tabelle große Dimensionen besitzt. Die hohe Skalierbarkeit und die niedrigen Kosten machen diesen Ansatz für Applikationen interessant, die mit einer großen Anzahl von Nutzern rechnen. [CCW06]

Abbildung 2.2 veranschaulicht die verschiedenen Ebenen noch einmal.

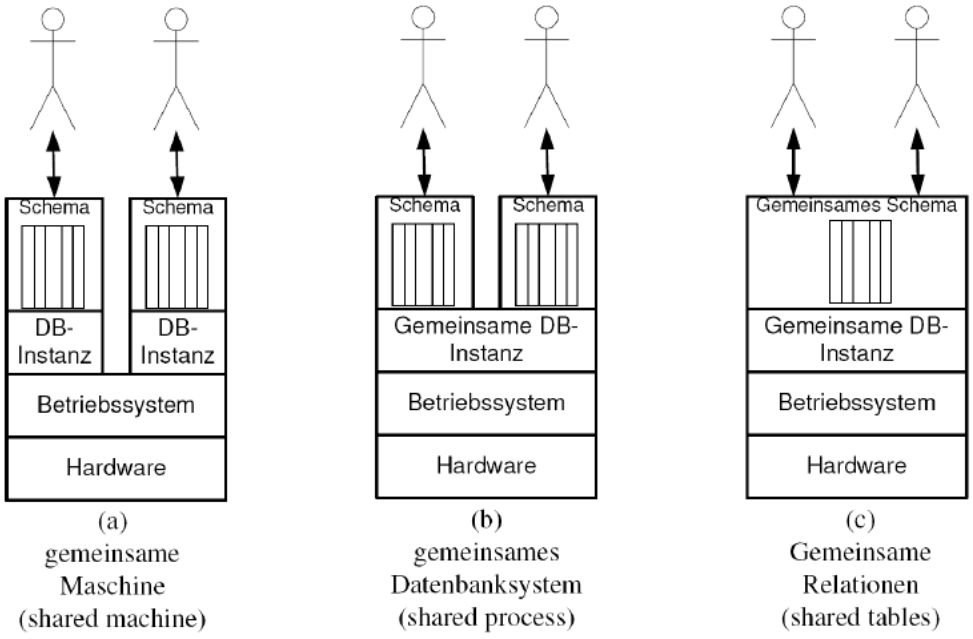


Abbildung 2.2: Die verschiedenen Stufen der Multi-Mandantenfähigkeit [KE11]

Als prominentes Beispiel für den Einsatz einer Multi-Mandanten-fähigen Architektur dient Salesforce. Dank Multi-Tenancy ist es Salesforce möglich, die gesamten auf Salesforce.com angebotenen Services für mehr als 55.000 Kunden (Stand 2009) weltweit auf nur 1.000 Servern laufen zu lassen. Dabei werden Daten im Bereich von Terabytes in 10 Datenbanken, die auf ungefähr 50 Servern betrieben werden, gespeichert. [Sch09]
Für ct.Box ist Multi-Tenancy eben wegen dieser hohen Skalierbarkeit interessant.

2.4 Google App-Engine

In diesem Abschnitt werden kurz die Möglichkeiten und Einschränkungen der Google App-Engine angesprochen. Diese sind für die Implementierung von ct.Box von großer Bedeutung.

2.4.1 Allgemeines

Die GAE erlaubt es Programmierern, ihre Anwendungen in drei verschiedenen Sprachen zu verfassen.

- Java[7] (und alle Sprachen die auf der Java VM aufsetzen, z.B. Skala, Groovy)
- Go[8] (experimental, zukünftige Änderungen an der API möglich)
- Python[9]

Die Wahl der Laufzeitumgebung bleibt also dem Entwickler überlassen. Unterschiede in der Performance auf den Servern von Google gibt es nicht, und auch die angebotenen Funktionen unterscheiden sich (von Go abgesehen) unter den verschiedenen Sprachen nur wenig. Das App Engine Software Development Kit (SDK) enthält für jede Laufzeitumgebung zudem einen lokalen Development-Server, durch den auf dem eigenen Rechner ein Webserver gestartet wird, der das Verhalten der GAE simuliert. Das ist vor allem für das lokale Testen einer Anwendung ausgesprochen nützlich, selbst wenn es noch kleine Unterschiede im Verhalten zwischen lokaler und globale Umgebung gibt. Als weiteres Hilfsmittel stellt Google ein Plugin für die weit verbreitete Entwicklungsumgebung Eclipse[10] zur Verfügung, welches das SDK und Funktionen zum Erstellen, Debuggen und Hochladen eines App Engine-Projektes beinhaltet. Dieses Plugin ist allerdings Java-Entwickler vorbehalten, mit dem Kommandozeilen-Programm "appcfg"[11] existiert jedoch ein laufzeitübergreifendes Tool zum Hochladen und Verwalten von GAE-Projekten.
Die von Google erbrachten Leistungen werden über die "Quotas"[12] abgerechnet, die alle 24 Stunden zurückgesetzt werden. Dabei werden jeder Applikation eine gewisse Anzahl von Ressourcen kostenlos zur Verfügung gestellt (z.B. 6.5 CPU-Stunden). Sobald man diese überschreitet muss entweder das "Billing" für die App aktiviert werden, oder die Applikation ist bis zum nächsten Reset der Quotas nicht erreichbar.

2.4.2 Die Sandbox-Umgebung

Als erste und wichtigste Einschränkung ist die sogenannte Sandbox-Umgebung zu nennen. Die Sandbox-Umgebung ist eine virtualisierte Laufzeitumgebung, in der alle Programme auf der Google App-Engine laufen. Durch die Sandbox

[7] http://java.sun.com/
[8] http://golang.org/
[9] http://www.python.org/
[10] http://www.eclipse.org/
[11] http://code.google.com/intl/de-DE/appengine/docs/java/tools/uploadinganapp.html
[12] http://code.google.com/intl/de-DE/appengine/docs/quotas.html

wird sichergestellt, dass die Programme von der darunterliegenden Hardware und dem Betriebssystem unabhängig sind und somit von Google ortsunabhängig auf ihre Rechenzentren verteilt werden können. Entwickler haben also keinen vollen Zugriff auf alle Möglichkeiten der Standardbibliotheken. Die Sandbox ist also einerseits eine wichtige Grundlage für die Skalierbarkeit einer Applikation, andererseits isoliert sie unterschiedliche Anwendungen voneinander, da jede Anwendung ihre eigene Umgebung erhält. Um diese Unabhängigkeit der Sandbox zu gewährleisten sind allerdings einige Einschränkungen nötig: [Goo11q]

- Andere Computer im Internet können nur über den von Google bereitgestellten UrlFetch-Service (im Gegensatz zu z.B. Java-Sockets) erreicht werden. Genauso kann eine Anwendung nur über das HTTP(S) - Protokoll von anderen Computern angesprochen werden.

- Es ist einer Anwendung nicht erlaubt auf das Dateisystem zuzugreifen. Um Dateien zu persistieren können der App-Engine Datastore oder die Memcache-Services verwendet werden. Generell können also nur Daten gelesen werden, die auch von der Anwendung erzeugt wurden.

- Anwendungscode wird in einem eigenen Prozess entweder als Antwort auf eine Web-Anfrage, auf einen sogenannten "qeued task", oder einen "scheduled task" ausgeführt und hat danach 30 Sekunden Zeit um seine Berechnungen auszuführen. Nach dieser Zeitspanne bricht die App-Engine den gesamten Prozess ab, um die Google-Server vor zu hoher Last zu schützen. Es ist deswegen auch nicht möglich eigene Prozesse oder Threads zu erzeugen.

Mit der "JRE Class White List"[13] existiert eine vollständige Liste der unterstützten Klassen der Java Standard Bibliothek. Außerdem steht eine Website[14] mit einer Liste unterstützter Bibliotheken und Frameworks zur Verfügung. Da sich die GAE in ständiger Weiterentwicklung befindet, werden einige dieser Einschränkungen aber mit der Zeit immer mehr aufgeweicht. Mit der Version 1.5 sind zum Beispiel Services dazugekommen, die es erlauben, langlebige Hintergrundprozesse zu starten und somit das 30 Sekunden-Limit für Anfragen außer Kraft setzen.

[13] `http://code.google.com/intl/de-DE/appengine/docs/java/jrewhitelist.html`

[14] `http://code.google.com/p/googleappengine/wiki/WillItPlayInJava`

2.4.3 Konzepte des Datastore

Nachdem der Zugriff auf das Dateisystem durch die Sandbox verhindert wird, gibt es in der Google App Engine nur eine Möglichkeit Daten persistent zu speichern: Den Datastore.
Der Google Datastore ist ein schemaloses, verteiltes, und objektorientiertes Datenbanksystem, das auf der Basis von Google's BigTable[15] aufgebaut ist.

Entities

Der Datastore hat also keine Tabellen wie bekannte relationale Datenbanken, sondern ist eher als eine verteilte Hashmap anzusehen, in der die Entities (Objekte) abgelegt werden und die ergänzend dazu Funktionen für Queries und Transaktionen auf ihren Daten anbietet. Ein Entity hat nun beliebig viele "Properties" (Attribute des Objekts), in denen Werte bestimmter Datentypen gespeichert werden können (z.B. Strings, Integers, oder eine Referenz auf ein anderes Entity), und einen Key (Schlüssel) der ein Entity im gesamten Datastore eindeutig identifiziert. Der Key wird von der App Engine selbstständig generiert und setzt sich aus mehreren Teilen zusammen: [Goo11h]

- Dem Objekttyp. Der Typ eines Entity's ist ein Name, den die Applikation dem Objekt gibt, um Kategorien für die Queries zu schaffen. Eine eCommerce-Applikation wie ct.Box wird also zum Beispiel Entities vom Typ Product oder Customer verwalten. Da der Datastore kein Schema besitzt, müssen zwei Objekte vom selben Typ aber nicht unbedingt auch die selben Properties besitzen. Falls ein solches Verhalten gewünscht ist, muss es von der Applikation sichergestellt werden.

- Einem "identifier", der entweder ein von der Applikation zugewiesener Name sein kann, oder ein vom Datastore generierter numerischer Wert.

- Optional aus einem "path" (Pfad), der eine eventuelle Parent-Child-Relationship ausdrückt, ein Entity also als Kind eines anderen Entity's deklariert. Durch diesen Pfad werden sogenannte "Entity-Groups", eine Hierarchie von Entities, festgelegt, die bei den Transaktionen eine wichtige Rolle spielen. Es können nämlich nur Entities einer Entity-Group gemeinsam in einer Transaktion verwendet werden.

- Dem Namespace, in dem ein Entity abgelegt werden soll. Auf die Namespaces-API wird in Abschnitt 2.3 näher eingegangen.

[15] `http://labs.google.com/papers/bigtable.html`

Sobald einem Entity ein Key zugewiesen wurde, kann dieser nicht mehr verändert werden. Aus der Sicht relationaler Datenbanksystem stellt also der Entity-Typ eine Tabelle dar, in der in jeder Zeile ein Entity mit seinen Properties als Spalten liegt.
Die Objekte werden im Datastore lexikographisch nach ihren Keys sortiert und gespeichert, Entities in einer Entity-Group liegen dadurch nah beieinander, weswegen bei Abfragen auf eine Gruppe weniger Server kontaktiert werden müssen. Wichtig zu erwähnen ist, dass, auch wenn mehrere Operationen in einer Transaktion laufen können, jede Lese- und Schreiboperation auf einem Entity atomar ist, also entweder komplett erfolgreich abgeschlossen, oder im Fehlerfall als Ganzes abgebrochen wird. Außerdem kann ein Entity, dass gespeichert werden soll, nicht größer als ein Megabyte sein. Das ist eine Einschränkung, die meistens größere Anpassungen im Code nach sich zieht , da hierdurch die verschachtelte Speicherung von Objekten nur begrenzt möglich ist. Ein Beispiel dafür wird in Abschnitt 4.1.2 gegeben.

Storage Options

Google bietet Entwicklern beim Erstellen einer Applikation 2 Versionen seines Datastore an: [Goo11b]

Der **Master/Slave** Datastore repliziert geschriebene Daten durch ein Master-Slave-System asynchron im Hintergrund. Da immer nur ein Server der Master für eine Schreiboperation sein kann, ist bei dieser Option starke Konsistenz für alle Leseoperationen garantiert. Außerdem sind hier die Kosten für CPU und Speicherplatz am geringsten. Ein Nachteil beim Master/Slave Datastore ist allerdings, dass im Fall von Wartungen oder Problemen Server temporär nicht erreichbar sein können.

Beim **High Replication** Datastore werden die Daten über ein von Google entwickeltes System auf Basis des Paxos-Algorithmus[16] auf die verschiedenen Datencenter verteilt. Dadurch wird eine hohe Verfügbarkeit sowohl für Lese-, als auch für Schreiboperationen sichergestellt. Das geschieht allerdings auf Kosten der Schreiboperationen und es kann auch nur noch eventuelle Konsistenz für Queries garantiert werden. Die abgerechneten Kosten (CPU-Zeit, Speicherplatz) sind zudem ungefähr drei mal höher als beim Master/Slave Datastore.

Als Standard-Wert ist für jede neue Applikation der High Replication Datastore eingestellt. Ob die höheren Kosten für die erhöhte Verfügbarkeit gerechtfertigt sind, ist von der jeweiligen Applikation abhängig. Für ct.Box wird

[16] `http://labs.google.com/papers/paxos_made_live.html`

der High Replication Datastore verwendet, da zum Einen damit zu rechnen ist, dass die Anzahl der Schreiboperationen um ein Vielfaches niedriger sein wird als die der Leseoperationen, zum Anderen die Verfügbarkeit des Backends ein kritischer Faktor für den Betrieb der darauf aufbauenden Shops ist und ein Ausfall einen hohen finanziellen Verlust bedeuten würde.

Performance & Einschränkungen

Google selbst gibt an, dass BigTable, und damit auch der GAE Datastore ein hoch verfügbares, hoch skalierbares System mit einer Speicherkapazität bis in die Petabytes sei.[CDG+06] Nachdem BigTable von Google selbst in vielen Produkten (z.B. Google Earth[17], Google Finance[18]) eingesetzt wird, ist dieser Aussage durchaus Glauben zu schenken. Durch das automatische Verteilen der Daten werden die Writes performant gemacht. Leseoperationen skalieren ebenfalls von selbst, da der Datastore nur solche Abfragen zulässt, die mit der Anzahl Ergebnisse skalieren, und nicht mit der Größe der Datenbasis. Das bedeutet, dass zum Beispiel eine Anfrage an den Datastore, die 100 Entities als Ergebnis liefert, immer genau gleich lang benötigt, unabhängig davon ob im Datastore über mehrere Hundert oder mehrere Millionen Datensätze gesucht wird. Dadurch ergeben sich allerdings auch Einschränkungen. Um Queries auf dem Datastore überhaupt verfügbar zu machen, sind vorgefertigte Indizes notwendig. Das ermöglicht nicht ganz so umfangreiche Abfragen wie bei SQL-fähigen Datenbanken. Der Datastore unterstützt zum Beispiel keine Join-Operationen. Außerdem können Filter mit einem Ungleichheits-Operator nicht auf mehrere Properties eines Entity angewandt werden und auch Ergebnisse einer Unteranfrage können nicht als Filter verwendet werden. [Goo11e] Die App Engine beinhaltet dabei schon Indizes für Queries über Entities für einen Typ, Queries mit Filtern und Sortierung auf einzelne Properties und Gleichheitsfilter auf eine beliebige Anzahl von Properties. Für kompliziertere Anfragen müssen die Entwickler selbst Indizes definieren. Es gibt weiterhin keine Möglichkeit, eine Teilmenge der Felder eines Objekts aus dem Datastore zu laden. Bei einer Leseoperation kann man sich entweder das gesamte Entity, oder nur seinen Schlüssel zurückgeben lassen.

2.4.4 Multi-Tenancy

Die Google-App-Engine erlaubt es seit Version 1.3.6 mit Hilfe der Namespaces-API mulitmandantenfähige Applikationen zu erstellen. Der Datastore wird dafür

[17] `http://www.google.de/intl/de/earth/index.html`
[18] `http://www.google.com/finance`

nach dem Shared-Table-Ansatz aufgeteilt, es teilen sich also alle Mandanten eine Datenbank und auch dasselbe Schema (im Kontext des schemalosen Datastores also dieselben Entity-Typen). Dazu wird zusätzlich zu den in 2.4.3 erwähnten Informationen der aktuell gesetzte Namespace (eine eindeutiger Bezeichner für jeden Mandanten, im Fall von ct.Box ein String der aus dem Namen des Shops generiert wird) in die Schlüssel der Entities hineincodiert. Der Namespace muss also global gesetzt sein, was durch den *NamespaceManager* erreicht wird. Das Einarbeiten in den Schlüssel geschieht dann automatisch bei der Verwendung von APIs, die Namespaces unterstützen. [Goo11l] Im Moment sind das:

- der Datastore
- der Memcache
- die Task queue

Es ist aber auch möglich, den Namespace für einzelne Operationen zu setzen. Hier ein Beispiel für die Verwendung des NamespaceManagers: [Goo11l]

```
// Set the namepace temporarily to "abc"
String oldNamespace = NamespaceManager.get();
NamespaceManager.set("abc");
try {
  ... perform operation using current namespace ...
} finally {
  NamespaceManager.set(oldNamespace);
}
```

Listing 2.1: Einmaliges Setzen des Namespace frame

Das Einbinden von Multi-Mandanten-Fähigkeit in eine Applikation bedeutet also keinen großen Aufwand und ist auch schnell implementiert. Da eine Applikation aber generell Zugriff auf jeden existierenden Namespace hat, muss innerhalb von ct.Box sichergestellt sein, dass keine Daten-Lecks zwischen Namespaces existieren.

2.5 REST

REST ist ein Architekturprinzip, nach dem unter anderem das World Wide Web aufgebaut ist. Es basiert auf einer zustandslosen Client-Server-Architektur, deren Komponenten über einheitliche Schnittstellen miteinander kommunizieren, und das in hierarchische Schichten unterteilt ist. [Fie00] Die zentrale Komponente in REST ist dabei die *Ressource*. Eine Ressource abstrahiert jede Art

von Information, die über einen Namen identifiziert werden kann. So kann zum Beispiel ein Bild oder ein Dokument, aber auch eine Ansammlung anderer Ressourcen eine Ressource darstellen. Kurz gesagt kann alles, das als Ziel eines Hyperlinks angegeben werden kann, unter der Definition des Begriffs "Ressource" zusammengefasst werden.[Fie00] Um eine bestimmte Ressource zu identifizieren, verwendet REST einen eindeutigen Identifikator ("resource ientifier"), durch den Ressourcen referenziert werden können. Im Kontext des Internets ist dieser Identifier zum Beispiel die URL eines statischen Dokuments.
Eine Ressource stellt allerdings keine eigene Entität dar, sondern nur eine Membership-Funktion, die auf eine Menge von Werten verweist. Diese Werte können nun entweder wieder selbst Ressource-Identifikatoren, oder sogenannte "resource representations", Repräsentationen des gegenwärtigen Zustands einer Ressource, sein.[Fie00] Die Repräsentation einer Ressource ist dabei abstrakt dargestellt als eine Byte-Folge mit den dazugehörigen Metadaten, die die Byte-Folge genauer beschreiben. Die Komponenten in REST interagieren also nicht mit Ressourcen, sondern mit deren Repräsentationen. Als Beispiel kann eine Ressource, die einen Kreis beschreibt, durch eine Repräsentation dargestellt werden, die diesen Kreis durch seinen Mittelpunkt und Radius im SVG-Format definiert. Eine andere Repräsentation kann den selben Kreis aber auch durch drei beliebige Punkte auf seiner Kreislinie in einer Komma-separierten Liste beschreiben. [Wikc] Je nach Art der Anfrage an den Server wird dann die passende Repräsentation für die Antwort herausgesucht.
Operationen auf den Ressourcen werden in REST nur durch die vier in HTTP 1.1 definierten Methoden GET, PUT, POST und DELETE [F^+] ausgeführt. Jeder Operator hat dabei seine eigene, auf CRUD gemappte, Bedeutung (siehe Tabelle 2.1).

HTTP	CRUD
GET	Read
POST	Create, Update, Delete
PUT	Create, Update
DELETE	Delete

Tabelle 2.1: CRUD-äquivalente Methoden für HTTP

Jede Ressource implementiert die dadurch definierte, einheitliche Schnittstelle. Die Semantik der einzelnen Methoden ist allerdings für jede Ressource unterschiedlich, wie auch in Abschnitt 4.3.1 gezeigt. Ein Aufruf per HTTP-GET

auf */rest/products* liefert eine Liste aller Produkte zurück, während ein GET auf */rest/products/1234* das Produkt mit der Id 1234 zurückgibt.
Für die Interaktion zwischen Komponenten und den Zugriff auf die Ressourcen sind bei REST die Konnektoren ("Connectors") zuständig. Dabei existieren unterschiedliche Arten von Konnektoren, wie zum Beispiel ein Client-Connector, der Requests an den Server senden kann, oder ein Server-Connector, der auf Verbindungen wartet und auf Anfragen antwortet.[Fie00] Dabei kann jeder dieser Konnektoren unterschiedlich implementiert werden. Ein Server-Connector könnte also zum Beispiel Anfragen über das HTTP-Protokoll entgegennehmen. Ein Client der mit diesem Server kommunizieren will, muss also auch einen HTTP-Konnektor besitzen, der enstprechende Anfragen formulieren kann.
Das letzte REST-eigene Konzept sind die Komponenten. Auch hier werden wieder mehrere Typen von Komponenten definiert. Diese sind in Tabelle 2.2 zusammen mit einigen Beispielen dargestellt.

Komponente	Beispiele
Origin Server	Apache httpd, Microsoft IIS
Gateway	Squid, CGI, Reverse Proxy
Proxy	CERN Proxy, Netscape Proxy, Gauntlet
Resolver	bind (DNS lookup library)
User Agent	Netscape Navigator, Lynx, MOMspider

Tabelle 2.2: REST Komponenten nach [Fie00]

Jede Komponente nimmt ihre eigene Rolle ein.
Ein User Agent schickt beispielsweise über einen Client-Connector Anfragen an den Server, und ist letztendlich auch der Empfänger der zurückgesendeten Antwort. Das bekannteste Beispiel für einen User-Agent ist dabei ein Web-Browser, der Requests sowohl versenden, als auch deren Antworten darstellen kann.[Fie00]
Auf Server-Seite ist der Origin-Server der Empfänger aller Nachrichten. Nur über den Origin-Server ist der Zugriff auf die Repräsentationen von Ressourcen möglich, und dieser Zugriff geschieht über ein generisches, transparentes und hierarchisch aufgebautes Interface.
Gateways und Proxies sind vermittelnde Komponenten, die entweder vor den Server oder den Client geschaltet werden, um zum Beispiel die Sicherheit zu verbessern.
Wie bereits erwähnt, ist REST eine zustandslose Architektur. Das bedeutet das jede Anfrage genug Informationen beinhalten muss, um vom jeweiligen Connec-

tor verstanden zu werden. Dadurch können auf Serverseite Speicherressourcen gespart werden, was allerdings auf Kosten der Größe der einzelnen Anfragen geht. Es entsteht also eine höhere Netzlast. Allerdings können so Anfragen auch parallel bearbeitet werden, und auch die Zuverlässigkeit eines Dienstes steigt, da beim Wiederanlauf nach einem Fehler kein vorheriger Zustand berücksichtigt werden muss. [DDM06] REST-Services skalieren daher sehr gut, was am Beispiel des enorm schnell wachsenden World Wide Web gut zu erkennen ist. Für ct.Box wird der Webservice deswegen nach REST aufgebaut. Nutzern soll es zudem möglichst einfach gemacht werden, eigene Clients für den Webservice zu entwickeln. Mit REST ist jeder Client, der HTTP-Anfragen formulieren kann, ein potentieller Nutzer der Services von ct.Box. Außerdem sind die hohe Skalierbarkeit und Zuverlässigkeit stichhaltige Argumente für den Einsatz von REST.

2.6 OAuth

Zuerst einmal definiert das OAuth-Protokoll verschieden Rollen beim Authentifizierungsprozess: [HLRH11]

Resource Owner Der Resource Owner ist der Besitzer einer geschützten Ressource. Damit ist es ihm möglich anderen den Zugriff auf diese Ressource zu gewähren.

Resource Server Auf dem Resource Server sind die geschützten Ressourcen abgelegt, und über diesen Server wird auf sie zugegriffen. Bei ct.Box sind das die Server der GAE.

Client Ein Client ist eine Applikation, die im Namen des Resource Owners Zugang zu dessen Ressourcen erhalten möchte. Die Äquivalente Klasse in ct.Box ist die *Application*-Klasse. Sie stellt eine Applikation dar, die im Namen eines *Shops* auf den Webservice zugreifen darf.

Authorization Server Der Authorization Server gibt Access Tokens an die Clients aus, sobald diese vom Resource Owner dazu autorisiert wurden. Dies sind für ct.Box ebenfalls die Server der GAE.

Das klassische Client-Server Authentifizierungsmodell sieht nun vor, dass ein Client Zugriff auf seine geschützten Ressourcen erhält, indem er sich beim Server anhand seiner Zugangsdaten autorisiert. Soll aber einer fremden Partei der Zugriff auf diese Ressourcen gewährt werden, muss der Client seine vertraulichen Daten an diese weitergeben. Die Drittpartei muss diese Daten

weiterhin speichern, um sie auch in Zukunft weiterhin nutzen zu können. Dadurch ergeben sich Sicherheitsprobleme, zum Beispiel bei einem Datenverlust der Drittpartei. Diese Partei hat außerdem vollen Zugriff auf alle geschützten Ressourcen des Clients und der Besitzer dieser Ressourcen keine Möglichkeit diesen Zugriff feingranular einzuschränken. [HLRH11]
OAuth löst diese Probleme, indem es den Client vom Resource Owner trennt. Nach dem OAuth-Protokoll richtet ein Client immer noch Anfragen an einen Resource Server, die Kontrolle über die Ressourcen liegt allerdings beim Resource Owner. Das wird dadurch erreicht, dass den Clients eigene Zugangsdaten in Form eines *Access Tokens* zugewiesen werden. Ein Access Token ist dabei ein einfacher String, in den jedoch Informationen wie seine Gültigkeitsdauer, oder sein Gültigkeitsbereich hineincodiert werden können. Diese Access Tokens erhalten die Clients mit der Zustimmung des Resource Owners vom Authorization Server und können sie dann verwenden, um Zugriff auf die Ressourcen zu erhalten. [HLRH11]

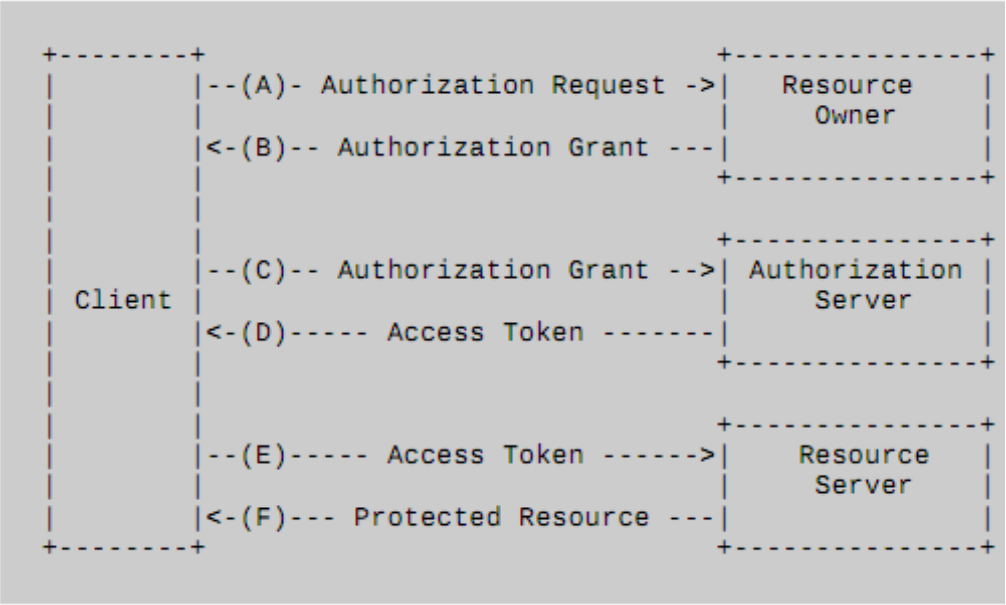


Abbildung 2.3: Die Abläufe im OAuth-Protokoll [HLRH11]

Der Ablauf des OAuth-Protokoll aus Abbildung 2.3 sieht nun wie folgt aus: [HLRH11]

A Der Client erfragt entweder direkt vom Resource Owner oder über einen Authorization Server die Autorisierung.

B Der Client erhält die genehmigte Autorisierung durch einen von vier *grant types*, auf die später noch eingegangen wird. Die Art dieses *grant types* hängt dabei von der Anfrage des Clients und den vom Authorization Server unterstützten Typen ab.

C Mit der gerade erhaltenen Autorisierung kann der Client nun vom Authorization Server einen Access Token anfordern.

D Der Authoritzation Server validiert die erhaltene Autorisierung und sendet einen Access Token zurück, falls die Validierung gelingt.

E Der Client verwendet den Access Token dazu, sich beim Resource Server zu authentifizieren und dort Ressourcen anzufordern.

F Der Resource Server überprüft den Access Token, und liefert die angeforderte Ressource zurück, falls der Token gültig ist.

OAuth unterscheidet weiterhin zwischen verschiedenen Client-Typen, wie zum Beispiel nativen Applikationen und Web-Applikationen. In ct.Box wird diese Unterscheidung aber nicht getroffen, sondern alle Applikationen (*Applications*) gleich behandelt. Dazu sind, wie bereits erwähnt, vier *grant types vorhanden*.

Authorization Code Beim Authorization Code Verfahren geschieht die Autorisierung in zwei Schritten. Der Client leitet den Resource Owner zuerst auf den Authorization Server und gibt eine URL an, zu der er nach der Autorisierung zurückgeleitet werden will. Ist die Autoriseriung durch den Resource Owner (zum Beispiel durch die Eingabe von Benutzername und Passwort in ein HTML-Formular) erfolgreich, so wird an diese URL als Parameter ein *authorization code* angehängt, den der Client empfängt. Mit diesem Code kann der Client im zweiten Schritt einen Access Token vom Authorization Server verlangen. [HLRH11]

Implicit Grant Der Implicit Grant ähnelt dem Authorization Code Verfahren sehr. Der einzige Unterschied besteht darin, dass nicht mehr 2 Requests gesendet werden müssen, um einen Access Token zu erhalten. Der Client erhält in diesem Fall den Access Token schon als Ergebnis der Autorisierungsanfrage. [HLRH11]

Resource Owner Password Credentials Bei diesem grant type vertraut der Resource Owner dem Client so sehr, dass er diesem die eigenen Zugangsdaten überlässt. Mit diesen Daten kann der Client dann beim Authorization Server einen Acces Token anfragen. [HLRH11]

Client Credentials In diesem Fall verwendet der Client eigene Zugangsdaten, um einen Access Token abzuholen. [HLRH11]

Allerdings müssen sich auch in ct.Box, wie in OAuth, Clients zuerst einmal beim Authorizationserver registrieren, um von diesen identifiziert werden zu können. Das geschieht in ct.Box im UserPanel, in dem ein *ShopAdmin* für seinen Shop eine neue *Application* anlegen kann. Dabei wird automatisch ein Access Token für diese Applikation generiert und im UserPanel angezeigt, so dass der *ShopAdmin* diesen an einen Außenstehenden weitergeben kann, der in seinem Namen Zugriff auf den Webservice von ct.Box erhalten soll. Damit weicht ct.Box ein wenig vom strengen OAuth-Protokoll ab.
OAuth sieht bei der Autorisierung bei jedem der vier *grant types* vor, dass sich ein Client mit Zugangsdaten des Resource Owners vom Authorization Server den Access Token abholt. Im Fall von ct.Box holt sich aber der Resource Owner einen Access Token ab, und gibt diesen an einen Client weiter. Für diese Art der Autorisierung ist im OAuth-Protokoll zwar kein *grant type* vorgesehen, durch die Weitergabe des Tokens an den Client ist aber die Autorisierung implizit geschehen.
Als letztes definiert OAuth unterschiedliche Typen von Access Tokens. In der Spezifikation werden dabei zwei Typen angegeben, es können aber auch eigene Typen definiert werden.

MAC -Tokens benutzen ähnlich zur HTTP-Basic Authentifizierung einen Identifier und ein Passwort. Allerdings wird das Passwort niemals mit Requests übertragen, sondern dazu verwendet, mit Hilfe einer Hash-Funktion (z.B. SHA-1) einen MAC-String zu berechnen, der stattdessen an den Server gesendet wird. [HLBA11] In diesen String werden noch zusätzliche Funktionen hineincodiert, zum Beispiel ein anderer, zufällig generierter String (*nonce*), um Replay-Angriffe[19] zu verhindern. Da der Server das Passwort kennt, mit dem der MAC-Token generiert wurde, kann er diesen entschlüsseln und auf Gültigkeit prüfen. Bei den MAC-Tokens handelt es sich also um ein symmetrisches Verschlüsselungsverfahren.

Bearer -Tokens verwenden im Gegensatz zu MAC-Tokens keine kryptographischen Methode, um festzustellen, ob der Besitzer des Tokens diesen auch wirklich verwenden darf. Bearer-Tokens sind so definiert, dass derjenige, der einen solchen Token besitzt, damit schon alle Rechte zum Zugriff auf geschützte Ressourcen hat. Hiermit können Bearer-Tokens beliebige Strings sein, die auf dem Server abgeglichen werden, oder aber auch codierte Informationen. Das bleibt den Anwendungen selbst überlassen.

[19] `http://de.wikipedia.org/wiki/Replay-Angriff`

Ist man auf mehr Sicherheit bedacht, sollten auf jeden Fall die MAC-Tokens verwendet werden. Allerdings verursachen diese sowohl auf dem Server beim Entschlüsseln, als auch beim Client beim Verschlüsseln der Tokens, für jede Anfrage mehr Aufwand. Um diesen für die Entwicklung von Applikationen für ct.Box so gering wie möglich zu halten, werden jedoch Bearer-Tokens verwendet.

3 System-Entwurf

Wie bereits im einleitenden Kapitel beschrieben wurde, ist ct.Box eine Applikation zum Speichern und Abrufen von e-Commerce relevanten Daten, wie zum Beispiel Produkt- und Bestelldaten. Nutzern von ct.Box soll dabei auf zweierlei Art Zugriff auf diese Funktionen gewährt werden: Zum Einen über einen Webservice, um den Zugriff vieler verschiedener Endgeräte auf den Service zu ermöglichen. Zum Anderen über eine Website, um die Verwaltung der hinterlegten Daten so einfach und komfortabel wie möglich zu gestalten. Die weiteren Anforderungen werden im folgenden Abschnitt beschrieben.

3.1 Anforderungen

Hier werden kurz die Anforderungen an ct.Box beschrieben.
Grundsätzlich soll ct.Box zwei Weboberflächen besitzen. Das *AdminPanel*, das commercetools Mitarbeiter zur Verwaltung der registrierten Shops nutzen können, und das *UserPanel*, das zur Pflege der Shops durch die Shop-Betreiber gedacht ist. Zusätzlich dazu sollen die Funktionen, die einem Shop-Betreiber im UserPanel zur Verfügung stehen, auch über einen Webservice verwendbar sein.

3.1.1 Funktionale Anforderungen

- Im AdminPanel sollen sich alle registrierten Shops, sowie Log-Dateien über deren Aufrufe an den Webservice anzeigen lassen. Weiterhin soll es Administratoren im AdminPanel möglich sein, neue Shops anzulegen, sowie vorhandene Shops zu bearbeiten und zu sperren. Bei einer Sperrung sollen dem Shop alle Funktionen von ct.Box verwehrt werden.
- Shop-Betreiber sollen im UserPanel Produkte, deren Varianten, Collections, sowie Bestellungen und registrierte Applikationen verwalten können. Unter “verwalten” sind hierbei die CRUD-Operationen zu verstehen.
- Über den Webservice sollen ebenfalls die CRUD-Operationen des UserPanels verfügbar sein. Die Kommunkation mit dem Webservice soll dabei über JSON-Objekte erfolgen.
- Der Webservice soll nach dem REST-Prinzip aufgebaut werden.

- Jeder Nutzer (Shop-Betreiber) von ct.Box soll eine eigene Subdomain erhalten. Aufrufe an ct.Box erfolgen dann nach dem Schema `https://<Shop>.ctbox.appspot.com`.

3.1.2 Nichtfunktionale Anforderungen

- Die Architektur sollte hoch skalierbar sein. Dafür sorgt die automatische Lastverteilung der GAE, die die Anzahl der laufenden Instanzen einer Applikation je nach Bedarf regelt.

- ct.Box soll Multi-Mandantenfähig sein, also mehrere Nutzer durch nur eine Applikationsinstanz bedient werden.

- Sowohl der Webservice als auch die Web-Oberfläche sollen vor unautorisierten Zugriffen geschützt werden. Unter den Bereich Sicherheit fällt auch die Isolation der Mandanten-spezifischen Daten. Dazu soll die Authentifizierung mit dem Webservice nach dem OAuth-Protokoll mit Bearer-Tokens erfolgen und alle Aufrufe an ct.Box HTTPS verwenden.

- Beide Weboberflächen sollen mit dem Google-Web-Toolkit (GWT) realisiert werden.

3.2 Use-Cases

Die Abbildungen 3.1, 3.3 und 3.2 stellen eine Übersicht über die Anwendungsfälle von ct.Box dar. In den Tabellen 5.1, 5.2, 5.3, 5.4 und 5.5 im Anhang sind die Use-Cases anhand der Produkt-Ressource dargestellt. Für Collections, Varianten, Applications und Bestellungen stellen sich die Anwendungsfälle analog dar. Die Use-Cases der Shops entsprechen ebenfalls dem hier gezeigten Schema, mit dem einzigen Unterschied, dass es nicht vorgesehen ist, Shops dauerhaft zu löschen, denn die Bestelldaten der einzelnen Shops müssen aus rechtlichen Gründen über längere Zeit aufbewahrt werden. Aus dem selben Grund können auch Bestellungen nicht direkt gelöscht werden.

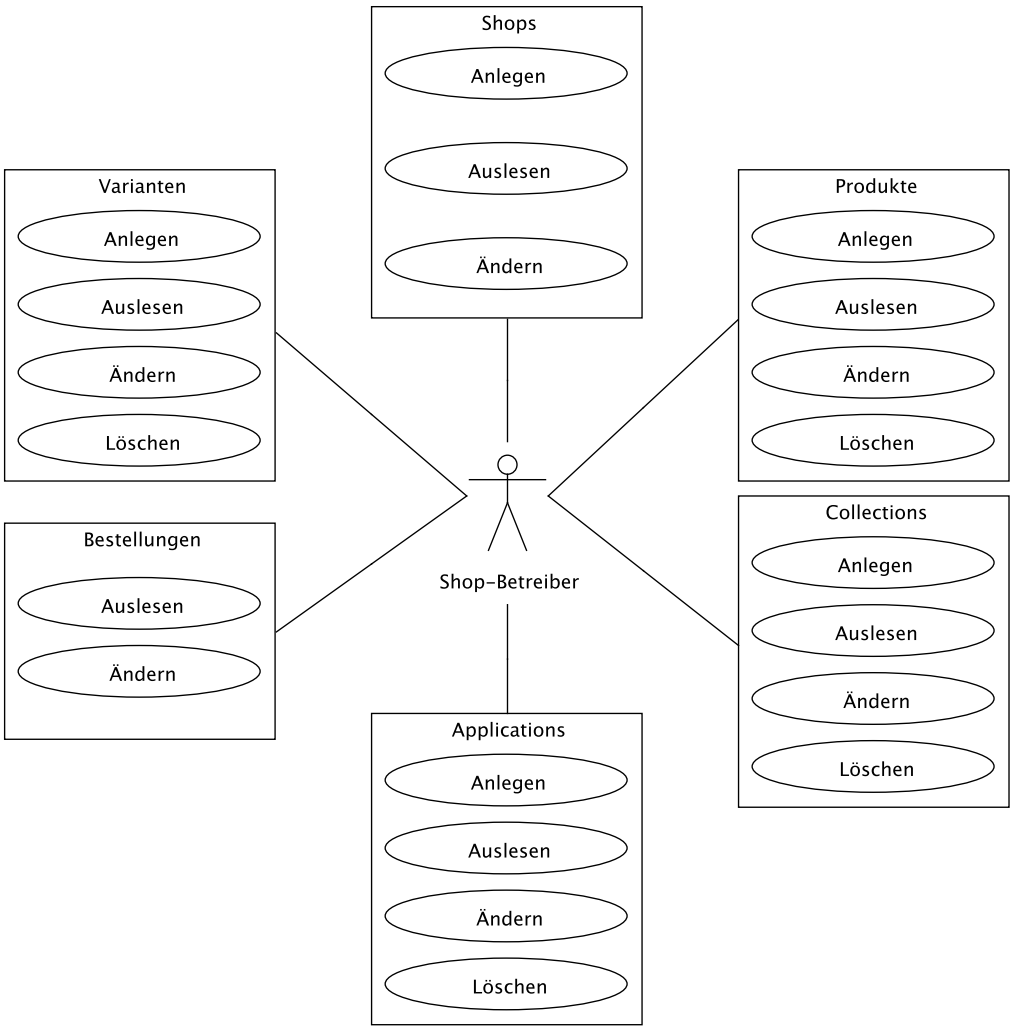


Abbildung 3.1: Anwendungsfälle für die Nutzer von ct.Box

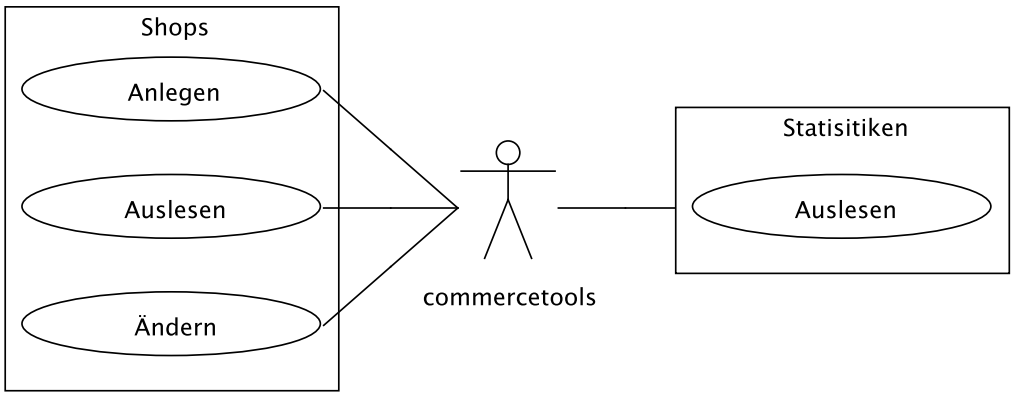


Abbildung 3.2: Anwendungsfälle für die Verwaltung von ct.Box

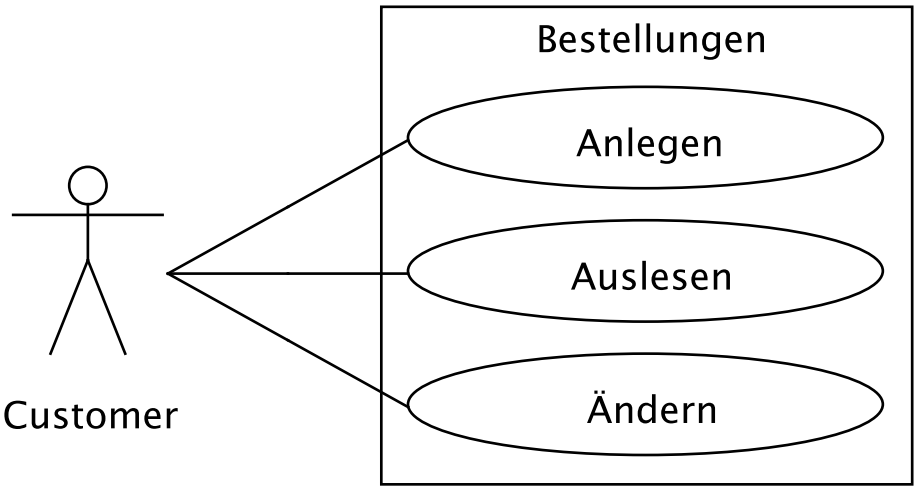


Abbildung 3.3: Anwendungsfälle für die Verwaltung von ct.Box

3.3 Komponenten

In Abbildung 3.4 sind die Pakete von ct.Box mit ihren Import-Beziehungen im Diagramm zu sehen.

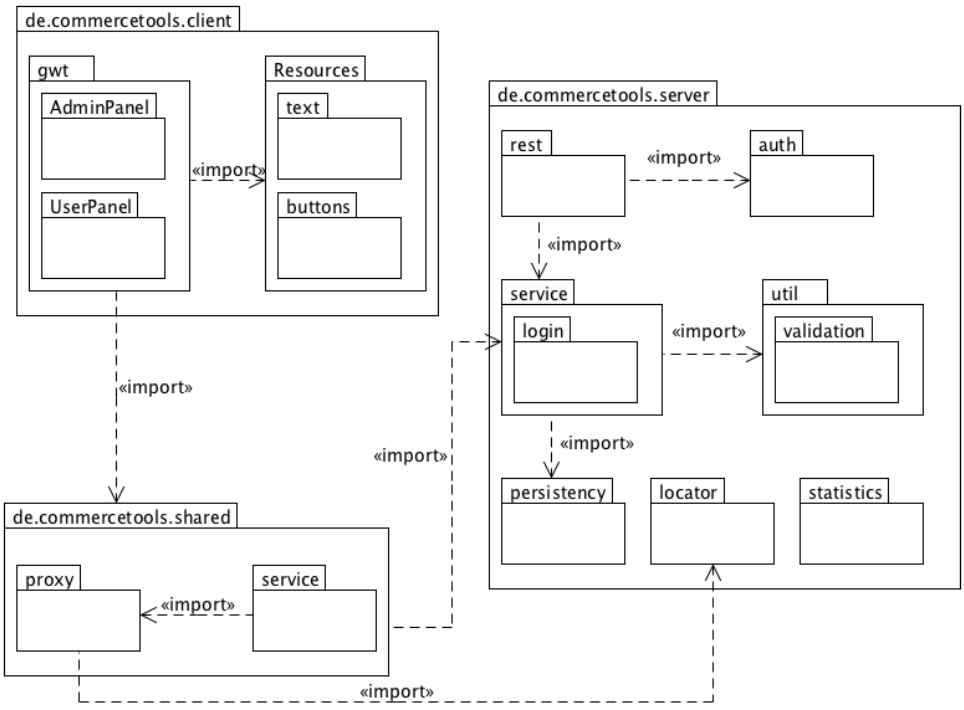


Abbildung 3.4: Pakete von ct.Box

Abbildung 3.5 zeigt eine detailliertere Ansicht des Server-Pakets.

Die Konzeption von ct.Box erfolgt nach dem 3-Schichten-Modell, wie in Abbildung 3.6 noch einmal hervorgehoben ist.

Die Präsentationsschicht In der Präsentationsschicht (Presentation-Layer) findet die Interaktion einer Applikation mit den Benutzern statt. Diese Schicht stellt den Benutzern ein User-Interface bereit um Anfragen an ct.Box zu senden und um die zurückgegebenen Daten zu visualisieren. Bei ct.Box besteht die Präsentationsschicht aus zwei Komponenten:

- Die Client-Komponenten enthält die Funktionen für das Web-Interface
- In der REST-Komponente sind die Klassen des Webservices enthalten

Die Applikationsschicht Darunter liegt die Applikationsschicht (Service-Layer), die die Business-Logik der Applikation enthält. Beide Komponenten der Präsentationsschicht greifen dabei auf die selben Services zu. Die Applikationsschicht koordiniert die Kommunikation zwischen Präsentationsschicht

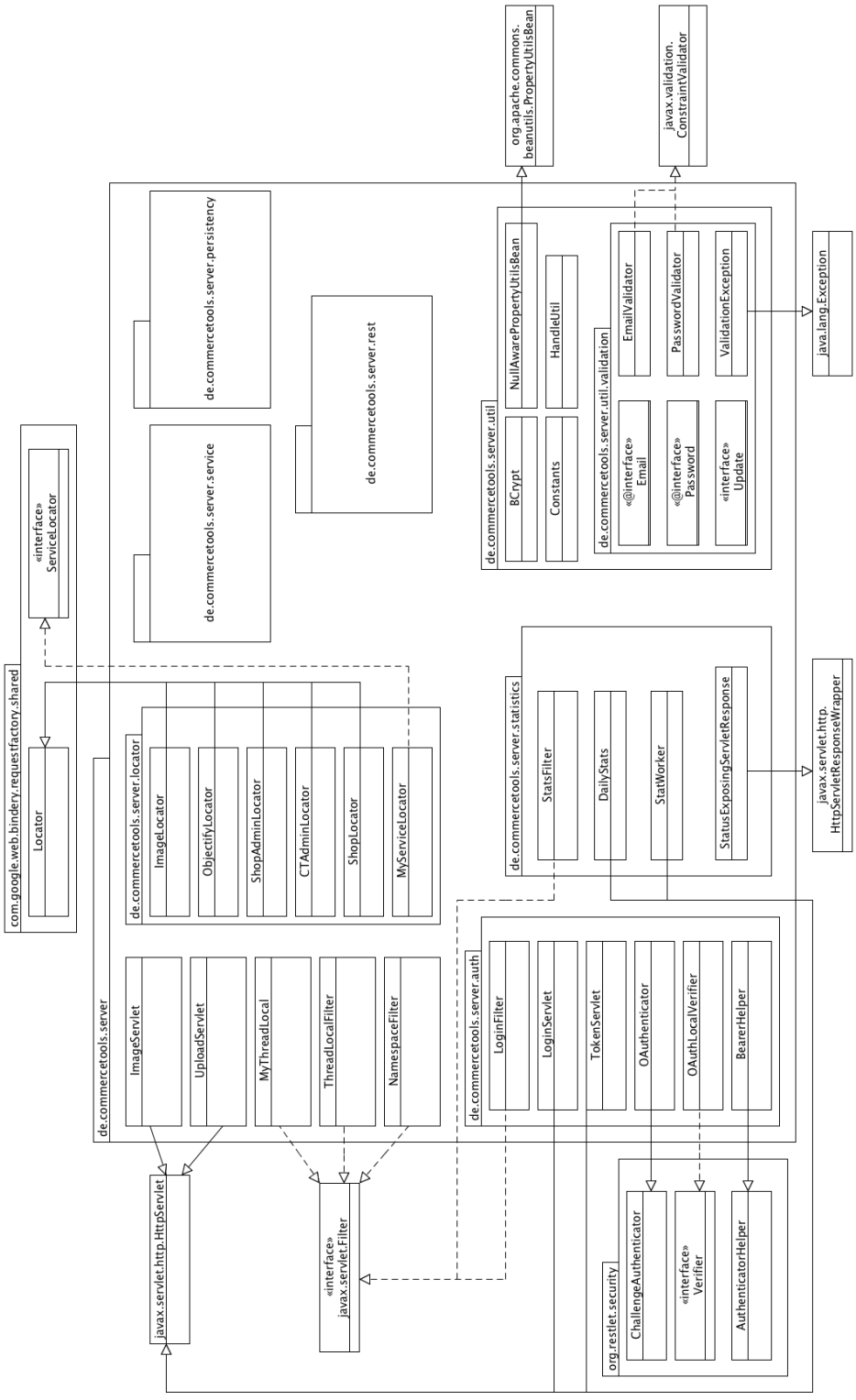


Abbildung 3.5: Die Klassen im *Server*-Package

und Persistenzschicht und implementiert zusätzliche Logik, zum Beispiel zur Validierung von Objekten. Sie enthält die Services-Komponente.

Die Persistenzschicht Die Persistenzschicht (Persistence-Layer) ist für die Speicherung und Beschaffung von Daten verantwortlich. Sie speichert zum Beispiel die Produktdaten im Datastore. ct.Box implementiert diese Schicht in der Persistenz-Komponente.

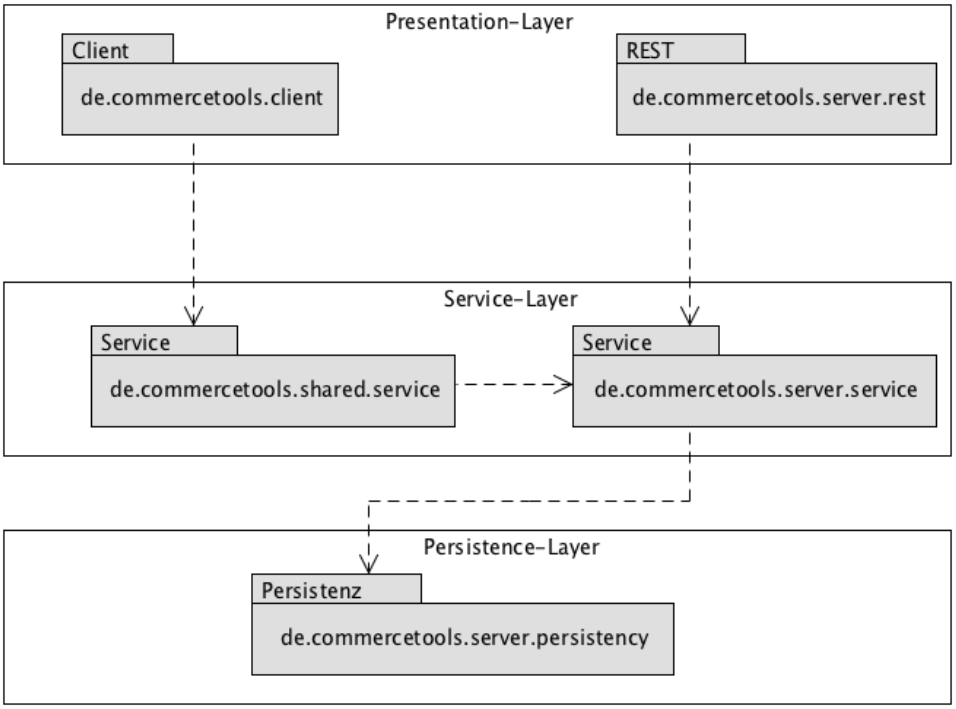


Abbildung 3.6: Komponenten von ct.Box

3.4 Deployment

Um ct.Box auf die Google App-Engine zu übertragen, kann die Deployment-Funktion des GAE Eclipse-Plugins verwendet werden. In Abbildung 3.7 auf Seite 30 ist in Anlehnung an [Dau10] zu sehen, wie die Teile von ct.Box auf die verschiedenen Systeme verteilt werden.

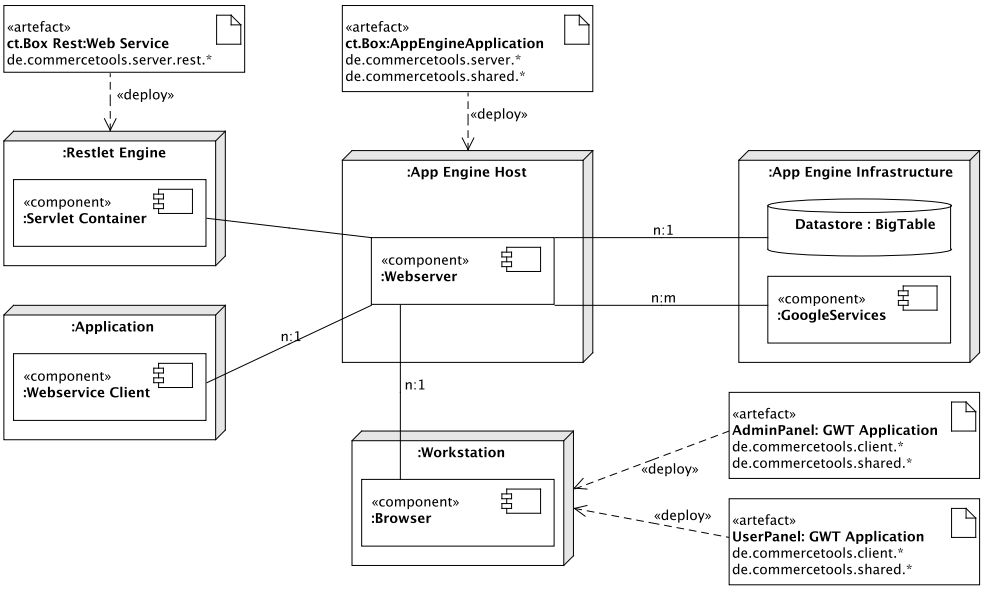


Abbildung 3.7: Deployment Diagramm

3.5 Datenmodell

Abbildung 3.8 zeigt das Datenmodell von ct.Box. Das Vorbild dafür ist aus der API von Shopify[1], einer Plattform zum schnellen Erstellen von Shops, entnommen. Diese API verfügt ebenfalls über ein REST-Interface, für das die dabei verwendeten Klassen dokumentiert werden.[Sho11] Um das Diagramm übersichtlicher zu halten, sind dabei die Felder und Methoden der Klassen nicht eingezeichnet. Da die Logik der Applikation in der Serviceschicht realisiert wird, enthalten die Objekte in der Persistenzschicht ohnehin nur Getter und Setter für ihre Membervariablen (mit Ausnahme von Lifecycle-Callbacks, siehe Abschnitt 4.1.1), sind also reine POJOs (Plain-Old-Java-Objects). Allerdings konnten auf Grund der Einschränkungen des Datastores nicht alle Beziehungen zwischen Klassen wie im Modell dargestellt umgesetzt werden. So können Bilder zum Beispiel nicht als Teil eines Products gespeichert werden. Darauf wird in Abschnitt 4.1.2 später in diesem Kapitel genauer eingegangen.

[1] `http://www.shopify.com/`

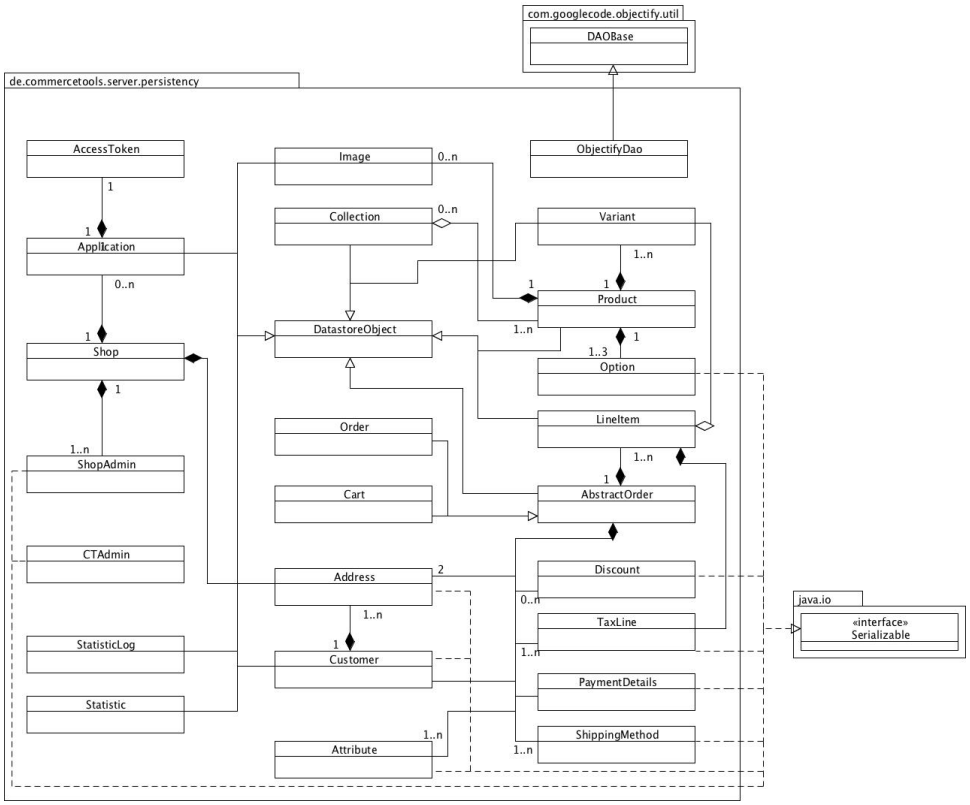


Abbildung 3.8: Das Datenmodell

Ein wichtiges Objekt im Modell ist dabei der *Shop*. Ein Shop stellt einen Kunden von commercetools dar, also einen Nutzer von ct.Box. Das Shop-Objekt enthält Informationen über den Kunden, wie die Adresse und eine Liste von Administratoren (*ShopAdmins*), die berechtigt sind, die Daten dieses Shops zu verwalten. Deswegen wird direkt beim Anlegen eines neuen Shops auch ein ShopAdmin mitregistriert, und das Löschen von ShopAdmins ist nur möglich, falls danach noch ein weiterer Administrator für diesen Shop vorhanden ist. Zu einem Shop gehören weiterhin *Application*-Objekte. Applications sind vom Kunden (oder von diesem beauftragten Dritten) entwickelte Applikationen, die Zugriff auf die Rest-API von ct.Box haben möchten. Dafür erhält eine neu erstellte Applikation ein *AccessToken*-Objekt, das eindeutig einer Applikation (und damit einem Shop) zugeordnet werden kann und zur Autorisierung am Webservice verwendet wird.
Daneben sind natürlich vor allem das *Product*-Objekt und die dazugehörigen Klassen von Bedeutung. Ein Produkt kann in mehreren Variationen (*Varian-*

ten) auftreten. Die Varianten eines Produkts unterscheiden sich hierbei durch die Werte der im Produkt spezifizierten Optionen. Ein Beispiel: In einem Shop für Schuhe könnte das Produkt “Schuh” als Optionen möglicherweise “Farbe” und “Größe” haben. Die Varianten dieses Produkts besitzen dann unterschiedliche Werte für diese Optionen. Es gibt also zum Beispiel eine Variante mit Farbe “blau” und Größe “46”, und eine andere Variante ebenso mit Farbe “blau” aber mit Größe “45” Ein Produkt muss daher mindestens eine Option besitzen, um die Varianten voneinander unterscheiden zu können. Maximal sind 3 Optionen pro Produkt erlaubt. Ein Kunde kauft also eigentlich keine Produkte, sondern eigentlich Varianten. Deswegen enthält ein *LineItem*-Objekt, das eine “Zeile” im Warenkorb darstellt, auch eine Variante, und kein Produkt. Aus einer Variante lässt sich aber sofort auf das zugehörige Produkt schließen, da eine Variante genau einem Produkt zugeordnet ist.

3.6 Sequenzdiagramm

In Abbildung 3.9 ist der Ablauf einer typischen Anfrage an den Webservice von ct.Box dargestellt. Andere Aufrufe unterscheiden sich davon im Grunde nur durch die aufgerufenen Service-Methoden. Dort ist auch gezeigt, wie die Statistiken, die im AdminPanel ausgelesen werden können sollen, generiert werden. Über einen ServletFilter, werden Daten wie die Dauer des Requests, der HTTP-Status der Antwort und das Ziel des Aufrufs erfasst und in Statistik-Objekten persistent gespeichert.

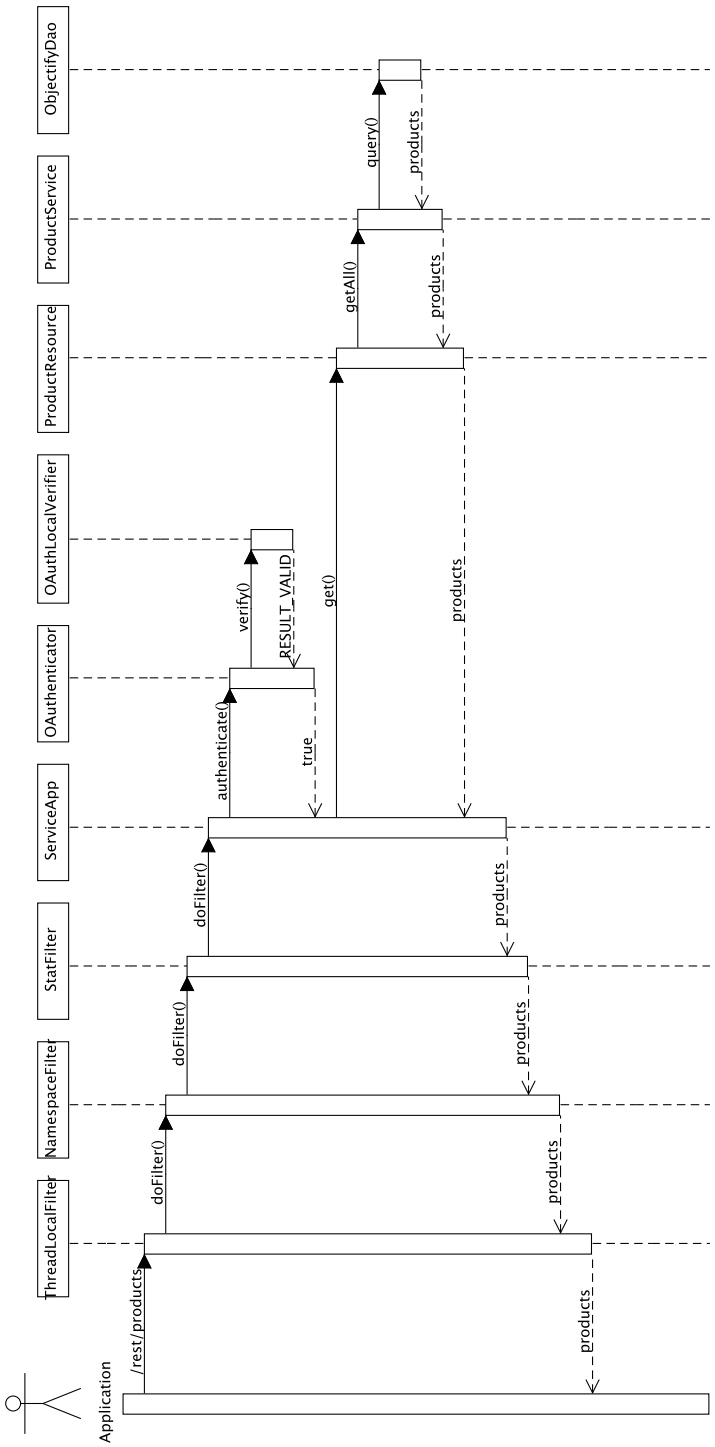


Abbildung 3.9: Abfrage der Produkte über den Webservice

4 Implementierung von ct.Box

Nach dem Entwurf wird nun die Implementierung von ct.Box beschrieben. Dabei wird zuerst auf die Grundlage von ct.Box, die Persistenzschicht, eingegangen.

4.1 Persistenzschicht

In diesem Kapitel wird auf die Persistenzschicht von ct.Box eingegangen, nachdem in der Einleitung schon auf die speziellen Anforderungen des GAE-Datastore eingegangen wurde.

4.1.1 Frameworks

Der App Engine Datastore bietet mehrere Optionen für den Zugriff auf seine Funktionen. Direkt über dem Datastore setzt die Low-Level-API auf, die einfache Operationen wie *get*, *put*, *delete* und *query* anbietet. Aufsetzend darauf sind in dem GAE SDK bereits Implementierungen der Java Data Objects (JDO)[1] und der Java Persistence API (JPA)[2] vorhanden.[Goo11e] Nachdem JDO und JPA eigentlich Standard-Interfaces für die Nutzung relationaler Datenbanken sind, benutzen ihre Implementierung auf der App Engine nicht die Low-Level-API (die speziell auf BigTable abgestimmt ist), sondern die DataNucleus Access Platform[3]. Dadurch sinkt selbstverständlich die Performance von Applikationen, die diese Standard-Interfaces nutzen, da eine zusätzliche Schicht zwischen Anwendung und Datastore liegt. Als weitere Konsequenz unterliegen auch JDO und JPA auf der App Engine einigen Einschränkungen. So können zum Beispiel part-of Beziehungen mit JDO zwar dargestellt werden, um einfache Beziehungen zwischen Entities auszudrücken müssen aber manuell "foreign keys" angelegt und von der Applikation verwaltet werden. Polymorphe Queries, also Anfragen auf einen Entitytyp, die auch Subtypen in das Ergebnis miteinbeziehen, sind ebenfalls nicht möglich, da jede Klasse als eigener Typ im Datastore repräsentiert wird. Außerdem ist es nicht möglich, Aggregatfunktionen (group by, sum, avg, min, max, having) in Queries anzuwenden.[Goo11p]

[1] `http://www.oracle.com/technetwork/java/index-jsp-135919.html`
[2] `http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html`
[3] `http://www.datanucleus.org/products/accessplatform/`

Die Einschränkungen für JPA sind dabei genau die selben.
Es ist logisch, dass auch Datastore-spezifische Einstellungen, wie zum Beispiel "Ancestor-Queries", also Queries mit einem Parent-Parameter, durch diese Standard-Interfaces nicht abgedeckt sind. JDO und JPA tragen zwar zur Portabilität einer Applikation bei, allerdings ist ihre Benutzung auf dem Datastore unnötig kompliziert. Daneben ist es natürlich auch möglich, die Low-Level-API zu verwenden, diese ist aber eher für erfahrene Anwender gedacht. Das erkennt man an der relativ komplizierten Transaction-API, oder den kleinen Hilfen für Entwickler, wie zum Beispiel Typ-sichere Queries, die ausgelassen wurden.[TP11] Es werden dazu keine POJOs, sondern GAE-spezifische Entity-Objekte gespeichert, wodurch die Portabilität des Codes deutlich verringert wird. Deswegen sind noch einige andere Frameworks entstanden, die ebenfalls die Low-Level-API nutzen und den Umgang mit dem Datastore vereinfachen sollen. Die folgenden drei werden Entwicklern vom Google App Engine Team selbst nahegelegt: [Goo11e]

TWiG [4] ist ein leichtgewichtiges Framework, das auf der Low-Level-API des Datastore aufsetzt. Dadurch erzielt es gegenüber JDO und JPA einen Geschwindigkeitsvorteil. TWiG benutzt hierbei die asynchronen Services der Low-Level-API, was es ermöglicht, Operationen auf dem Datastore parallel auszuführen. Dadurch sind auch sogenannte "OR"-Queries möglich, also parallel ablaufende Queries, deren Ergebnisse zusammengeführt werden, wobei Duplikate entfernt werden und sogar die Sortierung erhalten bleibt. Um die Abhängigkeit zum Datastore zu verringern, verzichtet TWiG außerdem auf den Einsatz von Datastore-Keys. TWiG-Entities können einfache POJOs, oder, falls eine genauere Konfiguration notwendig ist, mit Annotationen versehene Klassen sein. Der Einsatz von einfachen POJOs erleichtert dabei sowohl dem Umgang mit dem Frontend, als auch die Portierung einer Applikation auf ein anderes Datenbanksystem. Schwieriger ist der Umgang mit dem Memcache. Dazu müssen interne Klassen des Frameworks erweitert werden. [TP10] Intgeriert ist dafür ein interne Key-Instanz Cache, in dem Entity-Instanzen lokal zwischengespeichert werden und so schnell zurückgegeben werden können, ohne ein Anfrage an den Datastore zu richten. TWiG ist zudem recht gut Dokumentiert. Das Wiki[5] bietet eine gute Übersicht über den Funktionsumfang und erleichtert den Einstieg in TWiG ungemein. Dazu gibt es eine Diskussions-Gruppe[6], in der weiterführende Problem mit anderen Entwicklern diskutiert werden können. Nur über Umwege ist

[4] http://code.google.com/p/twig-persist/
[5] http://code.google.com/p/twig-persist/wiki/Contents
[6] http://groups.google.com/group/twig-persist

allerdings die Javadoc der aktuellen Version (2.0 beta) zu finden.

Slim3 [7] ist im Kern ein Model-View-Controller(MVC)-Framework für die GAE mit Java. Allerdings kann es auch nur als Datastore-Interface verwendet werden. Slim3 adressiert die selben Probleme wie TWiG, nämlich die Komplexität und die Performance von JDO/JPA.[Sli] Es ist ebenfalls ein Wrapper für die Low-Level-API und verwendet Annotationen, die beim Kompilieren Metadaten erzeugen, um die Abbildung von Modell auf Entity zu bewerkstelligen. Weiterhin bietet Slim3 die Möglichkeit polymorphe Queries auf dem Datastore auszuführen. Die Dokumentation von Slim3 ist allerdings nicht so ausführlich wie bei TWiG, und es ist auch klar zu erkennen, dass der Fokus hier nicht nur auf dem Datastore-Wrapper liegt, nachdem der Funktionsumfang bei TWiG doch größer ist. Es sind ebenfalls eine Diskussionsgruppe[8], sowie eine recht umfangreiche Dokumentation und eine Javadoc für Slim3 vorhanden. [Sli]

Objectify [9] stellt ebenfalls einen Wrapper für die Low-Level-API dar. Dabei versucht es das einfache Interface (*get*, *put*, *delete*, *query*) der Low-Level-API beizubehalten und trotzdem dem Entwickler so viele Hilfen wie möglich mit an die Hand zu geben. Objectify unterstützt nützliche Features wie Batch-Operationen, Asynchrone Befehle und polymorphe Queries. Dabei können mit Objectify, genau wie bei TWiG, POJOs im Datastore abgelegt werden.[Objc] Für genauere Einstellungen werden ebenfalls Annotationen (JPA/javax.persistence) verwendet. Besonders nützlich ist die Funktion, Objekte automatisch im globalen Memcache der GAE abzulegen. Dafür ist nur eine Annotation auf dem Entity nötig. Im Memcache werden aber keine ganzen Objekte, sondern nur die Felder eines Entitys gespeichert. Da Write-Operationen den Memcache nicht berücksichtigen, ist sein Einsatz hauptsächlich bei Applikationen mit viele Reads von Nutzen. Ergänzend dazu bietet Objectify noch einen lokalen Session-Cache, der komplette Objekte aufnehmen kann. Wird ein Entity per *get()* oder *query()* abgefragt und liegt schon im Session-Cache, so wird genau dieselbe Instanz des Entitys zurückgegeben. Dadurch lässt sich die Anzahl der Anfragen an die App-Engine Server teilweise enorm verringern. Allerdings ist der Session-Cache nicht global, wird also nicht von allen Applikations-Instanzen geteilt. Da er außerdem nicht Thread-sicher ist, darf er nicht von mehreren Threads gleichzeitig verwendet werden. Ein weiteres Feature von Objectify sind die sogenannten Lifecycle-Callbacks. Objec-

[7] `https://sites.google.com/site/slim3appengine/`
[8] `http://groups.google.com/group/slim3-user`
[9] `http://code.google.com/p/objectify-appengine/`

tify erlaubt es Entwicklern Methoden zu definieren, die entweder direkt bevor ein Entity gespeichert wird, oder direkt nachdem es aus dem Datastore geladen wurde, aufgerufen werden. Das ermöglicht zum Beispiel Migrationen im Datenmodell, selbst wenn die Applikation schon auf der GAE läuft.[Objb] Eine weitere Stärke von Objectify ist die Dokumentation. Neben einer Einführung in die Konzepte des Datastore existieren sowohl eine Javadoc, als auch eine gut verständliche Anleitung für die Funktionen von Objectify.[Objb] Für tiefergehende Fragen bietet sich die Möglichkeit, sich an die Objectify-Diskussionsgruppe zu wenden.

Wie man sieht, sind sich Objectify und TWiG im Umfang ihrer Funktionalitäten sehr ähnlich, während Slim3 hier ein wenig abfällt. Besonders Features wie das automatische Cachen von Entities, oder die Lifecycle-Callbacks, machen Objectify für die Implementierung von ct.Box allerdings attraktiver. Die Tatsache, dass TWiG bei der Abstraktion des Datastore einen Schritt weitergeht und die Verwendung von Key-Feldern in Entities nicht mehr notwendig sind, ist zwar für Applikationen nützlich, die erwarten, einmal von der App-Engine portiert werden zu müssen, für ct.Box ist es allerdings kein Problem das Abstraktionslevel niedriger zu halten. Dadurch sind zum Einen Datastore-spezifische Lösungen einfacher zu implementieren, und zum Anderen ist ct.Box eine speziell für die GAE entwickelte Applikation, weswegen Portabilität nur eine geringe Rolle bei den Designentscheidungen spielt.

4.1.2 Implementierung

Erste Schritte

Wenn Datenmodell und das zu verwendende Framework festgelegt sind, verläuft die Implementierung erst einmal ziemlich geradlinig. Zuerst muss natürlich Objectify installiert werden. Dazu muss nur die auf der Seite von Objectify angebotene Bibliothek in den /war/lib Ordner des Projekts kopiert, und zum Classpath das Projekts hinzugefügt werden. Danach werden für die im Datenmodell definierten Klassen (siehe Abblidung 3.8) Objekte (POJOs) erstellt, jeweils bereits mit den benötigten Feldern und den dazugehörigen Getter- und Setter-Methoden. Damit die Klassen mit Objectify persistiert werden können, müssen sie weiterhin einen Konstruktor ohne Parameter, oder keinen Konstruktor (da Java in diesem Fall den leeren Konstruktor erzeugt) besitzen. Dabei darf dieser Konstruktor jede Sichtbarkeit besitzen (private, protected, public). Jedes dieser Objekte, das als eigene Entität im Datastore gespeichert werden soll, braucht weiterhin einen Schlüssel (siehe 2.4.3). Dieser Schlüssel kann entweder vom Datastore generiert, wozu er zwingend vom Typ Long sein muss, oder von der Applikation vorgegeben werden. In diesem Fall kommt

jedes Feld eines Objekts, das vom Typ String oder Long ist, als Schlüssel-Feld in Frage. Um dieses Feld festzulegen, existiert in Objectify die Annotation *@Id*. Der Wert dieses Feldes wird dann beim Speichern des Objekts in den Schlüssel eingearbeitet.

Listing 4.1 zeigt die Felder eines Entitys am Beispiel der DatastoreObject-Klasse, die als Superklasse gemeinsame Felder für alle persistenten Klassen mit einer Long-Id kapselt.

```
public class DataStoreObject {
  @Id
  private Long id;
  @Unindexed
  private Integer version = 0;

  private Date created_at;
  private Date updated_at;
}
```

Listing 4.1: Festlegen des Schlüssel-Felds

Ein neues Schlüssel-Objekt wird dann im Allgemeinen wie folgt angelegt:

```
Key<Product> key = new Key<Product>(Product.class, someProductId);
```

In Listing 4.1 erkennt man schon ein zusätzliches Feature von Objectify. Über die *@Unindexed*-Annotation lassen sich Felder (oder ganze Klassen), die nicht in Queries benötigt werden, von der automatischen Index-Generierung des Datastore ausschließen. Dadurch lässt sich die Performance beim Schreiben von Entities steigern. Und auch wenn später einmal doch noch ein Index auf einem solchen Feld gebraucht wird, so kann die Annotation einfach entfernt werden um beim nächsten Speichern des Objekts kreiert der Datastore die Indizes wie erwartet. Das Gegenteil erreicht man mit *@Indexed*. Diese Annotation ist vor allem dann nützlich, wenn eine gesamte Klasse als *@Unindexed* deklariert wurde, und nur einzelne Felder indziert werden sollen. Außerdem lassen sich sogar noch Bedingungen festlegen, unter denen ein einzelnes Feld indiziert werden soll. [Objb] In der Standardeinstellung von Objectify sind jedoch alle Klassen und alle Felder in diesen Klassen als *@Indexed* markiert.

Relationships

Der nächste Schritt bei der Implementierung ist die Umsetzung der Beziehungen zwischen den Entities. Um mit Objectify eine Beziehung zwischen Entities auszudrücken, reicht es, den Schlüssel des (bzw. der) jeweils anderen En-

tity's selbst als Feld zu halten.[Objb] Anhand von Beispielen sind hier die gebräuchlichsten Arten von Beziehungen erklärt:

Parent-Child Relationship Die Eltern-Kind-Beziehung zwischen zwei Objekten hat bei der GAE eine besondere Bedeutung. Wie bereits erwähnt werden Entities innerhalb einer Parent-Child Hierarchie in einer Entity-Group zusammengefasst. Nur auf diesen Entity-Groups sind klassenübergreifende Transaktionen möglich. Damit zum Beispiel Produkte und Varianten innerhalb einer Transaktion verwendet werden können, enthält jede Variante also ein Feld mit dem Schlüssel ihres Eltern-Produkts. Zusätzlich muss dieses Feld mit der Annotation *@Parent* versehen werden, damit der Schlüssel der Variante richtig gebildet werden kann. In einer Variante sieht diese Beziehung also folgendermaßen aus:

```
public class Variant extends DataStoreObject {

  @Parent
  private Key<Product> parent_key;

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

Listing 4.2: Festlegen des Schlüssel-Felds

Damit ist die Parent-Child Beziehung hergestellt. Wichtig zu beachten ist, dass nun bei der Erstellung eines Schlüssels für eine Variante jedes Mal auch der Schlüssel des Produkts, zu dem diese Variante gehört, benötigt wird. Listing 4.3 zeigt das Anlegen eines Key-Objekts für eine Variante.

```
Key<Variant> key = new Key<Variant>((new Key<Product>(Product
    .class, someProductId), Variant.class, someVariantId);
```

Listing 4.3: Erstellen eines Schlüssels für eine Parent-Child-Beziehung

Vergisst man hier, den Produktschlüssel miteinzubeziehen, so ergibt sich erst zur Laufzeit ein Fehler, denn im Datastore kann logischerweise keine Variante mit dem gegebenen Schlüssel gefunden werden, da alle gültigen Variantenschlüssel ein Produkt als Vorfahren in ihrem Schlüssel enthalten.

One-to-One Um eine 1:1-Beziehung auszudrücken, genügt es analog zur Parent-Child-Beziehung, in einem Feld den Schlüssel seines Beziehungspartners

zu speichern. Soll ein solche Beziehung bidirektional gehalten werden, enthält dementsprechend auch der Partner einen Referenzschlüssel als Feld.

One-to-Many Listing 4.4 zeigt eine 1:N-Beziehung anhand von Produkten und Varianten. Wie wir bereits gesehen haben, enthält eine Variante ein Feld mit dem zu ihre gehörigen Produktschlüssel. An sich ist das auch schon für eine One-to-Many-Relationship ausreichend. Mit einer Query lassen sich alle Varianten finden, die zu einem bestimmten Produkt gehören.

```
public class Product extends DataStoreObject {

  //...

  @Transient
  private List<Variant> variants;

  @Transient
  private List<Image> images;
}

public class Variant extends DataStoreObject {

  @Parent
  private Key<Product> parent_key;

  //...
}
```

Listing 4.4: Eine 1:N-Beziehung

Wie man sieht enthält ein Produkt eine Liste von Varianten, die allerdings als *@Transient* gekennzeichnet sind. Gleiches gilt für die Bilder eines Produkts. Hier werden nicht nur Schlüssel als Felder gehalten, da diese Objekte über den Webservice komplett zurückgegeben werden sollen. Darauf wird im kommenden Abschnitt 4.3.1 detailliert eingegangen.
Als *@Transient* markierte Felder werden nicht persistent im Datastore abgelegt, sondern einfach ignoriert.[Objb] Nachdem Varianten als eigene Entities gespeichert sind ist es erstens nicht notwendig sie noch einmal beim Produkt mitzuspeichern, und zweitens auch gar nicht möglich, bedenkt man das 1MB Limit für die Größe eines Entity's. Diese Liste wird jedes Mal, wenn ein Produkt aus der Datenbank geladen wird, neu befüllt. Dazu haben sich die von Objectify bereitgestellten Lifecycle-Callbacks als sehr nützlich erweisen. Auf die Lifecycle-Callbacks wird aber zu einem

späteren Zeitpunkt noch ausführlicher eingegangen.
Auch hier gilt: Soll eine M:N-Beziehung modelliert werden, so müssen die gerade dargestellten Schritte einfach für beide Partner der Beziehung ausgeführt werden.
Eine andere Möglichkeit, eine 1:N-Beziehung zu modellieren, wäre in diesem Fall, dem Produkt zusätzlich ein Feld mit einer Liste der Schlüssel seiner Varianten hinzuzufügen. Dann müsste aber sowohl beim Erstellen, als auch beim Löschen einer Variante immer das dazugehörige Produkt ebenfalls aktualisiert werden. Das würde einen viel höheren Aufwand bedeuten. Anstatt von nur einem Aufruf an den Datastore, um eine Variante zu löschen, müssten bei dieser Alternative drei Requests gesendet werden. Einer, um die Variante zu entfernen, einer zum Laden des korrespondierenden Produkts und schließlich noch einer zum Speichern des geänderten Produkts.

Werden die Beziehungen wie gezeigt umgesetzt, entsprechen sie nur Aggregationen.
Um Kompositionen darzustellen, ist mehr Logik in der Applikation notwendig. Der Datastore kümmert sich also zum Beispiel nicht um das Löschen von Varianten, falls ein Produkt entfernt wird. Das bleibt den Applikationen überlassen. Außerdem gibt es im Datastore auch kein referentielle Integrität. Das heißt in ct.Box muss selbst sichergestellt werden, dass die für die Beziehungen gespeicherten Schlüssel auf gültige Entities verweisen und eventuelle Fehler entsprechend behandelt werden.
Es gibt aber auch noch andere Möglichkeiten, Kompositionen umzusetzen. So ermöglicht es der Datastore, Objekte in serialisierter Form als Felder eines anderen Entities zu persistieren. Dafür muss die zu serialisierende Klasse das Interface *Serializable* implementieren, und das Feld, das diese Klasse aufnimmt, mit *@Serialized* gekennzeichnet sein.
Listing 4.5 zeigt eine solche Konstellation am Beispiel der Optionen eines Produkts.

In diesem Fall ist wieder eine 1:N-Beziehung umgesetzt. 1:1-Beziehungen sind natürlich genauso möglich, indem das Feld keine Liste, sondern ein einzelnes Objekt erhält.
Da die Objekte vor dem Speichern serialisiert werden, können natürlich auf deren Feldern keine Queries ausgeführt werden. Folgende Query könnte also nicht ausgeführt werden: `List<Product> productsWithOptionName = dao.ofy().query(Product.class).filter("options.name", someName).list();`
Die Annotation *@Embedded* ermöglicht aber auch solche Queries, während weiterhin die Objekte verschachtelt ineinander gespeichert werden. Dazu werden die Felder des als *@Embedded* gekennzeichneten Objekts als Felder im überge-

```
public class Option implements Serializable{
  private static final long serialVersionUID = 2174007279794311439
      L;
  private String name;
  private Set<String> values;
  //...
}
public class Product extends DataStoreObject {
  //...
  @Serialized
  private List<Option> options;
}
```

Listing 4.5: Ein serialisiertes Feld

ordneten Objekt gespeichert, an ihren Name aber noch der Name der *@Embedded*-Klasse angehängt.[Objb]
Ein Beispiel aus der Objectify-Dokumentation ist in Listing 4.6 dargestellt. [Objb]

```
class LevelTwo {
    String bar;
}

class LevelOne {
    String foo;
    @Embedded LevelTwo two
}

class EntityWithEmbedded {
    @Id Long id;
    @Embedded LevelOne one;
}

class EntityWithEmbeddedCollection {
    @Id Long id;
    @Embedded List<LevelOne> ones = new ArrayList<LevelOne>();
}
```

Listing 4.6: Ein eingebettetes Feld

Eine Query auf diesen Entities könnte dann folgendermaßen aussehen:

Dabei wird natürlich vorausgesetzt, dass die gefilterten Felder indiziert sind. Bei eingebetteten Klassen funktioniert die Indizierung genauso wie bei “normalen” Klassen. Dabei werden die Indizierungseinstellungen an die eingebet-

```
Objectify ofy = ObjectifyService.begin();
ofy.query(EntityWithEmbeddedCollection.class).filter("ones.two.bar
    =", "findthis");
```

Listing 4.7: Ein eingebettetes Feld

teten Klassen vererbt. Einzig *@Indexed* oder *@Unindexed* auf Feldern innerhalb einer *@Embedded*-Klasse überschreiben die vererbten Default-Werte.

Datenzugriff mit Objectify

Damit ist das Datenmodell bereit in den Datastore geschrieben zu werden. Bevor mit Objectify aber Operationen auf dem Datastore ausgeführt werden können, müssen alle Entity-Klassen bei der zentralen Klasse von Objectify, dem *ObjectifyService*, registriert werden.[Objb] Im Gegensatz zu anderen Frameworks (z.B. JDO,JPA) durchsucht Objectify nämlich nicht den Java-Classpath nach Klassen, die mit @Entity oder ähnlichen Annotationen versehen sind. Dafür gibt es mehrere Gründe. Auch wenn das automatische Scannen äußerst komfortabel ist, gibt es doch einige Nachteile, die dieses mit sich bringen würde. Zum einen wäre Objectify von weiteren Bibliotheken abhängig, die auch in jedes Projekt, das Objectify verwendet, mit eingebunden werden müssten. Andererseits müssten Entwickler Logik implementieren, die beim Starten des Servlet-Containers den Scan anstößt. Der schwerwiegendste Nachteil besteht allerdings darin, dass solche Scans extrem langsam sind, da der Bytecode jeder einzelne Klasse und der eingebunden Bibliotheken durchsucht werden muss. Dadurch verzögert sich die Start-up-Zeit einer Applikation auf der App-Engine um circa drei bis fünf Sekunden. [Obja]
Nachdem die App-Engine nach Bedarf Applikationsinstanzen verwaltet, müssen dadurch Benutzer der Applikation länger warten, bevor sie diese nutzen können, selbst wenn eine Applikation beständig Traffic verzeichnen kann.
Die Registrierung der Entities muss erfolgen, bevor auf den Datastore zugegriffen wird. Dazu bieten sich zum Beispiel ein ServletContextListener [10], oder ein einfaches Servlet, das beim Applikationsstart aufgerufen wird, an.[Objb] Objectify zeigt in seiner Dokumentation aber noch eine andere Lösung auf. Wird auf Objectify über ein Data-Access-Object (DAO) [11] zugegriffen, so können in diesem Objekt in einem statischen Block alle Registrierungen vorgenommen

[10] http://download.oracle.com/javaee/6/api/index.html?javax/servlet/ServletContextListener.html
[11] http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

werden. Dadurch werden auch die Operationen auf dem Datastore weitestgehend abstrahiert. Objectify bietet dafür mit der *DAOBase*-Klasse schon eine Basisklasse an, die erweitert werden kann.[Obja] Allerdings müsste bei diesem Ansatz für jede Entity-Klasse eine eigene DAO-Klasse erstellt werden, was eine große Menge von duplizierten Code bedeuten würde, da die Basisoperationen auf dem Datastore für alle Objekte gleich aussehen. Um dieses Problem zu umgehen, wird bei ct.Box eine generische DAO-Klasse [Cha10] verwendet, die von *DAOBase* erbt. In dieser Klasse sind neben einigen nützlichen Hilfsfunktionen nur die elementaren Interaktionen mit dem Datastore implementiert. Erweiterte Logik, die zum Beispiel beim Löschen von bestimmten Entities benötigt wird, wird in der Serviceschicht realisiert. Listing 4.8 zeigt einen Ausschnitt der ObjectifyDao-Klasse. Wie man sieht werden hier in alle Entities beim ObjectifyService registriert. Die DAOBase-Klasse bietet außerdem die Methode *ofy()* an, die ein *Objectify*-Objekt zurückliefert. Das Objectify-Objekt ist die "Schaltzentrale" in Objectify. Über dieses Objekt werden Entities im Datastore gespeichert, daraus geladen und gelöscht. Auch Queries können über das Objectify-Objekt abgesetzt werden. Für jede dieser Operationen ist im Listing ein Beispiel enthalten. Der Datastore und Objectify ünterstützen weiterhin Batch-Operationen. Dafür werden mehrere get()-, delete() oder put()-Operationen in einer Anfrage an den Datastore geschickt und dort parallel verarbeitet. Da natürlich jede Anfrage an den Datastore Zeit benötigt, ist diese Methode mehrere Operationen auszuführen um ein vielfaches performanter, als zum Beispiel in einer Schleife alle Objekte einer Liste einzeln abzuarbeiten, was eine eigene Anfrage für jedes Objekt mit sich bringen würde.

```
public class ObjectifyDao<T> extends DAOBase {

  static {
    ObjectifyService.register(Collection.class);
    ObjectifyService.register(Order.class);
    ObjectifyService.register(Product.class);
    ObjectifyService.register(Variant.class);
    ObjectifyService.register(Application.class);

    //...
  }

  protected Class<T> clazz;

  public ObjectifyDao(Class<T> cl) {
    clazz = cl;
  }

  /**
   * Save an object to the datastore
```

```
     *
     * @param entity
     *             - Object to save
     * @return the key of the saved object
     */
    public Key<T> put(T entity)

    {
      return ofy().put(entity);
    }

    /**
     * Batch putting of objects
     *
     * @param entities
     *             - Objects to save to the Datastore
     * @return a Map of all Keys of the saved objects
     */
    public Map<Key<T>, T> putAll(Iterable<T> entities) {
      return ofy().put(entities);
    }

    /**
     * Delete an Entity from the Datastore
     *
     * @param entity
     *             - The Entity to delete
     */
    public void delete(T entity) {
      ofy().delete(entity);
    }

    /**
     * Get an Entity with a Long id
     *
     * @param id
     *             - The Long id of the Entity
     * @return the Entity
     * @throws NotFoundException
     *              if no Entity with the given Id exists
     */
    public T get(Long id) throws NotFoundException {
      return ofy().get(this.clazz, id);
    }

    /**
     * Get an entity by its Datastore-Key
     *
     * @param key
     *             - The Key of the Entity
```

```
     * @return the Entity
     * @throws NotFoundException
     *             if no Entity with the given Id exists
     */
    public T get(Key<T> key) throws NotFoundException {
      return ofy().get(key);
    }

     /**
     * Batch get of Entities
     *
     * @param keys
     *            - The keys of the entities to get
     * @return a Map of the fetched entities
     */
    public Map<Key<T>, T> get(Iterable<Key<T>> keys) {
      return ofy().get(keys);
    }

     /**
     * Get a list of all Entities of this type
     *
     * @return a List of all existing entities
     */
    public List<T> listAll() {
      Query<T> q = ofy().query(clazz);
      return q.list();
    }

    //...
  }
```

Listing 4.8: Generisches DAO-Objekt für Objectify

Für Anfragen an den Datastore wird also immer die Klasse der Entities benötigt. Stellt man sich den Datastore wieder als relationale Datenbank vor, in dem die Klasse eines Entity's einer Tabelle entspricht in der in den Reihen die Entity-Objekte abgelegt werden, macht das durchaus Sinn. Schließlich muss ja die Tabelle bekannt sein, aus der Informationen ausgelesen werden sollen.
Jetzt können auch die vorher erwähnten Lifecycle-Callbacks eingehender betrachtet werden. Wie wir gesehen haben, enthält ein Produkt für die 1:N-Beziehung zu seinen Varianten eine *@Transient*-Liste von Varianten, die jedesmal, wenn das Produkt aus dem Datastore geladen wird, entsprechend gefüllt wird. Dazu werden die Lifecycle-Callbacks verwendet. Ein Lifecycle-Callback ist dabei eine Methode, die direkt bevor ein Entity in den Datastore gespeichert wird, oder direkt nachdem es aus dem Datastore geladen wurde, aufgerufen wird. Dafür existieren zwei Annotationen. Mit *@PostLoad* annotierte Metho-

den werden nach dem Laden aufgerufen, mit *@PrePersist* annotierte vor dem Speichern. Es können durchaus auch mehrere Methoden mit diesen Annotationen versehen werden. Diese werden dann nach ihrer Reihenfolge im Quellcode ausgeführt.[Objb] Die Methode müssen hierfür in den Entity-Klassen selbst definiert werden.
Somit eignen sich die Lifecycle-Callbacks auch hervorragend für möglicherweise benötigte Schema-Migrationen. Wird einem Entity-Modell zum Beispiel ein neues Feld hinzugefügt, so kann eine *@PostLoad*-Methode dazu verwendet werden, diesem Feld beim Laden "alter" Entities einen Default-Wert zuzuweisen. Beim Speichern wird das neue Feld dann auch im Datastore hinzugefügt, wodurch die Migration komplett ist.[Obja]
Die Implementierung der *@PostLoad*-Methode für die Produkte ist in Listing 4.9 dargestellt.

```
public class Product extends DataStoreObject {

  //...

  @PostLoad
  private void afterLoad() {
    ObjectifyDao<Product> dao = new ObjectifyDao<Product>(Product.
        class);
    initLists();
    setVariants(dao.ofy().query(Variant.class).ancestor(this).
        order("created_at").list());
    setImages(dao.ofy().query(Image.class).filter("product_id =",
        this.getId()).order("position").list());
  }

  //...
}
```

Listing 4.9: Der Lifecycle-Callback in der Product Klasse

Die Methode *initLists()* stellt hier nur sicher, dass die Listen für die Varianten und Bilder auch initialisiert sind, bevor die Objekte darin gespeichert werden. Hier sieht man auch eine sogenannte "Ancestor-Query". Da Produkte und Varianten in einer Parent-Child-Beziehung stehen, können über eine Query mit der Filterfunktion *ancestor()* alle Varianten gefunden werden, deren Schlüssel in ihrem Pfad (siehe 2.4.3) einen Verweis auf das angegebene Produkt enthalten.
Darüber hinaus zeigt dieses Beispiel, wie die Sortierreihenfolge für Queries festgelegt werden kann. Auch hier ist zu beachten, dass die in der *order*-Methode

verwendeten Felder indiziert sein müssen, damit die Abfrage richtige Ergebnisse liefert.

4.1.3 Multi-Tenancy

Multi-Mandantenfähigkeit wird bei der App-Engine über das Benennen eines Namespaces realisiert.(siehe Abschnitt 2.3) In ct.Box wird der aktuelle Namespace in einem *ServletFilter*[12] (NamespaceFilter im Klassendiagramm) bei jedem Request neu gesetzt. Da jeder Shop eine eigene Subdomain besitzt, kann diese für jeden Request als Wert für den aktuellen Namespace verwendet werden. In Listing 4.10 ist die Vorgehensweise genau dargestellt.

Hier wird zusätzlich noch überprüft, ob der Zugriff auf ct.Box für diesen Shop gesperrt wurde, um in diesem Fall eine Fehlermeldung zurückzugeben. Wie man außerdem sehen kann, gibt es einen Default-Namespace, der einfach durch den leeren String repräsentiert wird. In diesem Namespace werden allgemeine Daten gespeichert, die nicht zu einem spezifischen Shop gehören, wie zum Beispiel die Accounts der commercetools-Administratoren.
Wichtig zu wissen ist noch, dass der NamespaceManager eine Thread-lokale Variable ist. [Haz10] Da die Google App-Engine jeden Request als einzelnen Thread handhabt, ist ein Namespace also Thread-lokal gesetzt. Das ist insofern wichtig, da sonst bei kurz aufeinanderfolgenden Anfragen auf unterschiedliche Subdomains der Namespace falsch gesetzt werden würde. Die 2. Anfrage würde den Namespace der vorhergehenden einfach "überschreiben".
Dadurch wird es auch erst möglich, Operationen wie in Listing 2.1 auf Seite 15 durchzuführen, nachdem sich sonst alle Threads einen NamespaceManager teilen würden. Der Nachteil beim Einsatz eines Filters ist, dass für jeden Request auf jeden Fall 2 Aufrufe an den Datastore (bzw. Memcache) gemacht werden müssen, was die Performance der Applikation beeinträchtigt.

4.1.4 Erkenntnisse

Im Ganzen gesehen ist die Implementierung einer Persistenzschicht ohne größere Probleme verlaufen. Hat man sich das Datenmodell zurechtgelegt, und ist sich der Einschränkungen der App-Engine bewusst, ist es kein großer Aufwand, ein erstes lauffähiges Modell zu erstellen. Besonders Objectify vereinfacht die Handhabung mit dem Datastore enorm. Alle Operationen beschränken sich auf einfache *get()*-, *put()*-, *delete()* oder *query()*-Aufrufe, durch die Verwendung eines generischen DAOs lassen sich auch nützliche Hilfsfunktionen zentral für alle Entities definieren, und die Lifecycle-Callbacks erleichtern die Implementierung der Beziehungen ungemein.

[12] `http://www.oracle.com/technetwork/java/filters-137243.html`

```
public class NameSpaceFilter implements Filter {
  @Override
  public void doFilter(ServletRequest request, ServletResponse
      response, FilterChain chain) throws IOException,
      ServletException {
    String serverName = request.getServerName();
    String[] splitted = serverName.split("\\.");
    String subdomain = splitted[0];
    List<String> namespaces = MyThreadLocal.getServiceFactory().
        adminService().getAllNamespaces(0, 1);
    if (namespaces.contains(subdomain)) {
      if (MyThreadLocal.getServiceFactory().shopService().get(
          subdomain).getLocked()) {
        ((HttpServletResponse) response).setStatus(
            HttpServletResponse.SC_FORBIDDEN);
        return;
      }
      NamespaceManager.set(subdomain);
      chain.doFilter(request, response);
    } else {
      if (subdomain.equals(SystemProperty.applicationId.get())) {
        NamespaceManager.set("");
        chain.doFilter(request, response);
      } else {
        ((HttpServletResponse) response).setStatus(
            HttpServletResponse.SC_NOT_FOUND);
        return;
      }
    }
  }
}
```

Listing 4.10: Setzen des aktuellen Namespace in einem ServletFilter

Diese hat sich dabei als kompliziertester Teil der bisherigen Arbeit herausgestellt. Bei einer naiven Implementierung (ohne sich mit den Eigenarten des GAE-Datastore auseinanderzusetzen) würde man Objekte wie die Varianten oder die Bilder einfach als Listen innerhalb des Produkt-Objekts speichern. Der Datastore ermöglicht diese Modellierung allerdings nicht. Um nicht zu weit davon abzuweichen, würden sich Ansätze anbieten, die die *@Embedded* oder *@Serialized* Annotationen verwenden, um die Objekte weiterhin verschachtelt zu speichern. Durch das 1MB-Limit für Entities ist diese Methode aber nur für kleine Objekte geeignet. Außerdem sollen Varianten und Bilder ja selbst Entities sein, die verwaltet werden können. Deshalb ist für ct.Box die Implementierung mit Fremdschlüsseln in den jeweiligen Entities nicht zu umgehen,

auch wenn dies erhöhten Aufwand in der Applikationsschicht bedeutet. Für kleinere Objekte wie die Optionen oder Adressen kann aber trotzdem mit der *@Serialized*-Annotation gearbeitet werden.
Ein Feature, das im GAE-Datastore bis jetzt noch nicht implementiert ist, in ct.Box aber von Nutzen gewesen wäre, ist die Volltextsuche auf den gespeicherten Entities (bzw. LIKE-Queries in SQL). So können Produkte im Moment nur über ihre Id identifiziert werden. Mit dem *handle*-Feld besteht für die Zukunft aber auch die Möglichkeit, Klarnamen zur Identifikation zu nutzen. Die Handles sollen dabei aus dem Produkttitel generiert werden, der allerdings nicht eindeutig ist. Damit kann es passieren, dass zwei Produkte mit dem selben Titel existieren. Diese Produkte müssen dann aber jeweils ein eindeutiges Handle besitzen. Als Lösung dafür würde beim Erstellen eines neuen Produkts eine Suche im Datastore nach Produkten mit einem ähnlichen Handle ausreichen, das ist allerdings nicht auf einfachem Weg möglich. Das Problem kann durch eine spezielle Query, zu sehen in Listing 4.11 umgangen werden.

```
int handleCount = dao.ofy().query(Product.class).filter("handle >=
    ", handle).filter("handle <", handle + "\uFFFD").count();
```

Listing 4.11: Abfrage nach Produkten mit einem ähnlichen Handle

Durch zwei Ungleichheitsfilter wird die Anzahl ähnlicher Produkt-Handles herausgefunden. `\uFFD` stellt hierbei den "größten" Unicode-Character dar, wodurch der durch `handle + "\uffD"` ausgedrückte String "größer" als alle möglicherweise bereits exisitierenden Handles ist. Ist der *handleCount* nun ungleich Null, so wird er dem neuen Handle einfach angehängt, womit dieses wieder eindeutig ist. In zukünftigen Versionen der GAE wird dieses Problem nicht mehr bestehen, da eine Volltextsuche bereits auf der Liste der beabsichtigten Neuerungen erscheint. [Goo11a]

4.2 Serviceschicht

In diesem Kapitel wird auf die Service-Schicht von ct.Box eingegangen. In der Serviceschicht wird die Businesslogik der Architektur umgesetzt. Sie kapselt die Funktionen der Persistenzschicht für die Präsentationsschicht und fügt diesen zusätzliche Logik hinzu.
Abbildung 4.1 zeigt die Serviceschicht im Diagramm.

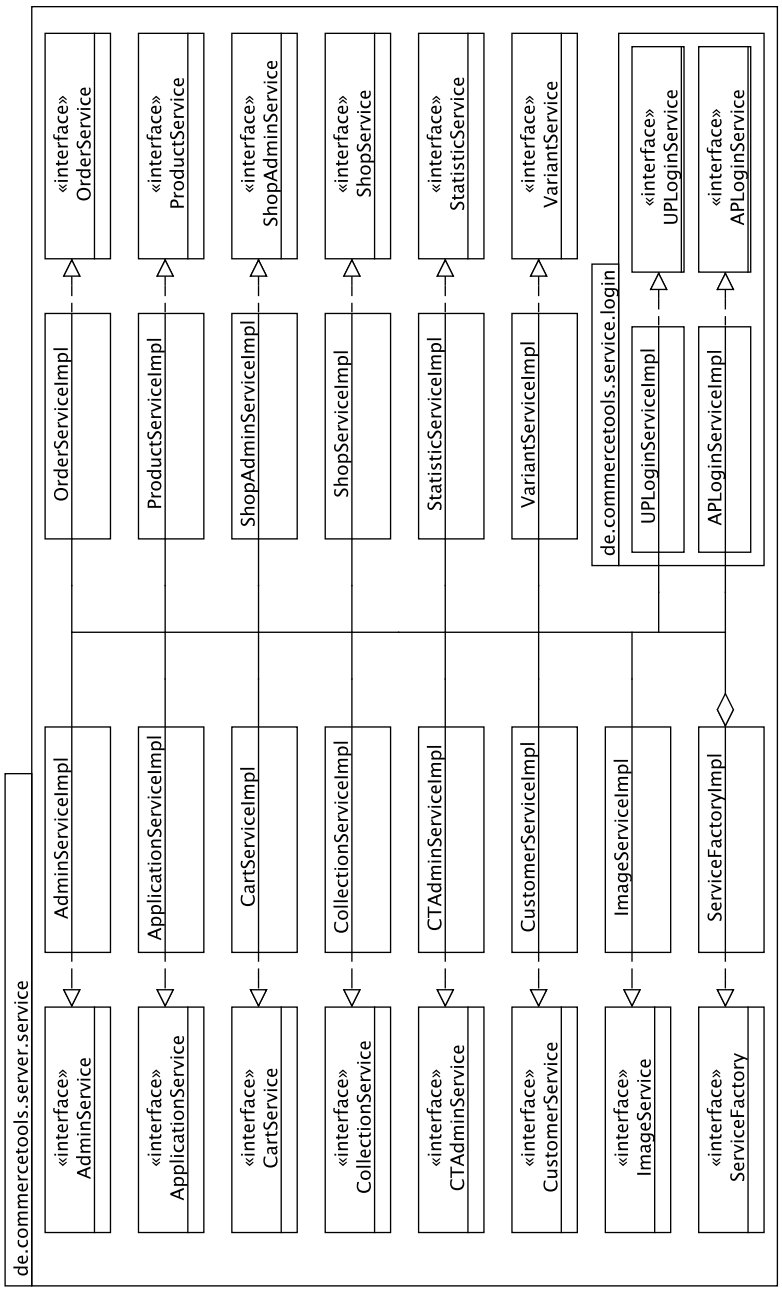


Abbildung 4.1: Die Serviceschicht von ct.Box

4.2.1 Implementierung

Wie in Abblidung 4.1 zu sehen ist, wurde für jede Ressource ein eigener Service erstellt. Die Funktionen eines jeden Service werden über ein Interface angeboten. Dadurch können die Implementierungen bei Bedarf ausgetauscht werden. Da die Services an vielen unterschiedlichen Stellen im Code gebraucht werden (sowohl in anderen Service-Klassen, als auch im Frontend), ist es sinnvoll, eine Factory-Klasse für alle Services zur Verfügung zu stellen. In dieser Factory-Klasse werden die Services anhand ihrer Implementierungen instanziiert, und die im ServiceFactory-Interface (Listing 4.12) deklarierten Getter-Methoden für sie angeboten. In Listing 4.13 ist diese Klasse dargestellt.
Um nur eine einzige Instanz jedes Services im Speicher zu halten, sind alle Variablen als statisch deklariert.

```
public interface ServiceFactory {
  public ApplicationService applicationService();
  public CollectionService collectionService();
  public CustomerService customerService();
  public ImageService imageService();
  public OrderService orderService();
  public ProductService productService();
  public ShopAdminService shopAdminService();
  public ShopService shopService();
  public UPLoginService upLoginService();
  public VariantService variantService();
  public CartService cartService();
  public StatisticService statService();
  public ValidatorFactory validatorFactory();
  public AdminService adminService();
  public APLoginService apLoginService();
  public CTAdminService ctAdminService();
}
```

Listing 4.12: Das Interface der ServiceFactory

```
public class ServiceFactoryImpl implements ServiceFactory {

  private static final ApplicationService appService = new
      ApplicationServiceImpl();
  private static final CollectionService collectionService = new
      CollectionServiceImpl();
  private static final CustomerService customerService = new
      CustomerServiceImpl();
  private static final ImageService imageService = new
      ImageServiceImpl();
  private static final OrderService orderService = new
      OrderServiceImpl();
```

```
    private static final ProductService productService = new
        ProductServiceImpl();
    private static final ShopAdminService shopAdminService = new
        ShopAdminServiceImpl();
    private static final ShopService shopService = new
        ShopServiceImpl();
    private static final UPLoginService loginService = new
        UPLoginServiceImpl();
    private static final VariantService variatnService = new
        VariantServiceImpl();
    private static final CartService cartService = new
        CartServiceImpl();
    private static final StatisticService statService = new
        StatisticServiceImpl();
    private static final AdminService adminService = new
        AdminServiceImpl();
    private static final ValidatorFactory validatorFactory;
    private static final CTAdminService ctAdminService = new
        CTAdminServiceImpl();
    private static final APLoginService apLoginService = new
        APLoginServiceImpl();

    static {
      // Build the ValidatorFactory
      Configuration<?> configuration = Validation.byDefaultProvider
          ()
          .configure();
      validatorFactory = configuration.buildValidatorFactory();
    }
    //Getter Methods for the Services
    //....
  }
```

Listing 4.13: Die Implementierung der ServiceFactory

Ähnlich wie bei dem schon vorher erwähnten Servlet-Filter für die Namespaces, wird auch hier ein Servlet-Filter eingesetzt, um für jeden Request, und damit jeden Thread, eine Thread-lokale Variable mit der ServiceFactory zu setzen. Somit hat für jeden Aufruf jede Klasse Zugriff auf die ServiceFactory, kann also jeden darin deklarierten Service nutzen.

Kommunikation mit den Frontends

Wie bereits erwähnt, bedient sich die Präsentationsschicht als Schnittstelle zu den Benutzern von ct.Box der Funktionen der Serviceschicht. Dafür ruft sie Methoden in den verschiedenen Services auf, die wiederum mit der Persistenzschicht interagieren, um Änderungen am Datenbestand durchzuführen. Für

die beiden Frontends von ct.Box unterscheiden sich diese Aufrufe allerdings grundsätzlich voneinander. Hier wird nur kurz darauf eingegangen, wie die Methoden der Serviceschicht aus den höheren Schichten aufgerufen werden. Genaueres zur Präsentationsschicht findet sich in Kapitel 4.3.

Der Webservice

Der Webservice kann in seinen Klassen direkt auf die Methoden der Serviceschicht zugreifen. Dazu muss er sich den benötigten Service aus der Threadlokalen ServiceFactory holen, und kann auf diesem dann die gewünschten Operationen ausführen. Listing 4.14 zeigt einen solchen Aufruf.

```
ProductService service = MyThreadLocal.getServiceFactory().
    productService();
List<Product> productList = service.getAll(limit, page);
```

Listing 4.14: Ein Service-Ausruf aus dem Webservice

Die Weboberflächen

Die Weboberflächen benötigen einen komplizierteren Ansatz, da sie mit dem Google-Web-Toolkit realisiert werden. In GWT können die Klassen unserer Service-Factory nicht verwendet werden, da der Java-Code für die Verwendung im Browser in Javascript konvertiert wird (siehe Abschnitt 4.3.2). Deshalb müssen die von GWT angebotenen Methoden zur Kommunikation mit dem Server verwendet werden. Dafür bestehen mehrere Möglichkeiten, die alle ihre Daten nach dem AJAX -Prinzip (Asynchronous JavaScript and XML) [13] asynchron, also nicht blockierend, an den Server schicken.

GWT-RPC Für Applikationen, die auf Serverseite Java verwenden, stehen Remote-Procedure-Calls (RPCs) zur Verfügung. Bei den RPCs werden Entities als Java-Objekte über Standard-HTTP zwischen Client und Server ausgetauscht. Dafür müssen diese Objekte serialisiert werden, also das Interface *Serializable* implementieren. Die Serializierung geschieht dann automatisch durch GWT. Um einen GWT-RPC-Service zu erstellen sind im Allgemeinen drei Schritte erforderlich. [Goo11n]

[13] http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

- Es muss ein Service-Interface, das von der Klasse *RemoteService* erbt, mit allen Methoden die per RPC aufgerufen werden sollen erstellt werden.
- Eine Klasse, die vom *RemoteServiceServlet* erbt, und das gerade erstellte Service-Interface implementiert, muss entwickelt werden.
- Ein asynchrones Interface, mit dem selben Namen des Service-Interface, gefolgt von "Async" (z.B. ProductServiceAsync), muss angelegt werden. Dieses Interface muss mit dem Service-Interface aus Schritt eins in einem Paket liegen. Außerdem müssen alle Methoden im asynchronen Interface die selbe Signatur aufweisen, wie die Methoden im Service-Interface, mit dem Unterschied, dass alle Rückgabewerte hier *void* sein müssen und als zusätzlicher Parameter ein *AsyncCallback*-Objekt übergeben wird. Dieses Callback-Objekt wird aufgerufen, sobald der Aufruf erfolgreich beendet ist, oder falls ein Fehler aufgetreten ist. Dazu muss es die Methode *onSuccess* beziehungsweise *onFailure* überschreiben.

In Listing 4.15 ist ein Beispiel für eine Implementierung von GWT-RPC zu sehen.

```
@RemoteServiceRelativePath("stockPrices")
public interface StockPriceService extends RemoteService {
  StockPrice[] getPrices(String[] symbols);
}

public interface StockPriceServiceAsync {

  void getPrices(String[] symbols, AsyncCallback<StockPrice
      []> callback);

}

public class StockPriceServiceImpl extends
    RemoteServiceServlet implements StockPriceService {
  public StockPrice[] getPrices(String[] symbols) {
    // TODO Auto-generated method stub
    return null;
  }
}
```

Listing 4.15: GWT-RPC Implementierung [Goo11n]

Ein Aufruf an einen RPC-Service geschieht dann wie in Listing 4.16 dargestellt.

```
private ArrayList<String> stocks = new ArrayList<String>();
private StockPriceServiceAsync stockPriceSvc = GWT.create(
    StockPriceService.class);

 private void refreshWatchList() {
   // Initialize the service proxy.
   if (stockPriceSvc == null) {
     stockPriceSvc = GWT.create(StockPriceService.class);
   }

   // Set up the callback object.
   AsyncCallback<StockPrice[]> callback = new AsyncCallback<
       StockPrice[]>() {
     public void onFailure(Throwable caught) {
       //Do something with errors.
     }

     public void onSuccess(StockPrice[] result) {
       updateTable(result);
     }
   };

   // Make the call to the stock price service.
   stockPriceSvc.getPrices(stocks.toArray(new String[0]),
       callback);
 }
```

Listing 4.16: GWT-RPC Aufruf [Goo11n]

Häufig wird in den Entities aber auch Code benötigt, der nicht mit GWT kompatibel ist. In diesem Fall müssen für jedes solche Entity sogenannte "Data-Transfer-Objects" (DTOs) erstellt werden, die die inkompatiblen Code-Stücke vor GWT verbergen, also nur einen Teil der Methoden und Felder des eigentlichen Entity's enthalten. Jetzt muss aber auch Code erstellt werden, der DTOs in ihre ursrpünglichen Entities konvertieren kann, und umgekehrt. Das obliegt allein dem Entwickler, GWT bietet hier keine Funktionen, die einem bei dieser Aufgabe unterstützen. Falls nun nachträglich noch Änderungen an den Entities vorgenommen werden, müssen diese natürlich auch noch in ihren DTOs nachgebessert werden. Das bedeutet Aufwand und ist fehleranfällig.

JSON via HTTP Für Server, die keine Java-Servlets unterstützen, existiert die Möglichkeit, Daten über das JSON-Format und HTTP-Requests auszutauschen.[Goo11d] Die JSON-Objekte können dann per JSNI [14] in Java-

[14] http://code.google.com/intl/de-DE/webtoolkit/doc/latest/

Script-Objekte konvertiert und im GWT-Code verwendet werden. Um Daten von einem entfernten Server abzurufen, existiert dazu noch eine Erweiterung, JSONP [15] (JSON with Padding), mit der die Same-Origin-Policy (SOP) von JavaScript umgangen werden kann. Da die App-Engine aber Java-Servlets unterstützt, werden diese Ansätze in ct.Box nicht eingesetzt und hier nicht näher darauf eingegangen. Mit dieser Methode könnte aber zum Beispiel eine mobile Applikation auf Basis des Webservices von ct.Box ein GWT-Applikation aufbauen.

RequestFactory Ein relativ neues Feature von GWT ist die RequestFactory. Seit Version 2.1 exisitiert mit ihr eine Alternative zu den RPCs. Sie wurde zuerst speziell dafür eingeführt, die Implementierung von Daten-orientierten Applikationen zu vereinfachen, kann aber mittlerweile auch allgemein als RPC-Mechanismus eingesetzt werden.[Bro11b] Im Gegensatz zu den GWT-RPCs wird bei der RequestFactory davon ausgegangen, dass die Entity-Objekte der Serverseite nicht im Client verwendet werden können, die Entities müssen also nicht in JavaScript übersetzbar sein. Für den Client werden deswegen Proxy-Objekte und Stub-Interfaces erstellt, die durch Annotationen und Konventionen in der Namensgebung auf die entsprechenden Serverobjekte abgebildet werden. Die Proxies und Stub-Interfaces sind dabei reine Interfaces, für die die RequestFactory automatisch Implementierungen generiert. Das erlaubt es auch, dass ein Server-Objekt durch unterschiedliche Proxy-Interfaces auf der Client-Seite repräsentiert wird, die jeweils ihren eigenen Teil des Server-Objekts sichtbar machen.
GWT unterscheidet hier zwei Typen von Server-Objekten: Entities und reine Wert-Objekte ("Values"). Entities unterscheiden sich dabei von den Value-Objekten durch eine eigene Identität (ausgedrückt durch ein Identifier-Feld) und zusätzlich durch eine Version. Ein Produkt in ct.Box wäre also als Entity-Objekt zu sehen, die Optionen eines Produktes wiederum als Value-Objekt, da diese keine eigen Identität besitzen, sondern als "Wert" eines Produkts gespeichert werden. Für diese beiden Typen existieren nun auch zwei Arten von Basis-Interfaces auf der Client-Seite. *EntityProxy* und *ValueProxy*.[Goo11i] Jedes Proxy-Interface muss von einem dieser beiden Interfaces erben, um mit der RequestFactory verwendet werden zu können. Man kann diese Proxies als Entsprechung der DTOs bei den GWT-RPCs sehen. Die Konvertierung von Proxy zu Server-Objekt und zurück wird zudem von der RequestFactory vorgenommen,

DevGuideCodingBasicsJSNI.html

[15] http://code.google.com/intl/de-DE/webtoolkit/doc/latest/tutorial/Xsite.html#design

was Entwicklern viel Aufwand spart. Um die Verbindung zwischen serverseitiger Klasse und Proxy-Interface herzustellen, müssen die Proxies entweder mit *@ProxyFor* oder *@ProxyForName* annotiert werden und die dazugehörige Server-Klasse angegeben werden.[Goo11i]
In Listing 4.17 sind solche Proxy-Klassen am Beispiel des Produkts und seiner Optionen gezeigt.

```
public interface DatastoreObjectProxy extends EntityProxy {

  public void setId(Long id);
  public Long getId();
  public Integer getVersion();
  public void setVersion(Integer version);

  // Getter/Setter for created_at, updated_at...

}
 @ProxyFor(value = Product.class, locator = ObjectifyLocator.
     class)
public interface ProductProxy extends DatastoreObjectProxy {

  public String getTitle();
  public void setTitle(String title);
  public List<OptionProxy> getOptions();
  public void setOptions(List<OptionProxy> options);
  public List<VariantProxy> getVariants();
  public void setVariants(List<VariantProxy> variants);
  //More Getters and Setters...
}

@ProxyFor(value=Option.class)
public interface OptionProxy extends ValueProxy{

  public String getName();

  public void setName(String name);

  public Set<String> getValues();

  public void setValues(Set<String> values);

}
```

Listing 4.17: Die Proxy-Interfaces für ein Produkt und seine Optionen

Anstatt der normalen Objekte werden in den Proxy-Interfaces immer die zugehörigen Proxies verwendet. In diesem Beispiel gibt also die Metho-

de *getVariants()* nicht eine Liste von Varianten, sondern eine Liste vom Typ *VariantProxy* zurück. Neben Proxy-Objekten kann die RequestFactory noch alle primitiven Datentypen, ihre zugehörigen Wrapper-Klassen (z.B. *java.lang.Long*), Strings, Enums, Datumsangaben (*java.util.Date*, *BigInteger*, *BigDecimal* und Listen und Sets der gerade genannten Typen transportieren.[Goo11i] So kann zum Beispiel, obwohl der *byte*-Typ unterstützt wird, kein *byte*-Array verwendet werden. Die Proxy-Interfaces erlauben es der RequestFactory, nur wirklich vorgenommene Änderungen an Entities an den Server zu schicken. Dazu überwacht sie auf Client-Seite die Aufrufe an die Setter-Methoden der Proxies, und sendet dann nur die geänderten Werte zum Server. Dort werden die Setter-Methoden noch einmal auf die Server-Objekte angewandt. Hier kommen der Identifier und die Version der Entities ins Spiel. Die RequestFactory lädt ein Entity anhand ihres Identifiers aus dem Datastore (entweder über ein separates *Locator*-Objekt, oder eine statische Methode auf dem Entity) und erkennt durch das Versions-Feld, ob Änderungen an dem Objekt durchzuführen sind. Dazu muss die Version eines Entities jedesmal, wenn es gespeichert wird, erhöht werden. Das geschieht mit Hilfe des *@PrePersist*-Callbacks von Objectify.
Da Objectify mit der *DAOBase*-Klasse schon eine Basis-Klasse für den Zugriff auf die Entities anbietet, gestaltet sich die Implementierung einer Locator-Klasse für unsere Entities relativ einfach (siehe Listing 4.18).

```
public class DataStoreObjectLocator extends Locator<
    DataStoreObject, Long> {
  @Override
  public DataStoreObject create(Class<? extends
      DataStoreObject> clazz) {
    try {
      return clazz.newInstance();
    } catch (InstantiationException e) {
      throw new RuntimeException(e);
    } catch (IllegalAccessException e) {
      throw new RuntimeException(e);
    }

  }

  @Override
  public DataStoreObject find(Class<? extends DataStoreObject
      > clazz, Long id) {
      DAOBase daoBase = new DAOBase();
      return daoBase.ofy().find(clazz, id);
  }

  @Override
```

```
    public Class<DataStoreObject> getDomainType() {
      // never called
      return null;
    }

    @Override
    public Long getId(DataStoreObject domainObject) {
      return domainObject.getId();
    }

    @Override
    public Class<Long> getIdType() {
      return Long.class;
    }

    @Override
    public Object getVersion(DataStoreObject domainObject) {
      return domainObject.getVersion();
    }
  }
```

Listing 4.18: Die DatastoreObjectLocator-Klasse

Wie man sehen kann, muss ein Locator die Methoden *create*, *find*, *getDomainType*, *getId*, *getIdType* und *getVersion* implementieren. Nachdem die meisten unserer Objekte eine Subklasse der *DataStoreObject*-Klasse sind und somit ein Long-Feld als *@Id* besitzen, kann für alle diese Klassen der selbe Locator verwendet werden. Für die wenigen anderen Klassen wurden eigene Locator-Klassen erstellt, die sich allerdings nur durch den Typ ihrer Id vom DataStoreObjectLocator unterscheiden.

Nachdem nun die Entities mit der Requestfactory verwendet werden können, müssen noch die Services umgesetzt werden. Dafür gibt es wieder zwei unterschiedliche Optionen. Entweder die Service-Funktionen werden als statische Methoden auf den Entities selbst realisiert, oder als Methoden auf einer separaten Service-Klasse, die durch einen sogenannten *ServiceLocator* bereitgestellt wird. In beiden Fällen wird ein Service in einem Stub-Interface deklariert, der das Interface *RequestContext* erweitert. Mit der *@Service*-Annotation auf dieser Klasse wird dann die Klasse auf dem Server angegeben, die den Service implementiert. Also entweder ein Entity mit den statischen Methoden, oder eine eigene Service-Klasse. Da in ct.Box schon eine eigene Service-Schicht vorhanden ist, können diese Service-Klassen direkt für die RequestFactory verwendet werden. Dazu werden einfach die Implementierungen dieser Services in der *@Service*-Annotation des dazugehörigen Stub-Interface angegeben.

Listing 4.19 verdeutlicht die Implementierung an einem Beispiel.

```
@Service(value = ProductServiceImpl.class, locator =
    MyServiceLocator.class)
public interface ProductRequest extends RequestContext {
  Request<List<ProductProxy>> getAll(int limit, int page);
  Request<List<ProductProxy>> getAllSorted(int limit, int
      page, String sorting, String valueToOrder);
  Request<Integer> getCount(Date since);
  Request<ProductProxy> get(Long id);
  Request<ProductProxy> create(ProductProxy product);
  Request<ProductProxy> update(Long id, ProductProxy update);
  Request<Void> delete(Long id);
  //...

}
```

Listing 4.19: Das Service-Interface für die Produkte

Die Signaturen der Methoden entsprechen denen in den Service-Implementierungen, mit dem Unterschied, dass keine Entities zurückgegeben werden, sondern Subklassen der *Request*-Klasse. Das erlaubt das asynchrone Aufrufen der Methoden, ähnlich der Asynchronen Callbacks bei den RPCs. Wären die Services nun nicht in einer eigenen Service-Klasse, sondern im Entity selbst, in diesem Fall damit im Produkt, implementiert, so müsste der Verweise in der *@Service*-Annotation *value=Product.class* lauten und auch kein ServiceLocator angegeben werden. In diesem Fall wird aber ein ServiceLocator benötigt, um eine Instanz einer Service-Implementierung erstellen zu können. Eine Klasse, die das *ServiceLocator*-Interface implementiert, kann dabei folgendermaßen aussehen:

lstsetlanguage=Java

```
public class MyServiceLocator implements ServiceLocator {
  @Override
  public Object getInstance(Class<?> clazz) {
    try {
      return clazz.newInstance();
    } catch (InstantiationException e) {
      throw new RuntimeException(e);
    } catch (IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Listing 4.20: Die ServiceLocator-Klasse [Goo11i]

Diese Klasse kann für alle Service-Stubs eingesetzt werden. Ähnlich wie bei der *ServiceFactory* werden auch für die RequestFactory die Services in einer zentralen Klasse zusammengefasst. Dazu implementiert diese Klasse das *RequestFactory* Interface und bietet Methoden mit den einzelnen Stub-Services als Rückgabewert an. Listing 4.21 zeigt einen Ausschnitt der RequestFactory-Klasse von ct.Box.

```
public interface BoxRequestFactory extends RequestFactory {
  CollectionRequest collectionRequest();
  OrderRequest orderRequest();
  ProductRequest productRequest();
  VariantRequest variantRequest();
  // More Requests ...
}
```

Listing 4.21: Die RequestFactory-Klasse

Nachdem nun Services und Entities für die RequestFactory umgesetzt wurden, können aus dem GWT-Client-Code heraus Aufrufe an die Service-Schicht von ct.Box gesendet werden. Hierfür muss natürlich zuerst die RequestFactory initialisiert werden. Die Initialisierung erfolgt dabei folgendermaßen:

```
final EventBus eventBus = new SimpleEventBus();
BoxRequestFactory requestFactory = GWT.create(
    BoxRequestFactory.class);
requestFactory.initialize(eventBus);
```

Listing 4.22: Initialisierung der RequestFactory

Danach kann man sich das passende Interface von der RequestFactory holen. Auf diesem können dann die Service-Methoden aufgerufen werden. Dies ist in Listing 4.23 abgebildet. Das *clientFactory*-Objekt, das hier verwendet wird, um eine Instanz der RequestFactory zu bekommen, kann dabei ähnlich wie die bereits beschriebene *ServiceFactory*-Klasse gesehen werden. Sie wird in Kapitel 4.3.2 noch genauer beschrieben.

```
    ProductRequest request = clientFactory.getRequestFactory().
        productRequest();
    Request<ProductProxy> r = request.get(someProductId).with("
        options", "variants", "images");
    r.fire(new Receiver<ProductProxy>() {
        @Override
        public void onSuccess(ProductProxy response) {
          //The call was successful, do something with the
              response
```

```
        }

        @Override
        public void onFailure(ServerFailure error) {
          //An error occurred, this is the place for Exception
              Handling
        }

        @Override
        public void onViolation(Set<Violation> errors) {
          //The Validation of the sent product failed
          //Invalid fields are contained in the Set<Violation>
        }
      });
```

Listing 4.23: Aufruf einer Service-Methode mit der RequestFactory

Ein *Request* wird über die *fire()*-Methode an den Server gesendet. Diese Methode nimmt als Parameter eine *Receiver*-Klasse, deren Typ der Rückgabewert des Requests ist. Diese Receiver-Klasse ist das Pendant der AsyncCallbacks der RPCs. Ist der asynchrone Aufruf erfolgreich, wird die *onSuccess*-Methode aufgerufen, bei einem Fehler die *onFailure*-Methode. Die *onViolation*-Methode kann überschrieben werden, um das Verhalten der GWT-Applikation bei einer fehlgeschlagenen Validierung des versandten Objekts anzupassen.
Da der Aufruf asynchron ausgeführt wird, werden alle Operationen, die nach dem *fire()* kommen, parallel ausgeführt. Man muss als Entwickler also beachten, dass der genaue Zeitpunkt, an dem der Aufruf beendet wird, nicht bekannt ist. Soll der Request blockierend ausgeführt werden, so müssen alle nachfolgenden Operationen innerhalb der onSuccess-Methode stehen. Nur dann ist sichergestellt, dass sie erst nach Beendigung des asynchronen Aufrufs ausgeführt werden. An diesem Beispiel erkennt man noch eine weitere Besonderheit der RequestFactory. Sie baut bei Anfragen an den Server nicht automatisch den gesamten Objektgraphen auf. Wird also wie in Listing 4.21 die *get*-Methode für ein Produkt aufgerufen, so wird im Standardfall auch nur das Produkt zurückgegeben, während das Feld für die Varianten des Produkts leer bleibt. Um dieses Verhalten zu ändern, existiert die *with()*-Methode auf der Request-Klasse. In dieser Methode können die Namen der Felder angegeben werden, die automatisch von der RequestFactory befüllt werden sollen. Auch Felder vom Typ *ValueProxy* müssen in der with-Methode angegeben werden, wenn sie mit zurückgegeben werden sollen.
Um all diese Aktivitäten auszuführen, verwendet die RequestFactory ein

Servlet, das *RequestFactoryServlet*. An dieses Servlet werden alle Anfragen durch die RequestFactory gesendet. Dazu muss das Servlet in der web.xml-Datei einer Web-Applikation wie in Listing 4.24 eingebunden werden.

```
<!-- RequestFactoryServlet -->
<servlet>
  <servlet-name>requestFactoryServlet</servlet-name>
  <servlet-class>com.google.web.bindery.requestfactory.server
      .RequestFactoryServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>requestFactoryServlet</servlet-name>
  <url-pattern>/gwtRequest</url-pattern>
</servlet-mapping>
```

Listing 4.24: Das Einbinden der RequestFactory in ein Projekt

GWT-RPC vs. RequestFactory

Mit den Remote-Procedure-Calls und der RequestFactory stehen also zwei Features für die Implementierung der Services für GWT in ct.Box zur Verfügung. Für ct.Box fiel die Entscheidung auf die RequestFactory. Das hat mehrere Gründe. Zum einen können die Services der bereits implementierten Service-Schicht direkt verwendet werden. Dafür sind nur Interfaces und ein einfacher Service-Locator zu erstellen, anstatt kompletter Servlets, die auch noch einzeln in der web.xml deklariert und gemappt werden müssten.
Andererseits erscheint bei den RPCs die Verwendung der Entities zunächst leichter, da dort sowohl im Client als auch auf dem Server die selben Objekte verwendet werden können. Sobald aber Inkompatibilitäten zu GWT in den Entities existieren, wird die Umsetzung mit den RPCs schwieriger. Wie bereits erwähnt müssen dann DTOs erstellt werden, komplette Java-Klassen die das Verhalten der realen Entities für GWT kapseln. Diese DTOs werden dann zur Kommunikation zwischen Client und Server verwendet, und es muss Code zur Konvertierung von DTO zu Entity und zurück geschrieben werden. Im Gegensatz dazu verwendet die RequestFactory auf der Client-Seite automatisch generierte Proxy-Objekte. Dafür müssen wie gezeigt nur Interfaces erstellt werden, was weniger duplizierten Code und damit auch eine geringere Fehlerwahrscheinlichkeit bedeutet. Dazu werden nur die Änderungen an Entities zwischen Client und Server verschickt, was höhere Performance und geringere Netzlast bedeutet und besonders im Hinblick auf die Quotas der App-Engine hilfreich ist.

Validierung von Entities

Eine weitere Funktion der Serviceschicht ist die Validierung von Entities.
Es ist selbstverständlich, dass Entities, beziehungsweise manche deren Felder, gewissen Einschränkungen (Constraints) unterliegen müssen. So muss ein Produkt einen Titel haben, unter dem es angeboten werden kann. Oder der Preis eines Produkts muss einen sinnvollen Wert (größer 0) aufweisen. Um eine zentrale Validierung der Entities zu implementieren, verwendet ct.Box den *Hibernate Validator*. [Com] Der Hibernate-Validator ist eine Implementierung des Java-Specification-Requests 303 (JSR-303). Diese Spezifikation definiert eine API und das dazugehörige Metamodell zur Validierung von Java-Beans. Dabei werden Annotationen, oder XML-Konfigurationsdateien zur Erstellung der Metadaten verwendet. [Gro09a]
Um die Entities möglichst frei von zusätzlichem Code zu halten und die Constraints möglichst austauschbar zu machen, wird in ct.Box die Konfiguration der Validierung über XML-Dateien vorgenommen. Dafür muss die Datei *META-INF/validation.xml* vorhanden sein, in der folgende Einstellungen vorgenommen werden müssen: [Gro09a]

default-provider Der Klassenname der Implementierung des *ValidationProvider*-Interface. Beim Hibernate Validator lautet dieser `org.hibernate.validator.HibernateValidator`.

message-interpolator Die Implementierung des *MessageInterpolator*-Interface. Der Hibernate Validator implementiert dieses in der Klasse `org.hibernate.validator.messageinterpolation.ResourceBundleMessageInterpolator`.

traversable-resolver Die Implementierung des *TraversableResolver*-Interface. Beim Hibernate-Validator ist dies der `org.hibernate.validator.engine.resolver.DefaultTraversableResolver`.

constraint-validator-factory Die Implementierung des *ConstraintValidatorFactory*-Interface. Beim Hibernate-Validator entspricht das der Klasse `org.hibernate.validator.engine.ConstraintValidatorFactoryImpl`.

constraint-mapping Die Pfade zu den einzelnen XML-Mapping Dateien, die dazu verwendet werden, die Constraints auf den Entities festzulegen.

ct.Box besitzt eine validation.xml-Datei wie in Listing 4.25 dargestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
  xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/
        configuration validation-configuration-1.0.xsd">

    <default-provider>
      org.hibernate.validator.HibernateValidator
    </default-provider>
    <message-interpolator>
      org.hibernate.validator.messageinterpolation.
          ResourceBundleMessageInterpolator
    </message-interpolator>
    <traversable-resolver>
      org.hibernate.validator.engine.resolver.
          DefaultTraversableResolver
    </traversable-resolver>
    <constraint-validator-factory>
      org.hibernate.validator.engine.ConstraintValidatorFactoryImpl
    </constraint-validator-factory>

    <constraint-mapping>META-INF/validation/product-constraints.xml<
        /constraint-mapping>
    <constraint-mapping>META-INF/validation/variant-constraints.xml<
        /constraint-mapping>
    <constraint-mapping>META-INF/validation/collection-constraints.
        xml</constraint-mapping>
  </validation-config>
```

Listing 4.25: Die Datei META-INF/validation.xml in ct.Box

Die Registrierung der Implementierung erfolgt über den *ValidationProvider*, der als Java Service-Provider-Interface deklariert ist. [Gro09a] Dazu muss sich die Jar-Datei des Hibernate-Validators im Classpath des Projekts befinden, und zusätzlich eine Textdatei im Ordner META-INF/services angelegt werden, die den vollständigen Namen der ValidationProvider-Klasse als Namen trägt (*javax.validation.spi.ValidationProvider*). [O'C09] In dieser Datei werden dann alle verfügbaren Implementierung des SPIs angegeben, im Fall von ct.Box ist das nur die Klasse *org.hibernate.validator.HibernateValidator*. Damit ist der Hibernate-Validator einsatzbereit. Es müssen nun noch die Constraints für die jeweiligen Entities festgelegt werden. Das geschieht in eigenen XML-Dateien, die in der validation.xml-Datei über die `<constraint-mapping>` -Tags eingebunden werden. Den Aufbau einer solchen Datei zeigt Listing 4.26.

```
  <?xml version="1.0" encoding="UTF-8"?>
  <constraint-mappings xmlns="http://jboss.org/xml/ns/javax/
      validation/mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/
        mapping validation-mapping-1.0.xsd">
```

```
    <default-package>de.commercetools.server.persistency</default-
        package>
    <bean class="Product" ignore-annotations="true">
      <field name="title">
        <constraint annotation="org.hibernate.validator.constraints.
            NotEmpty"></constraint>
      </field>
      <field name="vendor">
        <constraint annotation="org.hibernate.validator.constraints.
            NotEmpty"></constraint>
      </field>
      <field name="product_type">
        <constraint annotation="org.hibernate.validator.constraints.
            NotEmpty"></constraint>
      </field>
      <field name="options">
        <constraint annotation="org.hibernate.validator.constraints.
            NotEmpty"></constraint>
      </field>
    </bean>
  </constraint-mappings>
```

Listing 4.26: Definition von Entitiy-Constraints in der product-constraints.xml

Zuerst wird das *default-package* festgelegt, das den Paketnamen für alle nicht vollständig benannten Klassen definiert. Danach kommt die Bean-Definition für das Product, in der gleichzeitig festgelegt wird, dass alle die Validierung betreffenden Annotationen auf der Product-Klasse ignoriert werden sollen. Damit sind allein die in dieser XML-Datei festgelegten Constraints gültig. Innerhalb der `<bean>`-Tags können dann die Constraints wie hier gezeigt mit den `<field>`-Tags auf den Feldern einer Klasse, oder auch mit Hilfe des `<class>`-Tags auf der Klasse selbst angewandt werden. Über die `<constraint>`-Tags werden die einzelnen Constraints angegeben. Dabei können sowohl die bereits im Hibernate Validator vorhandenen Constraints [16] verwendet, als auch eigene erstellt werden. Die Standard-Constraints sind in den meisten Fällen allerdings ausreichend. Der hier verwendete *NotEmpty*-Constraint bedeutet bespielsweise, dass der Wert eines Feldes nicht *null* oder der leere String sein darf.
Jeder Constraint ist weiterhin Teil einer *Gruppe* (group) von Constraints, die zusammen angewandt werden. Dabei können Constraints auch in mehreren Gruppen vertreten sein. Ein Constraint kann einer Gruppe durch die `<group>`-Tags innerhalb eines `<constraint>`-Tags zugewiesen werden. Rein technisch gesehen ist eine Gruppe nur ein leeres Interface. Wird ein Constraint keiner bestimmten Gruppe zugewiesen, wie in Listing 4.26, ist er in der *Default*-Gruppe

[16] `http://docs.jboss.org/hibernate/validator/4.2/api/org/hibernate/validator/constraints/package-summary.html`

enthalten. Durch die Gruppen können so für jede Validierung unterschiedliche Constraints angewandt werden.
Um nun im Code ein Objekt zu validieren, muss zuerst die *ValidatorFactory*-Klasse initialisiert werden. Das geschieht, wie in Listing 4.13 bereits gezeigt, in der ServiceFactory. Dort wird ein statischer Block zum Aufbau der ValidatorFactory verwendet, da dies relativ teuer ist. Über die ValidatorFactory können *Validator*-Objekte gewonnen werden, die Thread-sicher und damit wiederverwendbar sind. Die Validierung der Entities erfolgt schließlich über die *validate()*-Methode der Validator-Klasse. Diese Methode gibt in einem Set alle möglicherweise stattgefunden Verletzungen der Constraints zurück. Ist dieses Set leer, so ist das Objekt gültig. In Listing 4.27 ist die Vorgehensweise noch einmal veranschaulicht.

```
@Override
public Set<ConstraintViolation<Product>> validate(Product product)
     {
  ValidatorFactory factory = MyThreadLocal.getServiceFactory().
      validatorFactory();
  // The container uses the factory to validate constraints
  Validator validator = factory.getValidator();
  return validator.validate(product, Default.class);
}
```

Listing 4.27: Aufruf der Validierung im ProductService

Wie man sieht, muss für jeden Aufruf von *validate()* die Gruppe ("ValidationGroup") mit angegeben werden, die zu Validierung verwendet werden soll. Da in ct.Box im Moment keine expliziten Gruppen verwendet werden, ist hier die Default-Gruppe angegeben.
Vor jedem Aufruf an ct.Box, der Entities speichern oder ändern soll, werden die übergebenen Objekte validiert. Das geschieht für die beiden Frontends unterschiedlich. Im Webservice muss vor diesen Requests die *validate()*-Methoden der einzelnen Services aufgerufen werden, und darf nur dann die geforderten Aktionen ausführen, falls keine Verletzung der Constraints vorliegt.
Aufrufe über die RequestFactory werden allerdings automatisch validiert. Sobald ein ValidationProvider im Projekt installiert ist, verwendet die RequestFactory diesen, um jeden Aufruf an den Server zu validieren.
Das ist einerseits bequem, hat jedoch auch unangenehme Nebenwirkungen, die im nächsten Abschnitt genauer besprochen werden.

4.2.2 Erkenntnisse

Im Vergleich zur Persistenzschicht hat sich die Implementierung der Serviceschicht als komplizierter herausgestellt. Die Basis-Services und ihre Implemen-

tierung konnten zwar mit Hilfe von Objectify schnell erstellt werden, und auch deren Umsetzung für GWT ist relativ schnell geschehen. Allerdings hat sich die Nutzung der RequestFactory als ein wenig umständlich erwiesen.
Die Proxy-Objekte können im Client nämlich nur innerhalb eines sogenannten Contexts (eines Requests) verwendet werden. Sobald dieser Request abgefeuert wurde, ist der dazugehörige Proxy "eingefroren". Um Änderungen an diesem Objekt vorzunehmen, muss zuerst ein neuer Request (und damit ein neuer Context) erstellt werden, und auf diesem dann seine *edit()*-Methode aufgerufen werden. Allerdings konnte dieses Verhalten bei der Implementierung nicht nachvollzogen werden . Scheinbar willkürlich wurden von GWT Exceptions mit dem Titel "Autobean has been frozen", aber ohne genauere Angaben zum Grund oder der Position im Code, generiert, selbst wenn nicht mehr mit den Proxies selbst, sondern mit den primitiven Werten derer Felder gearbeitet wurde. Auch ein neuer Context und die *edit*-Methode halfen nicht, diesen Fehler zu umgehen. Dazu konnte Code, der auf dem lokalen Entwicklungsserver der App-Engine eine Fehlermeldung generiert, nach dem Deploy auf die Live-Version der GAE ohne Fehler ausgeführt werden. Scheinbar ist die Kompatibilität der relativ neuen RequestFactory mit dem lokalen Server noch nicht zu hundert Prozent gegeben.
Eine weitere, wenn auch nur geringfügige, Schwierigkeit ist das Laden eines Objektgraphen mit der RequestFactory. Vergisst man als Entwickler beim Abschicken eines Requests die *with()*-Methode, um zum Beispiel die Varianten eines Produkts mit aus dem Datastore zu laden, so wird zur Laufzeit eine *NullPointerException* geworfen, die aber schnell behoben werden kann.
Zusätzlich müssen die Proxy-Interfaces per Hand gepflegt werden. Ergeben sich in den Entities oder Services Änderungen, müssen diese immer auch in den dazugehörigen Interfaces manuell angewandt werden. Wird das nicht eingehalten, so ergibt sich auch erst zur Laufzeit der Applikation ein Fehler, allerdings mit einer Meldung, die aussagekräftig genug ist, um diesen Fehler sofort beheben zu können.
Auch bei der Validierung der Entities traten Komplikationen auf. Ist eine Implementierung eines JSR-303 konformen Validators im Projekt installiert, so wird diese automatisch von der RequestFactory verwendet, um bei jedem Request noch vor dem Aufruf der Server-Methoden die versendeten Proxies auf ihre Gültigkeit zu überprüfen. Bei ungültigen Entities wird dann anstatt der *onSuccess()* die *onViolation()*-Methode der Receiver Klasse (siehe Listing 4.21) aufgerufen, die entsprechendes Exception-Handling ermöglicht. An sich ist das ein sehr nützliches Feature, das allerdings seine Tücken hat.
Will man zum Beispiel nur ein Feld eines Produkts ändern, würde es reichen, die *update*-Methode im *ProductService* mit einem ProductProxy aufzurufen, das nur dieses Feld gesetzt hat. Da aber jeder Aufruf validiert wird. dürfen die in

der *product-constraints.xml* definierten *NotEmpty*-Felder nicht *null* sein. Durch die Validation-Groups wäre es zwar möglich, für jeden Validierungsvorgang einzeln die anzuwendenden Constraints festzulegen, allerdings unterstützt die RequestFactory diese Gruppen nicht, sondern verwendet immer die Default-Gruppe, in der alle Constraints enthalten sind. Als Konsequenz müssen die überwachten Felder der Proxies für jeden Aufruf sinnvolle Werte enthalten, wodurch natürlich auch mehr Daten als eigentlich nötig zum Server gesendet werden. Außerdem werden die Signaturen der Methoden, die die Services aufrufen, unnötig "aufgebläht", da jedesmal die Werte der benötigten Felder mit übergeben werden müssen. Dazu kommt noch, dass die Validierung nicht in der Service-Layer erfolgen kann, da sie sonst doppelt vorgenommen werden würde. Dadurch muss später auch im Webservice die Validierung angestoßen werden, bevor die Methoden der Serviceschicht verwendet werden dürfen.
Generell lässt sich sagen, dass die RequestFactory zwar eine performante Lösung zur Kommunikation zwischen Client und Server ist, dafür jedoch größere Komplexität bei der Implementierung in Kauf genommen werden muss.
Hier hat sich die lokale Entwicklungsumgebung der App-Engine als äußerst hilfreich erwiesen. So konnten die aufgetretenen Fehler, mit Ausnahme der erwähnten Ungereimtheiten beim Einsatz der Proxies, schnell beseitigt werden, ohne jedes Mal das komplette Projekt in Google's Cloud hochladen zu müssen.

4.3 Präsentationsschicht

Die Präsentationsschicht stellt das User-Interface (UI), die Schnittstelle zwischen Applikation und Benutzer, dar. Für ct.Box ist der Benutzer eine Person, die mit Hilfe von ct.Box eine Web-Shop-Lösung erstellen möchte. Dazu werden ihm zwei Schnittstellen angeboten: Ein Webservice und eine Weboberfläche. Zusätzlich gibt es eine separate Webseite für die Verwaltung von ct.Box, die nur von Angestellten der Firma commercetools genutzt werden kann. Dieses Kapitel beschreibt die Realisierung der Präsentationsschicht und die dafür verwendeten Technologien.

4.3.1 Webservice

Der Webservice ist das Kernstück von ct.Box. Durch ihn wird es möglich, Shop-Lösungen auf vielen unterschiedlichen Platformen zu errichten, solange diese das HTTP-Protokoll unterstützen. Der Webservice wurde dafür nach dem Prinzip des Representational State Transfer (REST) aufgebaut, das bereits in der Einleitung (Kapitel 2.5) erläutert wurde.

Frameworks

Um die Umsetzung eines REST-konformen ("RESTful") Webservices zu vereinfachen, wurden einige Frameworks entwickelt. Die wichtigsten beiden sind dabei sicherlich Restlet [17] und Jersey [18].

Jersey [19] ist die Referenz-implementierung der Java-Spezifikation JAX-RS (JSR 311). [Gro09b] In dieser Spezifikation wird eine API zur Erstellung von RESTful-Webservices definiert. Diese API verwendet Annotationen, um POJOs auf REST-Ressourcen abzubilden. So wird der Identifikator einer Ressource über die *@Path*-Annotation angegeben. Zur Identifikation der zu verwendenden Methoden für die einzelnen HTTP-Requests stehen die Annotationen *@Get*, *@Post*, *@Put* und *@Delete* zur Verfügung, zur Angabe der MIME-Typen der Rückgabewerte der Methoden die *@Produces*-Annotation. Um schließlich noch festzulegen, welche Media-Typen in den Requests akzeptiert werden, kann die *@Consumes*-Annotation verwendet werden. Hat man seine Ressourcen entsprechend definiert, muss noch ein spezielles Java-Servlet in der web.xml der Web-Applikation eingebunden werden, das die ankommenden Requests an die entsprechenden Ressourcen weiterleitet. Bei der Deklaration dieses Servlets wird weiterhin das Paket, in dem die Ressourcen-Klassen liegen, als Parameter angegeben. Das `<servlet-mapping>`-Element dieses Servlets gibt gleichzeitig die Basis-URL für den Webservice an. An diese werden die *@Path*-Werte der Ressourcen angehängt um für diese ihre *Uniform Resource Identifiers* (URIs) zu generieren. Dies ist in Listing 4.28 zu sehen.

```
<web-app>

 <servlet>
   <servlet-name>Jersey Web Application</servlet-name>
   <servlet-class>com.sun.jersey.spi.container.servlet.
       ServletContainer</servlet-class>
     <init-param>
       <param-name>com.sun.jersey.config.property.packages</
           param-name>
       <param-value>org.foo.rest;org.bar.rest</param-value>
     </init-param>
 </servlet>

 <servlet-mapping>
   <servlet-name>Jersey Web Application</servlet-name>
   <url-pattern>/rest/*</url-pattern>
```

[17] http://www.restlet.org/
[18] http://jersey.java.net/
[19] http://jersey.java.net/

```
    </servlet-mapping>

  </web-app>
```

Listing 4.28: Einbinden von Jersey [Jer]

Um Jersey in das Projekt einzubinden, müssen dazu natürlich noch die benötigten Jar-Dateien (insgesamt 11 Stück mit ca 2.6 MB) in das /war/WEB-INF-Verzeichnis der WebApp kopiert werden.
Jersey stellt außerdem eine umfangreiche Dokumentation in Form eines "User Guide" [20] zur Verfügung, der zudem Beispiele zu den gebräuchlichsten Aktivitäten von Jersey enthält.

Restlet [21] ist ein weiteres Framework, das sich zum Ziel gesetzt hat, die Entwicklung von REST-konformen Webservices zu erleichtern. Seit Version 2.0 unterstützt es ebenfalls JSR-311 und kann deshalb genau wie Jersey konfiguriert und eingesetzt werden. Restlet besteht grundsätzlich aus zwei Teilen: Einer neutralen API, die das Bearbeiten von Requests nach REST ermöglicht, und der Restlet Engine, der Referenz-Implementierung dieser API. Weiterhin ist Restlet sowohl für Client-, als auch für Server-Applikationen geeignet und verwendet auf beiden Seiten die selbe API was die Lernkurve deutlich senkt. Darüber hinaus existieren mehrere Versionen von Restlet, die auf unterschiedliche Entwicklungsumgebungen abzielen. So gibt es Versionen für die Java Standard Edition, die Java Enterprise Edition, das Google Web-Toolkit, Android und die Google App-Engine. Besonders die für die GAE entwickelte Version macht es für die Implementierung von ct.Box interessant. In dieser Version läuft die Restlet Engine mit einem Java-Servlet-Container als Server-Connector.
[Wika] Neben der JSR-311 Implementierung modelliert Restlet die Kernkomponenten von REST (Ressourcen, Repräsentationen, Komponenten, Konnektoren,...) als eigene Java-Artefakte. Für die Implementierung am wichtigsten sind dabei die *Application*- und die *Resource*-Klasse. Logisch gesehen bildet eine *Application* in Restlet URIs auf Ressourcen ab, die durch die *Resource*-Klassen repräsentiert werden. Das geschieht durch die *Router*-Klasse. In den Ressourcen können dann die auf die HTTP-Methoden gemappten CRUD-Operationen behandelt werden.
Zusätzlich zu den Kernfunktionalitäten (Restlet API + Engine) ist noch eine große Anzahl von Erweiterungen für Restlet vorhanden, die auf der Restlet API aufsetzen. So gibt es zum Beispiel eine Erweiterung zum

[20] http://jersey.java.net/nonav/documentation/latest/user-guide.html
[21] http://www.restlet.org/

automatischen Serialisieren und Deserialisieren von Objekten ins JSON-Format mit Hilfe des Jackson JSON-Prozessors [22]. Diese Erweiterung wird auch in ct.Box verwendet, um die Server-Entities für die Antworten des Webservice automatisch in das JSON-Format zu konvertieren, und umgekehrt, um die erhaltenen JSON-Objekte in Java-Objekte umzuwandeln. Dazu ist zusätzlich die JSON-Erweiterung notwendig, die es Restlet ermög?-licht, JSON-Objekte als Repräsentationen für Ressourcen zu verwenden. Eine weiter Erweiterung ist der bereits erwähnte Servlet-Connector, durch den Restlet auf der App-Engine lauffähig ist. Die Installation solcher Erweiterungen geschieht dabei einfach durch das Einbinden der dazugehörigen Jar-Dateien in das Projekt.

Wägt man diese beiden Frameworks gegeneinander auf, so kann Jersey vor allem durch seine einfache und klare API und den damit verbundenen geringen Aufwand bei der Implementierung punkten. Außerdem ist die Dokumentation umfangreicher und besser verständlich gestaltet.

Restlet bietet offensichtlich einiges mehr an Features, besonders durch die große Anzahl von Erweiterungen, bringt damit aber logischerweise auch größere Komplexität mit sich. Die einfache Annotation der Entities ist außerdem im Fall von ct.Box nicht ausreichend, da in den Ressourcen auch Aufgaben wie Validierung und damit auch Exception-Handling erledigt werden muss. Die vielen Erweiterungen, und auch die speziell für die App-Engine bereitgestellte Version, haben den Ausschlag zur Entscheidung für Restlet zur Implementierung des Webservices für ct.Box gegeben. Da Restlet ebenfalls die Spezifikation von JSR-311 erfüllt, ist es im Nachhinein immer noch möglich, auch mit Restlet auf die einfachere Implementierung der Services umzusteigen.
Abbildung

Implementierung

Wie bereits angesprochen, ist die *Application*-Klasse in Restlet der zentrale Punkt, um das Interface des Webservice festzulegen. Dafür dient die Methode *createInboundRoot()*, die ein Objekt der *Restlet*-Klasse zurückgibt, das alle eingehenden Aufrufe an den Webservice empfängt. Ein *Restlet* ist dabei eine allgemein Superklasse für alle Klassen, die Aufrufe überhaupt verarbeiten können. Im Normalfall ist dies ein *Router*-Objekt, an das, wie in Listing 4.29 gezeigt, über die *attach()*-Methode einzelne Ressourcen mit ihren gewünschten URLs "angehängt" werden können. Dabei können in geschweiften Klammern auch Variablen angegeben werden, wie zum Beispiel bei den Produkten die Id über

[22] `http://jackson.codehaus.org/`

/products/id. Mit einem einfachen Methoden-Aufruf lassen sich diese dann in den Ressourcen-Klassen auslesen (siehe Listing 4.30). Wichtig ist hierbei, dass Ressourcen, die die selbe Basis-URL haben (also für das gerade genannte Beispiel */products/count*), vor den Ressourcen mit Variablen in der URL deklariert werden. Würde man das nicht beachten, so würde beim Anfordern von */products/count* nicht die *ProductCountResource-*, sondern die *ProductResource*-Klasse mit dem Parameter-Wert "count" aufgerufen werden.
Um den Webservice vor unautorisierten Zugriffen zu schützen, wird vor den Router ein *Authenticator*-Objekt geschalten. Genaueres dazu im Kapitel 4.4.

```
public class ServiceApp extends Application {
  /**
   * Creates a root Restlet that will receive all incoming calls
   */
  @Override
  public Restlet createInboundRoot() {
    //register Bearer-Token Authentication
    Engine.getInstance().getRegisteredAuthenticators().add(new
        BearerHelper());
    //replace Standard JacksonConverter
    replaceConverter(JacksonConverter.class, new
        MyJacksonConverter());

    // Create a router Restlet
    Router router = new Router(getContext());
    //Secured Router for REST
    Router secureRouter = new Router(getContext());

    ChallengeScheme Bearer = new BearerHelper().getChallengeScheme
        ();
    OAuthenticator guard = new OAuthenticator(getContext(), Bearer
        ,
        "System Authentication - Provide your credentials");
    guard.setVerifier(new OAuthLocalVerifier());
    //Set the Guard before the REST-Router
    guard.setNext(secureRouter);

    secureRouter.attach("/images", ImageResource.class);
    secureRouter.attach("/images/{id}", ImageResource.class);
    secureRouter.attach("/orders", OrderResource.class);
    // Reihenfolge wichtig: /count vor /{id}
    secureRouter.attach("/orders/count", OrderCountResource.class)
        ;
    secureRouter.attach("/orders/{id}", OrderResource.class);
    secureRouter.attach("/products", ProductResource.class);
    secureRouter.attach("/products/count", ProductCountResource.
        class);
    secureRouter.attach("/products/{id}", ProductResource.class);
    secureRouter.attach("/products/{product}/variants",
```

```
        VariantResource.class);
    secureRouter.attach("/products/{product}/variants/count",
        VariantCountResource.class);
    secureRouter.attach("/products/{product}/variants/{id}",
        VariantResource.class);

    secureRouter.attach("/collections", CollectionResource.class);
    secureRouter.attach("/collections/count",
        CollectionCountResource.class);
    secureRouter.attach("/collections/{id}", CollectionResource.
        class);
    secureRouter.attach("/collections/{id}/products",
        ColProductsResource.class);

    //The guard will receive all calls to this Application
    router.attachDefault(guard);

    return router;
  }

  /**
   * Registers a new converter with the Restlet engine, after
       removing the first registered converter of the given
   * class.
   */
  static void replaceConverter(Class<? extends ConverterHelper>
      converterClass, ConverterHelper newConverter) {

    List<ConverterHelper> converters = Engine.getInstance().
        getRegisteredConverters();
    for (ConverterHelper converter : converters) {
      if (converter.getClass().equals(converterClass)) {
        converters.remove(converter);
        break;
      }
    }

    converters.add(newConverter);
  }
```

Listing 4.29: Die *Application*-Klasse in ct.Box

Nachdem das Interface des Webservice erstellt wurde, müssen nun noch die Ressourcen angelegt werden. Dafür ist die Klasse *ServerResource* zuständig, von der alle Ressourcen auf dem Server erben. In Listing 4.30 ist dargestellt, wie eine solche Klasse aufgebaut ist.

```
public class ProductResource extends ServerResource {

  @Post
  public Representation store(Product product) {
    ProductService service = MyThreadLocal.getServiceFactory().
        productService();
    if (null == getRequestAttributes().get("id")) {
      Set<ConstraintViolation<Product>> violations = service.
          validate(product);
      if (violations.size() == 0) {
        product = service.create(product);
        MyJacksonRepresentation<Product> response = new
            MyJacksonRepresentation<Product>(product);
        setStatus(Status.SUCCESS_CREATED);
        return response;
      } else {
        setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
        JSONObject response = new JSONObject();
        try {
          response.put("error", "Validation of this Product failed
              ");
          for (ConstraintViolation<Product> cv : violations) {
            response.append("Violations", cv.getPropertyPath().
                toString() + " " + cv.getMessage());
          }
          JsonRepresentation res = new JsonRepresentation(response
              );
          return res;
        } catch (JSONException e) {
          setStatus(Status.SERVER_ERROR_INTERNAL, "Error
              generating the Json-Error response.");
          return null;
        }
      }
    } else {
      setStatus(Status.CLIENT_ERROR_BAD_REQUEST, "You must not
          provide an id when creating a new Product.");
      return null;
    }
  }

  @Put
  public Representation update(Product update) {
  //...
  }

  @Delete
  public void deleteProduct() {
  //...
```

```
    }

    @Get
    public Representation get() {
    //...
    }
```

Listing 4.30: Die Ressource der Produkte

Wie man erkennen kann, werden die Methoden hier ebenfalls über Annotationen auf die dazugehörigen HTTP-Methoden abgebildet. Auf den Parameter der URL wird über *getRequestAttributes().get("id")* zugegriffen. Ebenfalls zu erkennen ist, dass, wie in Abschnitt 4.2.2 erwähnt, die Validierung vor dem Aufruf der Servicemethoden vorgenommen wird. Interessant ist zudem der Rückgabewert der Methoden, die *Representation*-Klasse. Wie der Name schon andeutet entspricht diese Klasse dem Konzept der Repräsentationen in REST. Die *Representation*-Klasse ist die Oberklasse aller in Restlet verfügbaren Repräsentationen. Je nach gewünschtem Rückgabewert kann in den Methoden der Ressource dann die entsprechende Unterklasse zurückgegeben werden. Soll der Client also ein JSON-Objekt empfangen, so wird die *JsonRepresentation* verwendet. Um das Produkt-Entity automatisch zu einem Json-Objekt zu konvertieren und dieses zurückzugeben, wird die *MyJacksonRepresentation* benutzt. Diese Klasse erweitert die *JacksonRepresentation*-Klasse aus der bereits erwähnten Jackson Erweiterung von Restlet, um einige kleinere Einstellungen wie das Datumsformat bei der Konvertierung festzulegen. Dafür muss ebenfalls der Standard-Converter dieser Erweiterung ausgetauscht werden, so dass anstatt der *JacksonRepresentation* die *MyJacksonRepresentation*-Klasse verwendet wird. Das geschieht in der *ServiceApp* (Listing 4.29) in der Methode *replaceConverter()*. Der *MyJacksonConverter* ist dabei einfach zu implementieren, wie in Listing 4.31 gezeigt.

```
public class MyJacksonConverter extends JacksonConverter {
  @Override
  protected <T> JacksonRepresentation<T> create(MediaType
      mediaType, T source) {
    return new MyJacksonRepresentation<T>(mediaType, source);
  }
  @Override
  protected <T> JacksonRepresentation<T> create(Representation
      source, Class<T> objectClass) {
    return new MyJacksonRepresentation<T>(source, objectClass);
  }
}
```

Listing 4.31: Die *MyJacksonConverter* Klasse

Die automatische Serialisierung von Jackson hat aber auch einen Nachteil: Es werden standardmäßig alle Felder eines Entities bei der Konvertierung berücksichtigt. Da aber zum Beispiel die Produkte Felder enthalten, die der Client nicht sehen soll, muss Jackson mitgeteilt werden, dass bestimmte Felder bei der Serialisierung ausgelassen werden sollen. Das kann mit der *@JsonIgnoreProperties*-Annotation auf einem Entitiy erreicht werden. Dieser Annotation kann eine Liste der Namen der Felder übergeben werden, die nicht mit in die Serlialisierung miteinbezogen werden sollen. Für das Produkt stellt sich das wie folgt dar.

```
@Entity
@Cached
@Unindexed
@JsonIgnoreProperties({ "variant_keys", "version", "
    collection_keys", "required_fields", "image_keys" })
public class Product extends DataStoreObject {
  //...
}
```

Hauptsächlich die Datastore-Schlüssel werden also vor den Clients verborgen. Für diese reicht es aus, den Wert der Id-Felder der Entities zu kennen.
Restlet ermöglicht es auch, reine Entity-Objekte anstatt derer Repräsentationen als Rückgabewert der Servicemethoden anzugeben, indem es automatisch die beste verfügbare Repräsentation für ein Entity verwendet. Die Signatur der Methode *store()* würde dann zum Beispiel folgendermaßen aussehen:

```
@Post
public Product store(Product product) {
//...
}
```

Hier würde automatisch die *JacksonRepresentation* verwendet werden, um das Produkt in serialisierter Form in der Antwort anzugeben. In diesem Fall wäre es aber nicht möglich, im Fehlerfall ein JSON-Objekt, das die Fehlerbeschreibungen enthält, zurückzugeben.
Der letzte Schritt, um Restlet in ct.Box zu integrieren, ist ein Eintrag in der web.xml Datei. Dort muss ähnlich wie bei Jersey ein spezielles Servlet (`org.restlet.ext.servlet.ServerServlet`) definiert werden, das die eingehenden Servlet-Requests koordiniert und in Restlet-konforme Requests umwandelt. Diesem Servlet wird außerdem als Parameter die *ServiceApp*-Klasse übergeben, damit es die Anfragen an die richtigen Ressourcen weiterleiten kann. (Listing 4.32)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN
        "
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
<!-- RestletServlet -->
  <servlet>
    <servlet-name>RestletServlet</servlet-name>
    <servlet-class>org.restlet.ext.servlet.ServerServlet</servlet-
        class>
    <init-param>
      <param-name>org.restlet.application</param-name>
      <param-value>de.commercetools.server.rest.ServiceApp</param-
          value>
    </init-param>
  </servlet>

  <!-- Catch all rest requests -->
  <servlet-mapping>
    <servlet-name>RestletServlet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Listing 4.32: Installation von Restlet

Der Webservice ist also unter der URL /rest erreichbar. Um über den Webservice beispielsweise ein Produkt abzurufen, muss folglich die URL `/rest/products/<product_id>` abgefragt werden.

Erkenntnisse

Die Implementierung des Webservice verlief ohne größere Schwierigkeiten. Der Einstieg in Restlet gestaltet sich dank der Tutorials in der Dokumentation sehr einfach. Ein erster Webservice, der rudimentär die Entities zurückgeben kann ist schnell erstellt. Sobald es aber an komplexere Themen wie zum Beispiel die Konfiguration des von der Jackson-Erweiterung eingesetzten Mappers, oder das Setzen und Auslesen von benutzerdefinierten HTTP-Headern geht, erreicht die Dokumentation schnell ihre Grenzen. Hier hat sich die Restlet Mailing List [23] als große Hilfe erwiesen. Dort konnte, wenn auch mit einem gewissen Suchaufwand, immer eine Lösung gefunden werden.
Als größeres Hindernis hat sich jedoch ein nicht weiter nachvollziehbarer Bug

[23] http://restlet-discuss.1400322.n2.nabble.com/

in Restlet erwiesen. Nach einem Wechsel auf eine neuere Version von Restlet haben die Webservices, die die *JsonRepresentation* als Rückgabewert verwendeten, plötzlich nichts mehr zurückgegeben. Anstatt eines Json-Objektes wurde ein leeres Dokument empfangen. Der Quellcode wurde in der Zwischenzeit aber nicht verändert. Dabei gab es keine Fehlermeldung, und auch beim Debuggen konnten keine Fehler festgestellt werden. Selbst in der Mailing List konnte der Fehler nicht nachvollzogen werden, wodurch die Vermutung auf einen Projekt-spezifischen Fehler aufkam. [Dis11] Allerdings wies ein kurzes Beispiel-Projekt, das bei einem Mitglied der Mailing List ohne Probleme lief, denselben Fehler auf. Erst mit den Dateien der älteren Restlet-Version wurden die JSON-Objekte wieder zurückgegeben. Diese Version wurde dann weiter für die Entwicklung verwendet, wodurch sich keine Einschränkungen ergaben.
Es war außerdem auf Grund eines Bugs in Jackson [JA111] nicht möglich, dynamische Filter für die JSON-Objekte zu realisieren. Jackson ermöglicht es durch eine Annotation, Entities mit Filtern zu verknüpfen, über die Felder angegeben werden können, die bei der Serialisierung des Objekts übergangen werden. [Sal11] In ct.Box wäre es wünschenswert gewesen, über URL-Query-Parameter angeben zu können, welche Felder im JSON-Objekt der Serverantwort enthalten sein sollen. So hätte man die Antworten übersichtlich halten und gleichzeitig Netzwerklast und damit Antwortzeit sparen können. Die Filterung war auch schon funktionstüchtig implementiert, allerdings wurde durch den erwähnten Bug in Jackson jedesmal eine Exception geworfen, wenn ein Entity für die Verwendung mit Filtern registriert war, aber kein Filter darauf angewendet wurde. Nachdem kein Workaround für diesen Bug bekannt ist, wurde diese Funktion für ct.Box erst einmal zurückgestellt, bis der Fehler in Jackson behoben wurde.

4.3.2 Weboberflächen

In diesem Abschnitt wird die Realisierung der Webseiten detaillierter behandelt. Wie bereits in Kapitel 3.1 erwähnt, werden diese mit dem Google Web Toolkit realisiert. In Abbildung 4.2 ist dargestellt, wie die Oberflächen dabei ungefähr aussehen sollen. Gezeigt ist die Ansicht aller registrierter Shops. Das Layout ist nach dem Vorbild von Google Mail [24] grob in drei Bereiche eingeteilt:

- Eine statische Menüleiste links zur Navigation
- Ein statischer Header mit dem commercetools Logo und Links zu den allgemeinen Einstellungen sowie einem Logout-Button

[24] `https://mail.google.com`

- Den restlichen Platz nimmt die Hauptansicht (Main Panel) ein, in die je nach Navigationspunkt andere Inhalte geladen werden

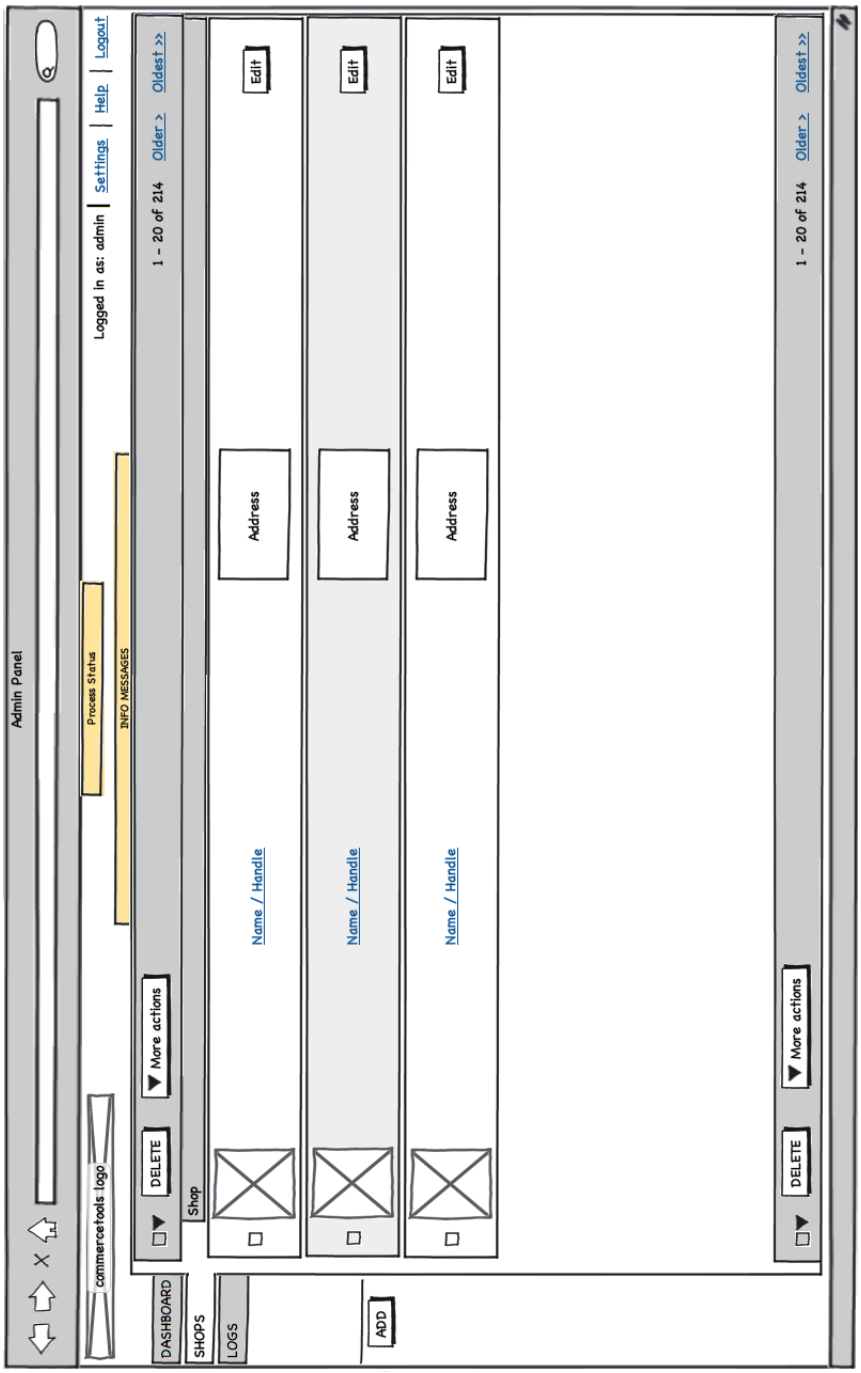


Abbildung 4.2: Mockup für das AdminPanel

GWT

GWT ist ein Toolkit zum Erstellen von AJAX-Anwendungen. Entwickler müssen dafür aber keine Kenntnis von JavaScript oder den Eigenarten der verschiedenen Browser besitzen. Um dies zu erreichen, werden GWT-Applikationen in Java-Code geschrieben, der von einem GWT-Compiler in hoch performantes JavaScript konvertiert wird, das auf allen Browsern lauffähig ist.[Goo11j] Dazu generiert der Compiler durch das sogenannte *Deferred Binding* für die unterschiedlichen Browser mehrere Versionen des Codes. Anstatt eine große JavaScript-Datei mit den Konfigurationen für alle Browser zu haben, kann so nur die wirklich benötigte Version beim Laden vom Browser verwendet werden. [Goo11c]
Dadurch ist es zusätzlich möglich, AJAX-Applikationen in der verwendeten Entwicklungsumgebung zu debuggen. Für die Entwicklung stellt GWT eine Java-API und ein Widget-Paket zur Gestaltung der Benutzeroberfläche bereit. Diese Widgets enthalten zum Beispiel Buttons, Tabellen oder Textfelder, im Prinzip jedes aus HTML bekannte Element, als Java-Objekte verpackt. Eine besondere Art der Widgets sind dabei die *Panels*, die im Grunde Behälter für andere Panels oder Widgets sind. Davon existieren in GWT mehrere Arten, zum Beispiel ein *VerticalPanel*, das die in ihm enthaltenen Widgets vertikal anordnet. Im ct.Box werden Panels als Behälter für die drei großen Bereiche des UI eingesetzt.
Eine GWT-Applikation besteht dabei aus einem oder mehreren Modulen, die ähnlich zu den Paketen in Java mehrere Funktionalitäten kapseln. In einem Modul werde alle Einstellungen vorgenommen, die eine GWT-Applikation zum Laufen braucht:[Goo11o]

Geerbte Module Äquivalent zum Import von Paketen in Java.

Entry Point Ein Einstiegspunkt ist eine Klasse, die das Interface *EntryPoint* und damit die Methode *onModuleLoad()* implementiert. Wird ein Modul geladen, so wird diese Methode aufgerufen. Sie ist somit der Startpunkt einer GWT-Applikation, ähnlich zur *main*-Methode einer Java-Applikation.

Source Path Hier wird angegeben, welche Pakete des Projekts Code enthalten, der zu JavaScript übersetzt werden kann. Dadurch lässt sich Server- und Client-seitiger Code in einem Projekt trennen.

Public Path Es können die Pakete zum Public Path hinzugefügt werden, die statische Ressourcen wie Bilder oder CSS-Dateien enthalten.

ct.Box besitzt zwei Module für die beiden Weboberflächen, die es anbietet. Listing 4.33 zeigt das Modul des AdminPanels. Für das UserPanel sieht das Modul aber im Grunde genauso aus.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='adminpanel'>
  <!-- Inherit the core Web Toolkit stuff. -->
  <inherits name='com.google.gwt.user.User' />
  <inherits name='com.google.web.bindery.requestfactory.
      RequestFactory' />

  <!-- Inherit the default GWT style sheet -->
  <inherits name='com.google.gwt.user.theme.clean.Clean' />

  <!-- Other module inherits -->
  <inherits name="com.googlecode.objectify.Objectify" />
  <inherits name="com.google.gwt.activity.Activity" />
  <inherits name="com.google.gwt.place.Place" />

  <!-- Include GWT DragAndDrop library -->
    <inherits name='com.allen_sauer.gwt.dnd.gwt-dnd'/>

    <!-- Include the GWT Visualization API -->
    <inherits name='com.google.gwt.visualization.Visualization' />

  <!-- Specify the app entry point class. -->
  <entry-point class='de.commercetools.client.AdminPanel' />

  <!-- Use AdminClientFactoryImpl by default -->
  <replace-with
    class="de.commercetools.client.gwt.AdminPanel.
        AdminClientFactoryImpl">
    <when-type-is
      class="de.commercetools.client.gwt.AdminPanel.
          AdminClientFactory" />
  </replace-with>

  <!-- Specify the paths for translatable code -->
  <source path='client' />
  <source path='shared' />
</module>
```

Listing 4.33: Das AdminPanel-Modul

Hier werden zuerst die benötigten Module, zum Beispiel die RequestFactory, importiert, bevor der Einstiegspunkt der Applikation mit der Klasse `de.commercetools.client.AdminPanel` festgelegt wird. Außerdem wird das *Deferred Binding* verwendet, um für das Interface *AdminClientFactory* dessen Implementierung *AdminClientFactoryImpl* zu verwenden. Will man nun eine Instanz der AdminClientFactory-Klasse, so muss anstatt dem Konstruktor der Implementierung die Methode `GWT.create(AdminClientFactory)` verwendet werden.

GWT-Module werden nun auf dem Server als JavaScript und den dazugehörigen Dateien (Bilder etc.) gespeichert. Um ein solches Modul anzeigen zu können, muss es in eine HTML-Seite eingebunden werden.[Goo11o] Diese Seite wird von GWT *Host Page* genannt. Die Host Page des AdminPanels ist in Listing 4.34 abgebildet.

```
<!doctype html>
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8"
    >
<title>Admin Panel</title>
<!--                                                -->
<!-- This script loads your compiled module.        -->
<!-- If you add any GWT meta tags, they must        -->
<!-- be added before this line.                     -->
<!--                                                -->
<script type="text/javascript" language="javascript"
  src="../adminpanel/adminpanel.nocache.js"></script>

<style type="text/css">
.loading_msg {
  background:#ffdd3d;
  color:#333;
  padding:5px;
  position:absolute;
  left:50%;
  display:none;
  border-radius: 0 0 5px 5px;
  box-shadow: 0 2px 5px #CCC;
}
.loader {
  position:absolute;
}
</style>
</head>

<!-- The body can have arbitrary html, or          -->
<!-- you can leave the body empty if you want      -->
<!-- to create a completely dynamic UI.            -->
<!--                                                -->
<body>
<div id="app_loader" class="loader"><img src="/static/media/ajax-
    loader.gif" /></div>
<div id="loading" class="loading_msg"></div>
  <!-- OPTIONAL: include this if you want history support -->
  <iframe src="javascript:''" id="__gwt_historyFrame" tabIndex
      ='-1'
    style="position: absolute; width: 0; height: 0; border: 0"></
        iframe>
```

```
    <!-- RECOMMENDED if your web app will not function without
        JavaScript enabled -->
    <noscript>
      <div style="width: 22em; position: absolute; left: 50%; margin
          -left: -11em; color: red; background-color: white; border:
          1px solid red; padding: 4px; font-family: sans-serif">
        Your web browser must have JavaScript enabled in order for
            this
        application to display correctly.
      </div>
    </noscript>
  </body>
  </html>
```

Listing 4.34: Die Host Page des AdminPanels

Der Java-Script Code des AdminPanels wird hierbei über das `<script>`-Tag eingebunden, in dem der Pfad der Datei im war-Verzeichnis der WebApp angegeben werden muss. Im Body der HTML-Datei kann nun beliebiger HTML-Code stehen. In diesem Fall wurde ein `<noscript>`-Element eingefügt, um eine Fehlermeldung anzuzeigen, falls der Browser kein JavaScript unterstützt oder es deaktiviert hat. Außerdem sind zwei `<div>`-Umgebungen zum Anzeigen einer Lade-Grafik zusammen mit dem dazugehörigen CSS-Style vorhanden.
Das `<iframe>`-Tag muss in dieser Form in der Host Page vorhanden sein, wenn die Browser-Historie von der GWT-Applikation unterstützt werden soll. Dieses Problem tritt nur bei AJAX-Applikationen auf, da diese nicht auf einer Navigation zwischen mehreren statischen HTML-Seiten basieren, sondern meistens in einer einzigen HTML-Seite laufen. Um trotzdem das von den Nutzern erwartete Verhalten im Bezug auf die Historie zu erreichen, müssen die Applikationen dort selbst nachhelfen.

Activities & Places

Seit Version 2.1 wird ist mit den *Activities & Places* (A&P) ein eigenes Framework zum Browser-History Management in GWT integriert. Mit den Activities und Places können URLs erstellt werden, die ohne Probleme zu den Lesezeichen eines Browsers hinzugefügt werden können, wodurch auch die Browser-Historie wie von den Nutzern erwartet funktioniert. Dazu werden *History Tokens* verwendet, Strings die an die URL angehängt werden, um den aktuellen Zustand der Applikation wiederzuspiegeln. [Goo11k] Activites und Places können aber zusätzlich noch dazu verwendet werde, das Model-View-

Presenter-Pattern[25] (MVP) in einer GWT-Applikation umzusetzen. Beim MVP-Pattern werden Ansicht (*View*) und Businesslogik (*Model*) strikt voneinander getrennt. Die Kommunikation zwischen diesen beiden Komponenten erfolgt über einen Präsentator (*Presenter*), der die Ansicht steuert und mit Daten aus dem Modell füllt.

Views Zuerst sollten aber die *Views* implementiert werden. Nach dem MVP-Prinzip besitzt ein View ein Interface, das je nach Art des Clients verschiedene Implementierungen besitzen kann. Die Views werden dadurch möglichst austauschbar gehalten. GWT stellt für Views das Interface *isWidget* bereit, das fast alle Widgets, sowie aus Widgets zusammengesetzte Views (sogenannte *Composites*) implementieren.
Um die Kommunikation zwischen View und Presenter zu ermöglichen, sollte im View-Interface außerdem eine Presenter-Schnittstelle definiert werden, die der dazugehörige Presenter implementiert. Am Beispiel des Menüs im Admin-Panel, zeigt Listing 4.35 das Interface des dazugehörigen Views.

```
public interface AdminMenu extends IsWidget {

  public interface MenuPresenter extends Presenter {
    void onDashboardClicked();
    void onShopsClicked();
  }
  void setPresenter(MenuPresenter presenter);
}
```

Listing 4.35: Das Interface des *AdminDahsboard*

Die Menü-Ansicht besitzt demnach ein eigenes Presenter-Interface, das ein Standard-Interface für alle Presenter erweitert. In diesem Standard-Interface ist nur eine Methode deklariert, mit der der aktuelle Place geändert wird.

```
public abstract interface Presenter {
  public abstract void go(final Place place);
}
```

Listing 4.36: Das Presenter Interface

Die tatsächliche Implementierung der View-Interfaces wurde in ct.Box mit Hilfe des *UiBinder*-Frameworks umgesetzt. Bis Version 2.0 konnte das User-Interface in GWT nur in reinem Java-Code geschrieben werden. Dafür müssen alle Eigenschaften der Widgets über die Methoden ihrer Java-Klassen festgelegt werden, wodurch der Code leicht unübersichtlich werden kann. Außerdem sind Layout und Verhalten bei dieser Variante nur schlecht getrennt. Die

[25] http://de.wikipedia.org/wiki/Model_View_Presenter

Einführung von UIBinder in Version 2.0 adressierte diese Probleme.
Seitdem kann das Layout des UI in speziellen XML-Dateien festgelegt werden. Dabei besteht große Ähnlichkeit zur Strukturierung einer Webseite über HTML, weswegen dieser Weg vor allem für Webentwickler mit Erfahrung in HTML und CSS interessant ist. In UIBinder besteht eine strikte Trennung zwischen Layout und Verhalten des UI, wodurch die Zusammenarbeit zwischen Designer und Programmieren erleichtert wird. Vor allem wird aber der Code übersichtlich und verständlich gehalten, weswegen UIBinder in ct.Box eingesetzt wird. [Goo11f] Wie bereits gesagt wird bei UIBinder Layout und Logik getrennt. Man hat also eine XML-Datei mit der Endung ".ui.xml" und eine dazugehörige Java-Klasse. In der XML-Datei können nun HTML-Code und GWT-Widgets beliebig verschachtelt eingesetzt werden, wobei HTML-Code entweder in einem *HTMLPanel* oder innerhalb eines *HTMLWidgets* stehen muss. [Goo11f] In Listing 4.37 ist die UIBinder XML-Datei für das Menü des AdminPanels zu sehen.

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.
    ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
  xmlns:g="urn:import:com.google.gwt.user.client.ui">

  <ui:style src="../../../Resources/ct_structure.css"></ui:style>
  <g:HTMLPanel styleName="{style.menu}">
    <ul>
      <li>
        <g:Label ui:field="dashboard" addStyleNames="{style.menu-
            link}">DASHBOARD</g:Label>
      </li>
      <li>
        <g:Label ui:field="shops" addStyleNames="{style.menu-link}
            ">SHOPS</g:Label>
      </li>
      <li>
        <g:Label ui:field="logs" addStyleNames="{style.menu-link}"
            >LOGS</g:Label>
      </li>
    </ul>
  </g:HTMLPanel>
</ui:UiBinder>
```

Listing 4.37: UIBinder für das AdminPanel Menü

Um Widgets in einer UIBinder-Datei verwenden zu können, müssen deren Pakete an einen XML-Namespace gebunden werden. In Listing 4.37 werden zum Beispiel die Widgets des Pakets *com.google.gwt.user.client.ui* an das Präfix *g* gebunden. Um ein solches Widget zu verwenden muss also ein XML-Element

mit dem Präfix *g* und dem Namen des Widgets verwendet werden, zum Beispiel `<g:Label>`. Dieses Element enthält außerdem das Attribut *addStyleNames*. Dieses Attribut hat die selbe Wirkung wie ein Aufruf der Methode *addStyleNames(int)* auf eine Label-Objekt im Java-Code. Tatsächlich wird dieses Attribut sogar in einen Methodenaufruf umgewandelt. [Goo11f]
Über das `<ui:style>`-Element können CSS-Regeln für das UI festgelegt werden. Durch das *class*-Attribute werden diese dann auf die einzelnen Elemente angewandt.
Zu dieser XML-Datei gehört nun eine Java-Klasse, die den selben Namen besitzen muss. In dieser Klasse hat man als Entwickler Zugriff auf die im UIBinder-Template definierten Elemente. Listing 4.38 zeigt eine solche Klasse am Beispiel der *AdminMenuImpl*-Klasse.

```
public class AdminMenuImpl extends Composite implements AdminMenu
    {
  interface AdminMenuImplUiBinder extends UiBinder<Widget,
      AdminMenuImpl> {
  }
  private static AdminMenuImplUiBinder uiBinder = GWT.create(
      AdminMenuImplUiBinder.class);

  public AdminMenuImpl() {
    initWidget(uiBinder.createAndBindUi(this));
  }
  private MenuPresenter presenter;

  @UiField
  Label shops;
  @UiField
  Label dashboard;

  @UiHandler("shops")
  void onShopsClick(ClickEvent e) {
    presenter.onShopsClicked();
  }
  @UiHandler("dashboard")
  void onDashClick(ClickEvent e){
    presenter.onDashboardClicked();
  }

  @Override
  public void setPresenter(MenuPresenter presenter) {
    this.presenter = presenter;
  }
}
```

Listing 4.38: Die *AdminMenuImpl*-Klasse

Durch die *@UiField*-Annotation lassen sich die Felder der Java-Klasse mit den gleichnamigen Elementen der XML-Datei verbinden. Dazu generieren die automatisch erzeugten *UiBinder*-Factory-Klassen aus den XML-Strukturen das eigentliche UI und füllen die Felder der Java-Klassen mit den generierten Objekten. [Goo11f] Dies geschieht beim Aufruf der Methode `uiBinder.createAndBindUi(this)`. Hier ist auch zu erkennen, wie das Handling von Events mit UiBinder funktioniert. Mit der *@UiHandler*-Annotation können Methoden angegeben werden, die für verschiedene Arten von Events (z.B. *ClickEvent*) aufgerufen werden.

Presenter Listing 4.38 zeigt ebenfalls, dass die komplette Verwaltung der Views den Presentern überlassen wird. Alle Aktionen der User werden durch Methoden in den Presenter-Klassen abgebildet. Diese sind dann auch dafür zuständig, entsprechende Änderungen an den Views durchzuführen.
In A&P entsprechen die Activities der Rolle des Presenter. Eine Activity repräsentiert dabei eine beliebige Handlung eines Users. Activities enthalten allerdings keinen Code zum Aufbau des User-Interface, sondern sind für Aufgaben wie die Initialisierung und das Laden des UI verantwortlich. Sie kommunizieren über die Serviceschicht mit dem Datenmodell von ct.Box und geben die erhaltenen Informationen an die Views weiter. Activities werden über einen *ActivityManager* gestartet und gestoppt. Dieser ActivityManager ist mit einem Behälter-Widget verbunden, in das die Activities ihre Views laden können. [Goo11k] Eine Activity implementiert nun entweder das Interface `com.google.gwt.activity.shared.Activity` oder erbt von der *AbstractActivity*-Klasse, die schon Default-Implementierungen für alle benötigten Methoden besitzt. Im folgenden Listing 4.39 ist die zum Menü gehörige Activity vorgestellt. In der *start()*-Methode wird der Activity der Behälter angegeben, in den es seinen View zu laden hat. Das geschieht über die Methode *setWidget()*. Das erklärt auch, warum die Views das Interface *isWidget* implementieren sollten, da sie sonst nicht in ihre Container geladen werden könnten.

```
public class AMenuActivity extends AbstractActivity implements
    MenuPresenter {
  // Used to obtain views, eventBus, placeController
  private AdminClientFactory clientFactory;

  public AMenuActivity(AMenuPlace place, AdminClientFactory
      clientFactory) {
    this.clientFactory = clientFactory;
  }

  @Override
```

```
    public void go(Place place) {
      clientFactory.getPlaceController().goTo(place);
    }

    @Override
    public void start(AcceptsOneWidget panel, EventBus eventBus) {
      AdminMenu menu = clientFactory.getAdminMenu();
      menu.setPresenter(this);
      panel.setWidget(menu);
    }

    @Override
    public void onDashboardClicked() {
      go(new ADashboardPlace(""));
    }

    @Override
    public void onShopsClicked() {
      go(new AShopListPlace(""));
    }
  }
```

Listing 4.39: Die Activity für das Menü des AdminPanels

Da das Menü für die Navigation im AdminPanel verantwortlich ist, muss also bei einem Klick auf einen Menüpunkt die Hauptansicht wechseln. Das geschieht durch die Änderung des aktuellen Place. Ein Place ist ein Java-Objekt, das den Zustand des User-Interface zu einem gewissen Zeitpunkt darstellt. Das bedeutet, dass Places in History Tokens umgewandelt werden können müssen, und umgekehrt. Dafür wird in jeder Place-Klasse, die ihren Zustand in der URL speichern soll, ein *PlaceTokenizer* angegeben, der die Konvertierung von Zustand zu Token und zurück vornehmen kann.[Goo11k] Listing 4.40 zeigt den Place für die Ansicht zum Editieren einzelner Shops. Andere Activities verwenden die in ihrem Presenter-Interface definierten Methoden, um über die RequestFactory Aufrufe an die Services zu schicken und ihre Views entsprechend der erhaltenen Antwort zu aktualisieren.

```
  public class AEditShopPlace extends Place {
    private String name;

    public AEditShopPlace(String name) {
      this.name = name;
    }
    public String getName() {
      return this.name;
    }

    @Prefix(value = "Shop")
```

```
    public static class Tokenizer implements PlaceTokenizer<
        AEditShopPlace> {
      @Override
      public AEditShopPlace getPlace(String arg0) {
        return new AEditShopPlace(arg0);
      }
      @Override
      public String getToken(AEditShopPlace arg0) {
        return arg0.getName();
      }
    }
  }
```

Listing 4.40: Eine Place-Klasse

Um den richtigen Shop aus dem Datastore laden zu können, muss dessen Name als Zustand in der URL gespeichert werden. Das geschieht normalerweise durch das Anhängen des Klassennamens des Places, gefolgt von einem Doppelpunkt und dem vom PlaceTokenizer zurückgegebenen Token, an die URL. Über die *@Prefix*-Annotation auf dem PlaceTokenizer lässt sich aber auch ein anderer Wert anstatt des Klassennamens festlegen.
Die *PlaceHistoryHandler*-Klasse ist schließlich dafür verantwortlich, dass die URL an den aktuellen Place angepasst wird. Dafür verwendet sie eine *PlaceHistoryMapper*-Klasse, in der alle Places, die dem PlaceHistoryHandler bekannt sein sollen, registriert werden. Das ist in Listing 4.41 dargestellt.

```
/**
 * PlaceHistoryMapper interface is used to attach all places which
     the PlaceHistoryHandler should be aware of. This is
 * done via the @WithTokenizers annotation or by extending
    PlaceHistoryMapperWithFactory and creating a separate
 * TokenizerFactory.
 */
@WithTokenizers({ ADashboardPlace.Tokenizer.class, AShopListPlace.
    Tokenizer.class, ACreateShopPlace.Tokenizer.class,
    AEditShopPlace.Tokenizer.class, AStatisticPlace.Tokenizer.
        class })
public interface AdminPlaceHistoryMapper extends
    PlaceHistoryMapper {
}
```

Listing 4.41: Der PlaceHistoryMapper

Zuständig für den Wechsel der Places in GWT ist die *goTo*-Methode der *PlaceController*-Klasse. Dieser PlaceController wird mit einem *EventBus* initialisiert, über den für jeden Wechsel zu einem neuen Place ein *PlaceChangeEvent* verschickt wird. Sowohl vom PlaceController als auch vom EventBus wird

nur eine Instanz für die gesamte Applikation benötigt, weswegen diese in einer Factory-Klasse (*AdminClientFactory*) als statische Variablen definiert werden. Von dort können sie dann über Getter-Methoden abgerufen werden. In der *AdminClientFactory* (siehe Listing 4.42) sind außerdem die Instanzen aller Views des AdminPanels gespeichert, da diese nicht jedes Mal neu aufgebaut werden müssen, sondern wiederverwendet werden können. Darüber hinaus enthält sie, wie schon in Kapitel 4.2.1 erwähnt, die RequestFactory von ct.Box. Die Initialisierung der AdminClientFactory erfolgt dabei beim Starten des AdminPanel-Moduls.

```
public class AdminClientFactoryImpl implements AdminClientFactory
    {
  private static final EventBus eventBus = new SimpleEventBus();
  private static final PlaceController placeController = new
      PlaceController(eventBus);
  private static final AdminDashboard dashboard = new
      AdminDashboardImpl();
  private static final AdminMenu menu = new AdminMenuImpl();
  private static final AdminShopList shopList = new
      AdminShopListImpl();
  private static final BoxRequestFactory requestFactory = GWT.
      create(BoxRequestFactory.class);
  private static final AdminShopAdd createShopView = new
      AdminShopAddImpl();
  private static final AdminShopEdit editShopView = new
      AdminShopEditImpl();
  private static final AdminPanelWidget adminPanel = new
      AdminPanelWidget();
  private static final AdminHeader adminHeader = new
      AdminHeaderImpl();
  private static final AdminStatistics adminStats = new
      AdminStatisticsImpl();
  static {
    requestFactory.initialize(eventBus);
  }
  @Override
  public EventBus getEventBus() {
    return eventBus;
  }
  @Override
  public PlaceController getPlaceController() {
    return placeController;
  }
  @Override
  public AdminDashboard getAdminDashboard() {
    return dashboard;
  }
  @Override
  public AdminMenu getAdminMenu() {
```

```
    return menu;
  }
  @Override
  public AdminShopList getShopList() {
    return shopList;
  }
  @Override
  public BoxRequestFactory getRequestFactory() {
    return requestFactory;

  //Getters for all Views...
}
```

Listing 4.42: Die Implementierung der AdminClientFactory

An den selben EventBus, der zur Initialisierung des PlaceControllers verwendet wurde, müssen nun noch die bereits erwähnten *ActivityManager* angeschlossen werden. Diese horchen auf die PlaceChangeEvents im EventBus und liefern die passende Activity für den angeforderten Place zurück. Dazu verwenden sie *ActivityMapper*. Ein ActivityManager verfügt außerdem über die Methode *setDisplay()*, über die der Behälter angegeben werden kann, in den der View einer Activity geladen werden soll. Die Implementierung eines ActivityMappers ist in Listing 4.43 zu sehen.

```
public class AMainActivityMapper implements ActivityMapper {
  private AdminClientFactory clientFactory;
  public AMainActivityMapper(AdminClientFactory clientFactory) {
    super();
    this.clientFactory = clientFactory;
  }
  @Override
  public Activity getActivity(Place place) {
    if (place instanceof ADashboardPlace)
      return new ADashboardActivity((ADashboardPlace) place,
          clientFactory);
    else if (place instanceof AShopListPlace)
      return new AShopListActivity((AShopListPlace) place,
          clientFactory);
    else if (place instanceof ACreateShopPlace)
      return new ACreateShopActivity((ACreateShopPlace) place,
          clientFactory);
    [...]
    return null;
  }
}
```

Listing 4.43: Ein ActivityMapper

Ein solcher ActivityMapper ist für jeden der drei Bereiche des User-Interface zuständig. Da allerdings der Header und das Menü statische Anzeigen sind, sich also nicht für unterschiedliche Places ändern sollen, wird in der *getActivity()*-Methode ihrer ActivityMapper immer die selbe Activity zurückgegeben, wodurch sich auch die Anzeige nicht ändert.
Zusammengesetzt werden alle diese Elemente im Entry Point des Moduls (Listing 4.44, in dem auch die Views in der AdminClientFactory initialisiert werden.

```
public class AdminPanel implements EntryPoint {
  private Place defaultPlace = new ADashboardPlace("");

  @Override
  public void onModuleLoad() {
    // Create ClientFactory using deferred binding so we can
        replace it with different implementations in gwt.xml
    AdminClientFactory clientFactory = GWT.create(
        AdminClientFactory.class);
    EventBus eventBus = clientFactory.getEventBus();

    PlaceController placeController = clientFactory.
        getPlaceController();
    AdminPanelWidget appWidget = clientFactory.getAdminPanel();
    SimplePanel mainPanel = appWidget.getMainPanel();
    SimplePanel menuPanel = appWidget.getMenuPanel();
    SimplePanel headerPanel = appWidget.getHeaderPanel();

    // Start ActivityManager for the menu widget with our
        ActivityMapper
    ActivityMapper menuActivityMapper = new CachingActivityMapper(
        new AMenuActivityMapper(clientFactory));
    ActivityManager menuActivityManager = new ActivityManager(
        menuActivityMapper, eventBus);
    menuActivityManager.setDisplay(menuPanel);

    // Start ActivityManager for the header widget with our
        ActivityMapper
    ActivityMapper headerActivityMapper = new
        CachingActivityMapper(new AHeaderActivityMapper(
        clientFactory));
    ActivityManager headerActivityManager = new ActivityManager(
        headerActivityMapper, eventBus);
    headerActivityManager.setDisplay(headerPanel);

    // Start ActivityManager for the main widget with our
        ActivityMapper
    ActivityMapper mainActivityMapper = new CachingActivityMapper(
        new AMainActivityMapper(clientFactory));
```

```
        ActivityManager mainActivityManager = new ActivityManager(
            mainActivityMapper, eventBus);
        mainActivityManager.setDisplay(mainPanel);

        AdminPlaceHistoryMapper historyMapper = GWT.create(
            AdminPlaceHistoryMapper.class);
        PlaceHistoryHandler historyHandler = new PlaceHistoryHandler(
            historyMapper);

        historyHandler.register(placeController, eventBus,
            defaultPlace);
        RootPanel.get().clear();
        RootPanel.get().add(appWidget);

        historyHandler.handleCurrentHistory();
    }
}
```

Listing 4.44: Der Entry Point des AdminPanel-Moduls

Hier ist noch einmal schön die Funktionsweise der Activities und Places zu erkennen. Die drei Anzeigeregionen (*mainPanel*, *menuPanel* und *headerPanel*) sind in einem View (*AdminPanelWidget*) definiert und werden dort richtig angeordnet. Jede dieser Regionen erhält einen eigenen ActivityManager, der sich über die *setDisplay()*-Methode mit ihnen verbindet. Ein ActivityManager horcht nun auf *PlaceChangeEvents* im *EventBus* und tauscht die aktive Activity aus, falls ein neuer Place angefordert wird. Dazu wird der ActivityMapper benötigt, der die Activity zu dem angeforderten Place zuordnen kann. Diese Activity kann dann ihren View in das dazugehörige Panel laden. Währenddessen kümmert sich der *PlaceHistoryHandler* darum, die richtigen History Tokens in der URL anzuzeigen, und verwendet dazu den bereits erwähnten *AdminPlaceHistoryMapper*.
Das *RootPanel* stellt die oberste Ebene in der View-Hierarchie dar. Es wird niemals selbst erstellt, sondern immer über die *get()*-Methode erhalten. Alle selbst erstellten Views müssen dann dem RootPanel als Kindelemente hinzugefügt werden.
Um lschließlich die Anzeige zu starten wird die Methode *handleCurrentHistory()* aufgerufen, die den aktuellen Token in der URL verarbeitet. Dadurch ist es möglich, das Modul über die Lesezeichen des Browsers zu starten.

4.3.3 Erkenntnisse

Die Weboberflächen zu implementieren hat sich zwar nicht als schwierig, jedoch als ziemlich langwierig herausgestellt. Die Einarbeitung in die MVP-Pro-

grammierung und vor allem darin, wie diese in den Activities und Places umgesetzt wird, hat dabei die längste Zeit in Anspruch genommen. In der GWT-Dokumentation zu den Activities und Places wird noch dazu auf zwei Dokumente zum Verständnis der MVP-Entwicklung verwiesen [26] [27], die allerdings nicht auf die Activities und Places eingehen und damit eher zur Verwirrung als zu deren Verständnis beitragen.
Hat man das Konzept und die Art und Weise, auf die die Komponenten der A&P miteinander zusammenarbeiten, aber einmal verstanden, gestaltet sich die Entwicklung sehr komfortabel und fast schon zur Routinearbeit. Es müssen ein Place und eine Activity für ein View erstellt werden, der Tokenizer des Places im PlaceHistoryMapper registriert werden, und die Activity im entsprechenden ActivityMapper für den gerade erstellten Place zurückgegeben werden.
Wie man an diesem Kapitel erkennen kann, ist dafür aber relativ viel Code nötig, und es muss für jeden einzelnen View durchgeführt werden, was einiges an Zeit kostet.
Dazu kommen die in Kapitel 4.2.2 erwähnten Probleme mit der RequestFactory, da diese ja in den Activities zum Aufruf der Services verwendet wird.
Allerdings war es sehr angenehm, die Benutzeroberfläche hauptsächlich in Java entwickeln zu können, wodurch wiederum viel Zeit für die Einarbeitung in JavaScript und damit verbundene Technologien gespart werden konnte.
Zusammenfassend lässt sich sagen, dass die Implementierung der Weboberflächen am meisten Zeit in Anspruch genommen hat, besonders da hier die Probleme der RequestFactory und der Aufwand der A&P-Entwicklung zusammenkamen.

4.4 Sicherheit und Authentifizierung

In diesem Abschnitt werden die Maßnahmen beschrieben, die zum Schutz der Webseiten und des Webservice vor unautorisiertem Zugriff getroffen wurden.

4.4.1 Schutz der Weboberflächen

Um die Weboberflächen zu schützen, wird in ct.Box eine Benutzersystem verwendet. Dabei existieren zwei Klassen von Benutzern. Die Benutzer von ct.Box, also die Shop-Betreiber, und die Administratoren von commercetools. Für jede

[26] `http://code.google.com/intl/de-DE/webtoolkit/articles/mvp-architecture.html`
[27] `http://code.google.com/intl/de-DE/webtoolkit/articles/mvp-architecture-2.html`

dieser beiden existieren eigene Repräsentationen im Datenmodell, nämlich die *ShopAdmin*- und die *CTAdmin*-Klasse, die allerdings identische Felder besitzen. Eben da diese beiden Klassen quasi identisch sind, kann in diesem Kapitel einfach von einem "Admin" gesprochen werden, wenn eines der beiden Admin-Objekte gemeint ist. In Listing 4.45 ist ein Ausschnitt der *ShopAdmin*-Klasse dargestellt.

```
@Entity
@Cached
public class ShopAdmin implements Serializable {
  @Id
  private String user_name;

  private String first_name;
  private String last_name;
  private String email;
  private String password;
  private Date updated_at;
  private Date created_at;
}
```

Listing 4.45: Die Felder eines *ShopAdmin*

Genau wie ein *CTAdmin*, besitzt ein *ShopAdmin* einen eindeutigen Benutzernamen, durch den er im Datastore identifiziert werden kann. Dazu sind noch der volle Name, E-Mail und Password eines Administrators gespeichert. Da Die Passwörter werden selbstverständlich nicht im Klartext im Datastore hinterlegt, sondern davor durch die Bibliothek *jBCrypt*[28] verschlüsselt. jBCrypt ist eine Java-Implementierung des auch im Betriebssystem OpenBSD[29] verwendeten Blowfish-Algorithmus[30]. [Mil10] Zusätzlich zum Hashing des Passworts wird in jBCrypt außerdem Salting[31] verwendet, um Angriffe durch Wörterbücher, wie zum Beispiel *Rainbow Tables* zu erschweren.
Der Login der Administratoren erfolgt über eine Webseite, die vor die beiden GWT-Module geschaltet wird. Diese Webseite besteht im Grunde nur aus einer HTML-Form mit zwei Eingabefeldern für Benutzername und Passwort. Die Werte dieser Felder werden dann per HTTP-POST an ein Java-Servlet gesendet, das diese auf ihre Gültigkeit überprüft. Um zu erkennen, ob sich ein ShopAdmin oder ein CTAdmin einloggen möchte, wird wie beim Setzen des Namespace (Kapitel 4.1.3) die Subdomain des HTTP-Requests überprüft. Stimmen die gesendeten Werte mit den Feldern eines in der Datenbank gespeicher-

[28] http://www.mindrot.org/projects/jBCrypt/
[29] http://www.openbsd.org/de/
[30] http://de.wikipedia.org/wiki/Blowfish
[31] http://de.wikipedia.org/wiki/Salt_(Kryptologie)

ten Admins überein, wird das Admin-Objekt (*ShopAdmin/CTAdmin*) in einer Session-Varaiblen gespeichert und ein HTTP-Redirect zur passenden Benutzeroberfläche durchgeführt. Dies geschieht durch einen Aufruf an den LoginService in der ServiceLayer von ct.Box. In Listing 4.46 ist das verwendete Java-Servlet zu sehen.

```
public class LoginServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
      HttpServletResponse response) throws NotFoundException,
      IOException,
       ServletException {

    String username = "";
    String password = "";

    if (request.getParameter("username") != null)
      username = request.getParameter("username").trim();
    if (request.getParameter("password") != null)
      password = request.getParameter("password").trim();

    request.getSession().removeAttribute("error");

    //Get the subdomain
    String serverName = request.getHeader("Host");
    String[] splitted = serverName.split("\\.");
    String subdomain = splitted[0];

    if (subdomain.equals(SystemProperty.applicationId.get())) {
      /* login to AdminPanel */
      APLoginService service = MyThreadLocal.getServiceFactory().
          apLoginService();
      try {
        if (username.equals("")) {
          request.getSession().setAttribute("error", "The given
              username or password are not correct.");
          response.sendRedirect("/login.jsp");
        } else {
          CTAdmin admin = service.loginServer(username, password,
              request.getSession());
          if (admin == null) {
            request.getSession().setAttribute("error", "The given
                username or password are not correct.");
            response.sendRedirect("/login.jsp");
          } else {
            if (SystemProperty.environment.value() ==
                SystemProperty.Environment.Value.Development) {
              String url = "http://" + serverName + "/admin/
                  adminpanel.jsp?gwt.codesvr=127.0.0.1:9997";
              response.sendRedirect(url);
            } else {
```

```
            String url = "http://" + serverName + "/admin/
                adminpanel.jsp";
            response.sendRedirect(url);
          }
        }
      }
    } catch (NotFoundException e) {
      request.getSession().setAttribute("error", "The given
          username or password are not correct.");
      response.sendRedirect("/login.jsp");
    }

  } else {
    /* login to UserPanel */
    UPLoginService service = MyThreadLocal.getServiceFactory().
        upLoginService();
    try {
      if (username.equals("")) {
        request.getSession().setAttribute("error", "The given
            username or password are not correct.");
        response.sendRedirect("/login.jsp");
      } else {
        ShopAdmin user = service.loginServer(username, password,
            request.getSession());
        if (user == null) {
          request.getSession().setAttribute("error", "The given
              username or password are not correct.");
          response.sendRedirect("/login.jsp");
        } else {
          String loc = service.getSavedLocale(subdomain);
          String locale = "";
          if (!loc.equals(""))
            locale = "&locale=" + loc;
          if (SystemProperty.environment.value() ==
              SystemProperty.Environment.Value.Development) {
            String url = "http://" + serverName + "/admin/
                userpanel.jsp?gwt.codesvr=127.0.0.1:9997"
                + locale;
            response.sendRedirect(url);
          } else {
            String url = "http://" + serverName + "/admin/
                userpanel.jsp" + locale;
            response.sendRedirect(url);
          }
        }
      }
    } catch (NotFoundException e) {
      request.getSession().setAttribute("error", "The given
          username or password are not correct.");
      response.sendRedirect("/login.jsp");
```

```
        }
      }
    }
  }
```

Listing 4.46: Das LoginServlet

Listing 4.47 zeigt die Implementierung des Logins für die *CTAdmins* in der Service-Layer. Die Implementierung für die *ShopAdmins* ist identisch, nur werden dort natürlich anstatt der *CTAdmin*-Objekte, *ShopAdmin*-Objekte in den Sessions gespeichert und von dort geladen.

```
public class APLoginServiceImpl implements APLoginService {
  ObjectifyDao<CTAdmin> dao = new ObjectifyDao<CTAdmin>(CTAdmin.
      class);

  @Override
  public CTAdmin loginServer(String userName, String password,
      HttpSession session) throws NotFoundException {
    CTAdmin admin = dao.get(userName);
    boolean validPw = BCrypt.checkpw(password, admin.getPassword()
        );
    if (admin != null && validPw) {
      storeUserInSession(admin, session);
      return admin;
    }
    return null;
  }

  @Override
  public CTAdmin loginFromSessionServer() {
    return getUserAlreadyFromSession();
  }

  @Override
  public void logout() {
    deleteUserFromSession();
  }

  private void storeUserInSession(CTAdmin admin, HttpSession
      session) {
    session.setAttribute("admin", admin);
    session.setMaxInactiveInterval(1800);
  }

  private void deleteUserFromSession() {
    HttpSession session = RequestFactoryServlet.
        getThreadLocalRequest().getSession();
    session.removeAttribute("admin");
    session.invalidate();
```

```
    }

    private CTAdmin getUserAlreadyFromSession() {
      CTAdmin admin = null;
      HttpSession session = RequestFactoryServlet.
          getThreadLocalRequest().getSession();
      Object userObj = session.getAttribute("admin");
      if (userObj != null && (userObj instanceof CTAdmin)) {
        admin = (CTAdmin) userObj;
      }
      return admin;
    }

    @Override
    public CTAdmin loginFromSessionWithRequest(HttpServletRequest
        request) {
      CTAdmin admin = null;
      HttpSession session = request.getSession();
      Object userObj = session.getAttribute("admin");
      if (userObj != null && (userObj instanceof CTAdmin)) {
        admin = (CTAdmin) userObj;
      }
      return admin;
    }
  }
```

Listing 4.47: Die Felder eines *ShopAdmin*

Wie man sieht, ist ein Session-Objekt 30 Minuten gültig. Nach 30-minütiger Inaktivität muss sich ein User also neu einloggen. Hier existieren zwei Methoden, um einen eventuell eingeloggten Admin aus der Session zu holen. In der *loginFromSessionServer()*-Methode wird die Session des *RequestFactoryServlets* benutzt. Diese Methode wird nur von den GWT-Oberflächen dazu verwendet, den Benutzernamen des aktuell eingeloggten Admins auf den Webseiten anzuzeigen. Deswegen muss hier das *RequestFactoryServlet* verwendet werden, da über dieses die Requests mit ihren HttpSessions verarbeitet werden.
Die *loginFromSessionWithRequest*-Methode nimmt als Parameter ein *HttpServletRequest*, dessen Session zum Speichern des Admins eingesetzt wird. Mit dieser Methode überprüft ein Servlet-Filter jeden Aufruf (*HttpServletRequest*) an URLs, unter der die beiden Oberflächen liegen (/admin/*). Findet der Filter ein Objekt in der Session, so darf der Aufruf fortfahren, andernfalls wird man zur Login-Seite weitergeleitet. Um Sessions auf der App-Engine überhaupt zu ermöglichen, muss in der *appengine-web.xml*-Datei eines GAE-Projekts folgende Einstellung vorgenommen werden:

```
<sessions-enabled>true</sessions-enabled>
```

Die Session-Verwaltung der App-Engine legt die Session-Objekte dadurch im Datastore ab, und verwendet sogar den Memcache. Dafür müssen alle Objekte, die in einer Session gehalten werden sollen, serialisierbar sein, also das Interface *java.io.Serializable* implementieren. [Goo11m]
Listing 4.48 veranschaulicht schließlich noch den Aufbau des verwendeten ServletFilters.

```
public class LoginFilter implements Filter {
  private static final Logger log = Logger.getLogger(LoginFilter.
      class.getName());

  @Override
  public void doFilter(ServletRequest req, ServletResponse res,
      FilterChain chain) throws IOException,
       ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;

    String serverName = request.getHeader("Host");
    String[] splitted = serverName.split("\\.");
    String subdomain = splitted[0];
    if (subdomain.equals(SystemProperty.applicationId.get())) {
      if (adminIsLoggedIn(request)) {
        if (toUserPanel(request)) {
          log.warning("Admin wants to access UserPanel. Setting
              Response Status to 403.");
          response.sendError(403);
        } else {
          chain.doFilter(req, res);
        }
      } else {
        if (SystemProperty.environment.value() == SystemProperty.
            Environment.Value.Development)
          response.sendRedirect("http://" + serverName + "/login.
              jsp");
        else
          response.sendRedirect("https://" + subdomain + ".appspot
              .com/login.jsp");
      }
    } else {
      if (userIsLoggedIn(request)) {
        if (toAdminPanel(request)) {
          log.warning("User wants to access AdminPanel or
              MapReduce. Setting Response Status to 403.");
          response.sendError(403);
        } else {
          chain.doFilter(req, res);
        }
      } else {
```

```
            if (SystemProperty.environment.value() == SystemProperty.
                Environment.Value.Development)
              response.sendRedirect("http://" + serverName + "/login.
                  jsp");
            else
              response.sendRedirect("https://" + subdomain + ".
                  testbackend.appspot.com/login.jsp");
          }
        }
      }

      private boolean adminIsLoggedIn(HttpServletRequest request) {
        APLoginService service = MyThreadLocal.getServiceFactory().
            apLoginService();
        CTAdmin admin = service.loginFromSessionWithRequest(request);
        if (admin != null) {
          return true;
        }
        return false;
      }
      private boolean userIsLoggedIn(HttpServletRequest request) {
        UPLoginService service = MyThreadLocal.getServiceFactory().
            upLoginService();
        ShopAdmin user = service.loginFromSessionWithRequest(request);
        if (user != null) {
          return true;
        }
        return false;
      }
      private boolean toAdminPanel(HttpServletRequest request) {
        String target = request.getRequestURI();
        boolean retVal = target.startsWith("/admin/adminpanel");
        return retVal;
      }
      private boolean toUserPanel(HttpServletRequest request) {
        String target = request.getRequestURI();
        boolean retVal = target.startsWith("/admin/userpanel");
        return retVal;
      }
    }
```

Listing 4.48: Der LoginFilter

Im LoginFilter wird wieder anhand der Subdomain entschieden, ob nach einem *CTAdmin*, oder einem *ShopAdmin*-Objekt in der Session gesucht werden soll. Wird dort keines gefunden, so wird ein Redirect zur Login-Seite durchgeführt. Da über die Login-Methoden des Services nur geprüft wird, ob ein Admin in der Session ist, und nicht ob dieser Admin die angefragte Ressource auch verwenden darf, muss das ebenfalls im LoginFilter geschehen. Dazu die-

nen die Methoden *toAdminPanel* und *toUserPanel*. Ein eingeloggter *CTAdmin* darf schließlich nicht in das UserPanel gelangen, und umgekehrt ein *ShopAdmin* nicht in das AdminPanel. Ist dies der Fall, so wird der HTTP-Fehler *403 - Forbidden* zurückgegeben, andernfalls ist der Request valide und darf fortfahren (*chain.doFilter()*).
Damit die Benutzernamen und Passwörter nicht ungeschützt zwischen Client und Server übertragen werden, wird in ct.Box SSL über HTTPS verwendet. Aller URLs einer auf der App-Engine laufenden Applikation können generell sowohl über HTTP, als auch über HTTPS abgefragt werden. Im der web.xml kann für bestimmte URLs aber die Verwendung von HTTPS obligatorisch festgelegt werden. Dazu muss ein `<security-constraint>` mit einem `<user-data-constraint>` angelegt werden, dessen `<transport-guarantee>` auf *CONFIDENTIAL* gesetzt ist, wie in Listing 4.49 dargestellt. [Goo11g]

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>HTTPS</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Listing 4.49: Aktivieren von HTTPS

Laut Dokumentation wird das `<web-resource-name>`-Element nicht benötigt [Goo11g], wird es allerdings weggelassen, so wird in Eclipse eine Fehlermeldung angezeigt, weswegen es in der web.xml belassen wurde. In diesem Listing sieht man außerdem, dass alle Anfragen an ct.Box über HTTPS gesendet werden. Wird eine Anfrage über HTTP formuliert, wird diese automatisch auf die selbe URL unter Verwendung von HTTPS weitergeleitet. [Goo11g]

4.4.2 Schutz des Webservice

Neben den Weboberflächen muss auch der Webservice geschützt werden. Das geschieht wie in Kapitel 3.1 gefordert nach dem OAuth-Protokoll mit Bearer Tokens.

Implementierung

Wie in Abschnitt 2.6 bereits angegeben, sind die Clients des OAuth-Protokolls in ct.Box durch die *Application*-Klasse repräsentiert. Eine *Application* besitzt eine Long-Id als eindeutigen Identifikator und eine Referenz auf ein *AccessToken*-

Objekt, das sie dazu verwenden kann, sich beim Webservice zu authentifizieren. Listing 4.50 zeigt die *Application*-Klasse.

```
@Entity
@Cached
@Unindexed
public class Application extends DataStoreObject {

  @Indexed
  private String name;
  private String description;

  @Indexed
  private Key<AccessToken> token;
  private String token_key;

  //Getter und Setter...
}
```

Listing 4.50: Die Application-Klasse

Applications werden in ct.Box auch im Memcache gehalten (*@Cached*), da diese in jedem Aufruf an den Webservice bei der Autorisierung geladen werden. Wie bereits gesagt verwenden die Applications Access Tokens, um sich zu identifizieren. Die *AccessToken*-Klasse besitzt dazu ein String-Feld, das den Schlüssel-String, der zur Authentifizierung verwendet wird, aufnimmt. Dieses Feld ist gleichzeitig die Id eines *AccessToken*-Entities, was zum Einen verhindert, dass zwei *AccessTokens* mit dem selben Schlüssel existieren, und zum Anderen das schnelle Laden eines AccessTokens anhand seines Datastore-Schlüssels ermöglicht. Zusätzlich dazu werden auch die AccessToken-Objekte gecached, um die Autorisierung für den Webservice möglichst performant zu gestalten. In Listing 4.51 ist die Implementierung der Access Tokens in ct.Box zu sehen.

```
@Entity
@Cached
@Unindexed
public class AccessToken {
  @Id
  private String key;
  private Key<Application> application;
  private Date created_at;
  private Date updated_at;
  private Integer version = 0;

  //Getter und Setter...
```

Listing 4.51: Die AccessToken-Klasse

In Restlet erfolgt die Authentifizierung von Requests durch vier Komponenten:

ChallengeScheme Ein *ChallengeScheme* ist das Schema, das verwendet wird, um einen Client zu authentifizieren. Beispiele hierfür wären HTTP-Basic und HTTP-Digest. Ein ChallengeScheme-Objekt wird aber nur dazu verwendet, ein solches Schema zu identifizieren, enthält aber keine Logik um es zu verarbeiten. [Res]

AuthenticatorHelper Ein *AuthenticatorHelper* ist dafür zuständig, die rohen Daten aus dem HTTP-Request in eine dem verwendeten *ChallengeScheme* entsprechende Form zu bringen. Listing 4.52 zeigt den AuthenticatorHelper von ct.Box.

```
public static ChallengeScheme Bearer = new ChallengeScheme("
    HTTP_Bearer", "Bearer");
  public BearerHelper(ChallengeScheme challengeScheme,
      boolean clientSide, boolean serverSide) {
    super(challengeScheme, clientSide, serverSide);
  }
  public BearerHelper() {
    super(Bearer, true, true);
  }

  @Override
  public void parseResponse(ChallengeResponse challenge,
      Request request, Series<Parameter> httpHeaders) {
    String raw = challenge.getRawValue();
    if (raw == null) {
      // Log the blocking
      getLogger().warning(
          "Invalid credentials given by client with IP: "
              + ((request != null) ? request.getClientInfo().
                  getAddress() : "?"));
    } else {
      challenge.setScheme(Bearer);
      challenge.setIdentifier(raw);
    }
  }
}
```

Listing 4.52: Der AuthenticatorHelper von ct.Box

Hier wird außerdem noch das ChallengeScheme-Objekt für die Bearer-Tokens definiert.

ChallengeAuthenticator Ein *ChallengeAuthenticator* authentifiziert einen Request. Dazu muss er die Methode *authenticate* implementieren.[Tur10] Ein

ChallengeAuthenticator ist dabei eine Subklasse des *org.restlet.routing.Filter*, der genau wie ein Servlet-Filter Requests verarbeitet, bevor sie zu ihrem eigentlichen Ziel gelangen. Die *authenticate*-Methode des ChallengeAuthenticators von ct.Box ist in Listing 4.53 abgebildet.

```
public class OAuthenticator extends ChallengeAuthenticator {
  @Override
  protected boolean authenticate(Request request, Response
      response) {
    boolean result = false;
    boolean loggable = getLogger().isLoggable(Level.FINE); /*
        &&request.isLoggable() */
    if (getVerifier() != null) {
      switch (getVerifier().verify(request, response)) {
      case Verifier.RESULT_VALID:
        // Valid credentials provided
        result = true;
        if (loggable) {
          ChallengeResponse challengeResponse = request.
              getChallengeResponse();
          if (challengeResponse != null) {
            getLogger().warning(
                "Authentication succeeded. Valid credentials
                    provided for identifier: "
                    + request.getChallengeResponse().
                        getIdentifier() + ".");
          } else {
            getLogger().warning("Authentication succeeded.
                Valid credentials provided.");
          }
        }
        break;
      case Verifier.RESULT_MISSING:
        // No credentials provided
        if (loggable) {
          getLogger().warning("Authentication failed. No
              credentials provided.");
        }

        if (!isOptional()) {
          challenge(response, false);
        }
        break;
      case Verifier.RESULT_INVALID:
        // Invalid credentials provided
        if (loggable) {
          getLogger().warning("Authentication failed. Invalid
              credentials provided.");
        }
        if (!isOptional()) {
```

```
            if (isRechallenging()) {
              challenge(response, false);
            } else {
              forbid(response);
            }
          }
          break;
        case Verifier.RESULT_STALE:
          if (loggable) {
            getLogger().warning("Authentication failed. Stale
                credentials provided.");
          }
          if (!isOptional()) {
            challenge(response, true);
          }
          break;
        case Verifier.RESULT_UNKNOWN:
          if (loggable) {
            getLogger().warning("Authentication failed.
                Identifier is unknown.");
          }
          if (!isOptional()) {
            if (isRechallenging()) {
              challenge(response, false);
            } else {
              forbid(response);
            }
          }
          break;
        }
      } else {
        getLogger().warning("Authentication failed. No verifier
            provided.");
      }
      return result;
    }
  }
```

Listing 4.53: Authentifizierung im OAuthenticator

Verifier Ein *Verifier* wird wie gezeigt von einem *ChallengeAuthenticator* verwendet, um die Gültigkeit der übergebenen Zugangsdaten zu überprüfen. Dafür greift er auf den im Datastore gespeicherten AccessToken zu und vergleicht ihn mit dem im Request enthaltenen (siehe Listing 4.54).

```
public class OAuthLocalVerifier implements Verifier {

  public AccessToken getLocalToken(String identifier) throws
      NotFoundException {
    ObjectifyDao<AccessToken> dao = new ObjectifyDao<
        AccessToken>(AccessToken.class);
    AccessToken token = dao.ofy().get(AccessToken.class,
        identifier);
    return token;
  }

  protected String getIdentifier(Request request) {
    return request.getChallengeResponse().getIdentifier();
  }

  @Override
  public int verify(Request request, Response response) {
    int result = RESULT_VALID;
    if (request.getChallengeResponse() == null) {
      result = RESULT_MISSING;
    } else {
      try {
        if (!computeVerify(request, response)) {
          result = RESULT_INVALID;
        }
      } catch (NotFoundException e) {
        // The identifier is unknown.
        result = RESULT_UNKNOWN;
      }
    }
    return result;
  }

  public boolean computeVerify(Request request, Response
      response) throws NotFoundException {
    boolean result = false;
    String id = getIdentifier(request);
    AccessToken localToken = getLocalToken(id);
    if (localToken != null && id.equals(localToken.getKey()))
        {
      result = true;
      Long app_id = localToken.getApplication().getId();
      Form responseHeaders = (Form) response.getAttributes().
          get("org.restlet.http.headers");
      if (responseHeaders == null) {
        responseHeaders = new Form();
        response.getAttributes().put("org.restlet.http.
            headers", responseHeaders);
      }
```

```
            responseHeaders.add("app_id", String.valueOf(app_id));
          }
          return result;
        }
      }
```

Listing 4.54: Verifikation des übergebenen AccessTokens

In Listing 4.29 auf Seite 75 war bereits zu sehen, wie diese Komponenten zusammengesetzt werden. Zuerst muss das neue ChallengeScheme bei der Restlet-Engine registriert werden. Das geschieht durch den Aufruf von `Engine.getInstance().getRegisteredAuthenticators().add(new BearerHelper());` Danach wird ein *ChallengeAuthenticator*-Objekt mit dem dazugehörigen *ChallengeScheme* erstellt und diesem der passende Verifier zugewiesen (`guard.setVerifier(new OAuthLocalVerifier());`).
Schließlich wird der Authenticator wie ein Filter vor den Router der Restlet-Applikation gesetzt (`guard.setNext(secureRouter);`), wodurch alle Aufrufe an den Webservice von diesem überprüft werden.

4.4.3 Erkenntnisse

Die Absicherung von ct.Box gegen unautorisierte Zugriffe verlief ohne größere Probleme, jedoch mit einigen kleinen Hindernissen. So sollte die Login-Seite für die Weboberflächen zuerst als eigenes GWT-Modul realisiert werden. Dieses Modul war auch schon fertig implementiert und konnte den Login-Service nutzen, um einen User einzuloggen. Allerdings funktioniert der HTTP-Redirect zwischen GWT-Modulen nicht wie erwartet, weswegen es zum Beispiel nicht möglich war, vom Login-Modul zum AdminPanel weiterzuleiten. Nachdem GWT ein AJAX-Framework ist, werden normalerweise keine Seitenwechsel innerhalb einer Applikation durchgeführt. Soll dies doch geschehen, muss die Funktion *Window.open()* verwendet werden. Diese Funktion ist aber im Login-Filter nicht verfügbar, da dieser nicht im Client, sonder auf dem Server läuft. Aus diesem Grund wurde zum Login dann die hier vorgestellte HTML-Seite in Verbindung mit einem Java-Servlet verwendet. Ansonsten verlief die Absicherung der Weboberflächen aber ohne weiter Probleme.
Anders verlief die Entwicklung beim Webservice. Hier war wieder die Dokumentation von Restlet die entscheidende Schwachstelle. Dort steht zwar ein Minimalbeispiel dafür, wie Aufrufe an Restlet abgesichert werden können und auch, dass es möglich ist, eigene ChallengeSchemes zu verwenden, aber wie das funktioniert wird nirgends beschrieben. Nach einer kurzen Suche konnte in einer Mailing List von Restlet eine Antwort gefunden werden, die zumindest

den groben Ablauf der Authentifizierung und die daran beteiligten Klassen beschreibt. [Tur10] Erst nachdem die Implementierungen dieser Klassen im Sourcecode der Standardimplementierung von Restlet untersucht wurden, konnte mit der Entwicklung für ct.Box begonnen werden. Besonders im Anbetracht der Tatsache, dass es eigentlich keinen großen Aufwand bedeutet, ein eigenes ChallengeScheme und die damit verbundenen Klassen zu erstellen, war die Zeit, die für deren Implementierung benötigt wurde, unverhältnismäßig hoch. Dazu kam, dass während der Entwicklung ein Wechsel von den bereits eingebundenen OAuth-MAC-Tokens zu den Bearer-Tokens vorgenommen wurde, um die Entwicklung von Client-Applikationen zu vereinfachen.
Schließlich hat die Entscheidung, ein eigenes Benutzersystem anstatt der bereits implementierten *User*-API zu verwenden, mehr Aufwand verursacht. Dafür ist die Nutzergruppe von ct.Box nicht auf Personen mit einem Google-Account beschränkt.

4.5 Performance von ct.Box

Wie bereits erwähnt, ist die Performance für den in dieser Arbeit entwickelten Prototyp noch kein kritischer Faktor. Nichtsdestotrotz soll in diesem Abschnitt kurz darauf eingegangen werden.
Als Einheit für die Performance sollen dabei die Antwortzeiten des Webservice dienen. Weiterhin ist die Anzahl der Instanzen, die der App-Engine Scheduler für eine gewisse Anzahl gleichzeitiger Nutzer aufrecht erhält, interessant. Um die Antwortzeiten zu messen, wurde das Tool JMeter[32] verwendet. Mit diesem Tool können durch Multithreading gleichzeitige Aufrufe an den Webservice simuliert werden. Dafür werden sogenannte *Thread Groups* definiert, die angegeben wie viele, und zu welchem Zeitpunkt, Threads gestartet werden sollen. Zusätzlich dazu wurden die"JMeter-Plugins"[33] benutzt, die JMeter um einige nützliche Features, wie flexiblere *Thread Groups* erweitern.
Um die Antwortzeiten zu messen wird die Anzahl der gestarteten Threads Schritt für Schritt erhöht, bis gleichzeitig 100 Threads auf den Webservice zugreifen. Diese 100 Threads sind dabei in 3 Thread-Gruppen aufgeteilt, die jeder als eigener Mandant einen eigenen Namespace verwenden, um auf den Webservice zuzugreifen. Tabelle 4.1 und Abbildung 4.3 zeigen die Ergebnisse dieses Tests.

[32] `http://jakarta.apache.org/jmeter/`
[33] `http://code.google.com/p/jmeter-plugins/`

#Benutzer	Minimum	Durchschnitt	Maximum	#Instanzen
1	303	438	2268	2
5	290	427	4130	3
10	290	437	1748	5
20	292	651	14423	10
50	289	1132	17972	10
100	325	1959	20673	13

Tabelle 4.1: Antwortzeiten in ms beim Abrufen eines Produkts

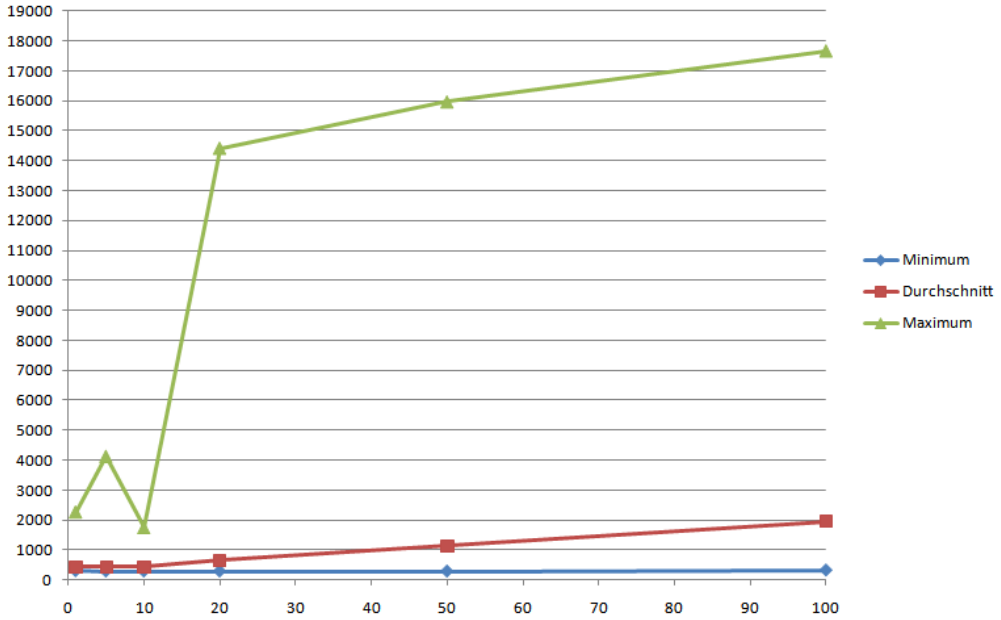


Abbildung 4.3: Performance Diagramm

Wie man erkennen kann, bleibt die durchschnittliche Antwortzeit relativ konstant. Das lässt darauf hoffen, dass ct.Box auch für eine größere Anzahl von Benutzern gut skaliert. Allerdings schießt das Maximum bei 20 gleichzeitigen Nutzern rapide in die Höhe. Das kann sich möglicherweise durch die Anzahl der Instanzen erklären, die die App-Engine dafür startet. Requests, die den Start einer neuen Instanz anstoßen, benötigen nämlich bedeutend länger als Requests an bereits laufende Instanzen. Da beim Spung von 10 auf 20 Nutzer auch doppelt so viele Instanzen gestartet werden, könnte dies der Grund für die hohen Maximalwerte sein.

Auch ist die Anzahl der Instanzen, die für die relativ kleine Menge an Benutzern benötigt wird, noch recht hoch. Das kann dadurch begründet werden, dass ct.Box noch kein Multithreading verwendet, und somit weniger Anfragen pro Instanz verarbeitet werden können. Das Hinzufügen von neuen Instanzen bei Bedarf erfolgt dabei sofort, das Abschalten ungenutzter Instanzen nach ungefähr 30 Minuten. Im Verlauf der Testreihen wurden insgesamt circa 42.000 Requests an ct.Box gesendet, und dabei ungefähr 73% der CPU-Quotas verbraucht (siehe Abbildung 4.4). Rechnet man das hoch, bedeutet das, dass ct.Box ungefähr 57.000 Anfragen bedienen kann, bevor die freien Quotas überschritten werden. Allerdings wurden bei den Tests nur Leseoperationen eingesetzt. Da Schreiboperationen auf dem Datastore aber bekanntlich teurer sind, liegt die exakte Zahl der möglichen Anfragen wohl um einiges niedriger.

Requests

Quotas are reset every 24 hours. Next reset: 13 hours

Resource	Daily Quota			Rate
CPU Time		73%	4.76 of 6.50 CPU hours	Okay
Requests		0%	42,290 of Unlimited	Okay
Outgoing Bandwidth		12%	0.12 of 1.00 GBytes	Okay
Incoming Bandwidth		1%	0.01 of 1.00 GBytes	Okay
Secure Requests		0%	38,823 of Unlimited	Okay
Secure Outgoing Bandwidth		12%	0.12 of 1.00 GBytes	Okay
Secure Incoming Bandwidth		1%	0.01 of 1.00 GBytes	Okay
Backend Usage		0%	$0.00 of $0.72	Okay

Storage

Datastore API Calls		0%	214,179 of Unlimited	Okay
Datastore Queries		0%	175,183 of Unlimited	Okay
Blobstore API Calls		0%	0 of Unlimited	Okay
Total Stored Data		1%	0.01 of 1.00 GBytes	Okay
Blobstore Stored Data		0%	0.00 of 1.00 GBytes	Okay
Data Sent to Datastore API		0%	0.02 of Unlimited GBytes	Okay
Data Received from Datastore API		0%	0.72 of Unlimited GBytes	Okay
Datastore CPU Time		0%	3.56 of Unlimited CPU hours	Okay
Datastore Entity Fetch Ops		0%	170,768 of Unlimited	Okay
Datastore Entity Put Ops		0%	38,613 of Unlimited	Okay
Datastore Entity Delete Ops		0%	33 of Unlimited	Okay
Datastore Index Write Ops		0%	887,575 of Unlimited	Okay
Datastore Query Ops		0%	175,182 of Unlimited	Okay
Datastore Key Fetch Ops		0%	155,230 of Unlimited	Okay
Datastore Id Allocation Ops		0%	6 of Unlimited	Okay
Number of Indexes		6%	13 of 200	Okay
Prospective Search Subscriptions		0%	0 of 10,000	Okay

Abbildung 4.4: Bei den Tests verbrauchte Quotas

5 Fazit

In dieser Arbeit wurde die Entwicklung einer Multi-mandantenfähigen Architektur auf der Google App-Engine untersucht. Dazu wurden zuerst die grundlegenden Aspekte der Google App Engine und der in ct.Box eingesetzten Konzepte, wie REST und OAuth, dargestellt, bevor die Implementierung von ct.Box eingehender beschrieben wurde. Zu diesem Zweck wurde ein Überblick über die verfügbaren Frameworks gegeben und diese mit Alternativen verglichen, ehe deren Einsatz am Beispiel von ct.Box erläutert wurde.
Die Erfahrungen, die im Laufe der Implementierung gesammelt werden konnten, waren dabei größtenteils positiv, und die Ziele der Arbeit konnten erfüllt werden. ct.Box läuft als multimandantenfähige Applikation auf der Google App Engine, und kann über einen Webservice angesprochen werden.
Nach einer kurzen Einarbeitung in die Webprogrammierung mit Java-Servlets im Allgemeinen, und den Einschränkungen der App Engine im speziellen, kann sehr schnell mit der Implementierung begonnen werden. Vor allem erfahrene Java-Entwickler sollten keinerlei Probleme haben, in die Entwicklung einzufinden. Die oft bemängelten Probleme, die der Datastore bei der Portierung von RDBMS-Applikationen mit sich bringt, sind bei ct.Box nicht aufgetreten, da es ja von Grund auf für die App Engine entwickelt wurde.
Besonders die Einführung von Multi-Mandantenfähigkeit in eine GAE-Applikation hat sich dank der Namespaces-API als ausgesprochen einfach dargestellt. Zwar sollten Entwickler verstehen, wie die Namespaces im Datastore umgesetzt werden, die letztendliche Implementierung beschränkt sich aber dann doch auf den Aufruf einer einzigen Methoden zum Setzen eines Namespace. Hier erledigt die App Engine gute Arbeit dabei, Entwicklern Aufwand abzunehmen.
Allein die Umsetzung der Beziehungen im Datastore bedarf ein wenig an Gewöhnung. Ansonsten spielten die Einschränkungen des Datastore in ct.Box keine große Rolle. Das einzige Feature, das vermisst wurde, war die Volltextsuche auf den gespeicherten Entities. Hierbei ist festzuhalten, dass Objectify den Umgang mit dem Datastore erheblich vereinfacht.
Im Allgemeinen sind die Einschränkungen der App Engine bei ct.Box nur sehr begrenzt in Erscheinung getreten. Das liegt wie gesagt daran, dass ct.Box von Anfang an für die App-Engine konzipiert wurde, und deren Restriktionen dabei schon berücksichtigt wurden. Die Portierung bereits bestehender Applika-

tionen stellt sich mit Sicherheit schwieriger dar.
Im Gegensatz zur App Engine, haben sich aber die Restriktionen des Google Web Toolkit als unumgänglich herausgestellt. Neben den enormen Mengen an Quellcode, die zum Erstellen der Benutzerberfläche nötig waren, mussten für die Verwendung der RequestFactory sowohl für die Service-, als auch für die Persistenzschicht, mit den Proxy-Objekte Zwischenschichten erstellt werden, die auch noch manuell gepflegt werden müssen. Während der Entwicklung von ct.Box ist es mitunter einige Male vorgekommen, dass nach kleineren Änderungen am Datenmodell vergessen wurde, diese auch in den Proxies umzusetzen. Die dadurch entstanden Laufzeitfehler kosten Zeit und sind doch vermeidbar. Besonders unpraktikabel war aber der Umgang mit den Proxy-Objekten selbst. Die Tatsache, dass ein solches Objekt nach jedem Aufruf "gesperrt" wird, ist vollkommen ungewohnt, und selbst nach einer längeren Eingewöhnungszeit konnte immer noch nicht genau festgestellt werden, wann diese Sperrung auftritt und wann nicht. Genau dieses Verhalten ist es zwar, das die Performanz der RequestFactory ermöglicht, aber mit wiederverwendbaren Objekten, wie sie in GWT-RPC existieren, ist leichter zu arbeiten. Und selbst eigentlich nützliche Features, wie die automatische Validierung der Proxies, führen bei der RequestFactory zu weiteren Problemen. Das kann daran liegen, dass die RequestFactory eine relativ neue Funktion von GWT ist. Im Nachhinein ist die Entscheidung, die RequestFactory zu verwenden, nicht mehr so eindeutig, wie sie zuerst erschien. Die einfachere Nutzung der Objekte mit den RPCs kann deren Mehraufwand in der Implementierung möglicherweise aufwiegen. Um hier eine fundierte Entscheidung treffen zu können, müssten beide Ansätze implementiert und miteinander verglichen werden, was im Rahmen dieser Arbeit nicht möglich war. Es wäre theoretisch sogar möglich, komplett auf die RequestFactory und die RPCs zu verzichten, da Restlet eine Client-Bibliothek für GWT anbietet. Mit dieser wäre es möglich, den Webservice von ct.Box als Serviceschnittstelle für die Weboberflächen zu nutzen. Allerdings müssen in auch in diesem Proxies für die Ressourcen erstellt werden, und die Entities müssen GWT-kompatibel sein, da sonst auch für diese wieder DTOs erstellt werden müssen.
Über den gesamten Verlauf der Arbeit hat sich die Dokumentation der Google App-Engine, ebenso wie die des Google Web-Toolkits und von Objectify, als äußerst hilfreich erwiesen, und sollte als erste Anlaufstelle bei Problemen genutzt werden. Die von Google bereitgestellte Dokumentation deckt dabei die meisten Fragen ab, die für einen Entwickler interessant sind, und nur in seltenen Fällen sollte es nötig sein, anderweitige Quellen zu Rate zu ziehen. Anders war die Erfahrung mit Restlet, in dessen Dokumentation die Konzepte meist nur durch Minimalbeispiele veranschaulicht werden. Dort musste viel Zeit mit der Suche nach Lösungen und Referenzimplementierungen verbracht werden,

da die Zusammenhänge nicht sofort ersichtlich waren.
Wirft man einen Blick auf die Zukunft von ct.Box, so bleibt noch einiges offen. Im Moment nimmt die Applikation fertige Bestellungen entgegen und kann diese speichern und wieder zurückgeben. Die Erstellung einer Bestellung und die komplette Warenkorb-Logik muss aber noch in den Client-Applikationen erfolgen, da hierfür noch keine Implementierung in ct.Box existiert. Hierfür könnte zum Beispiel die Google-Checkout-API [1] verwendet werden.
Da in dieser Arbeit nur ein Prototyp von ct.Box entwickelt wurde, wurde nicht explizit auf die Performance der Applikation geachtet, da diese auch im Nachhinein noch verbessert werden kann. Wichtiger war es, eine funktionierende Applikation mit allen Grundfunktionalitäten auf die Beine zu stellen. Um zum Beispiel die Performance der Weboberflächen zu verbessern, könnten zusätzlich noch eigene Entities für die Listenanzeige erstellt werden, die nur die Felder besitzen, die auch in der Liste angezeigt werden. Erst wenn der Nutzer in eine Detailansicht geht, müssten dann die kompletten Objekte verwendet werden. Sollten der Fehler in den Filtern von Jackson behoben werden, könnte damit der Webservice beschleunigt werden, da somit die Größe der gesendeten Objekte um ein Vielfaches abnimmt.
Generell kann die Performance noch insofern verbessert werden, dass ct.Box multi-threaded auf der App Engie läuft. Das gewinnt insbesondere deswegen an Bedeutung, da vor kurzem angekündigt worden ist, dass die App-Engine die Preview-Version verlässt. Gleichzeitig dazu wird das Preismodell angepasst werden. Dabei werden die freien Ressourcen gekürzt und die Preisberechnung vom CPU-Verbrauch auf Instanz-Stunden (eine Apllikationsinstanz, die für eine Stunde läuft) umgestellt. Dazu werden alle Aufrufe an APIs, die bislang per CPU-Stunde abgerechnet wurden, zukünftig stattdessen pro Aufruf abgerechnet.[Goo11r] Dadurch ergeben sich teilweise enorm gestiegene Preise, vor allem für die Entwickler kleinerer Applikationen, die mit diesen kein Geld verdienen. Die Applikation PlusFeed zum Beispiel kostet mit dem neuen Modell anstatt 2,63$ nun 68,46$ pro Tag.[Bro11a] Die voraussichtlichen Kosten für ct.Box (basierend auf der Nutzung des Tages, an dem die Lasttests durchgeführt wurden) liegen jedoch nur bei ungefähr 3$ pro Tag. Besonders da Applikationen von der App-Engine praktisch nicht auf andere Cloud-Anbieter portierbar sind Um die Kosten von ct.Box niedrig zu halten, wird es in Folge dessen wichtiger, die Anzahl der aktiven Instanzen möglichst gering zu halten. Hier wird Multithreading hoffentlich den gewünschten Erfolg bringen. Gleichzeitig zu den neuen Preisen werden aber endlich auch Service Level Agreements für zahlende Kunden eingeführt, die 99,95% Verfügbarkeit verspricht. [Goo11r]

[1] `https://checkout.google.com/sell/`

Abschließend ist festzustellen, dass mit der Google App-Engine eine hervorragende Plattform zum Entwickeln von Multi-mandantenfähigen Applikationen zur Verfügung steht. Die angekündigten Änderungen lassen darauf schließen, dass Google mit der App-Engine den Schritt zu einer Plattform für Business-Anwendungen machen möchte. Dabei könnten aber die vielen kleinen Applikationen, die bis jetzt die freien Ressourcen der App-Engine nutzten, auf der Strecke bleiben.

Anhang

Use-Cases

Use-Case	Produkt anlegen
Beteiligte Akteure	Shop-Betreiber (User)
Standardablauf	1. Der User schickt ein Produkt-JSON-Objekt per HTTP-POST an die Produkt-Ressource von ct.Box. 2. ct.Box validiert die Korrektheit des empfangenen Produkts 3. ct.Box speichert das Produkt und sendet es an den User zurück. 4. Der User empfängt das neu angelegte Produkt.
Alternativer Ablauf	3. ct.Box sendet ein JSON-Objekt mit den invaliden Feldern des Produkts zurück. 4. Der User empfängt eine Fehlermeldung.
Vorbedingungen	Der User authentifiziert sich erfolgreich am Webservice.
Ergebnis	Ein neues Produkt wurde im Datastore gespeichert.

Tabelle 5.1: Erstellen eines Produkts

Use-Case	Produkt ändern
Beteiligte Akteure	Shop-Betreiber (User)
Standardablauf	1. Der User schickt ein Produkt-JSON-Objekt per HTTP-PUT an die Produkt-Ressource von ct.Box. 2. ct.Box validiert die Korrektheit des empfangenen Produkts 3. ct.Box überschreibt das geänderte Produkt und sendet es an den User zurück. 4. Der User empfängt das geänderte Produkt.
Alternativer Ablauf	3. ct.Box sendet ein JSON-Objekt mit den invaliden Feldern des Produkts zurück. 4. Der User empfängt eine Fehlermeldung.
Vorbedingungen	Der User authentifiziert sich erfolgreich am Webservice.
Ergebnis	Die Felder eines existierenden Produkts wurden geändert.

Tabelle 5.2: Ändern eines Produktes

Use-Case	Produkt löschen
Beteiligte Akteure	Shop-Betreiber (User)
Standardablauf	1. Der User ruft die Produkt-Ressource per HTTP-DELETE mit der Produkt-ID als Parameter auf. 2. ct.Box löscht das Produkt mit der angegebenen ID. 3. Der User empfängt eine Antwort mit dem HTTP-Statuscode 200.
Alternativer Ablauf	2. ct.Box findet kein Produkt mit der angegebenen ID im Datastore 3. Der User empfängt eine Fehlermeldung.
Vorbedingungen	Der User authentifiziert sich erfolgreich am Webservice.
Ergebnis	Ein Produkt wurde aus dem Datastore gelöscht.

Tabelle 5.3: Löschen eines Produktes

Use-Case	Mehrere Produkte auslesen
Beteiligte Akteure	Shop-Betreiber (User)
Standardablauf	1. Der User ruft die Produkt-Ressource per HTTP-GET auf. 2. ct.Box lädt die angeforderten Produkte aus dem Datastore. 3. ct.Box sendet die geladenen Produkte an den User. 4. Der User empfängt ein JSON-Objekt mit den angeforderten Produkten.
Vorbedingungen	Der User authentifiziert sich erfolgreich am Webservice.
Ergebnis	Eine Liste von Produkten wurde empfangen.

Tabelle 5.4: Auslesen mehrerer Produkte

Use-Case	Einzelnes Produkt auslesen
Beteiligte Akteure	Shop-Betreiber (User)
Standardablauf	1. Der User ruft die Produkt-Ressource per HTTP-GET mit der Produkt-ID als Parameter auf. 2. ct.Box lädt das Produkt mit der angegebenen ID. 3. ct.Box sendet das geladene Produkt an den User zurück. 4. Der User empfängt das Produkt als JSON-Objekt.
Alternativer Ablauf	2. ct.Box findet kein Produkt mit der angegebenen ID im Datastore 3. ct.Box sendet eine Fehlermeldung zurück. 4. Der User empfängt die Fehlermeldung.
Vorbedingungen	Der User authentifiziert sich erfolgreich am Webservice.
Ergebnis	Ein Produkt wurde aus dem Datastore ausgelesen.

Tabelle 5.5: Lesen eines Produktes

Abkürzungsverzeichnis

Abkürzung	Bedeutung
A&P	Activities und Places
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CRM	Customer-Relationship-Management
CRUD	Create,Read,Update,Delete
CSS	Cascading Style Sheets
DAO	Data Access Object
DTO	Data Transfer Object
GAE	Google App Engine
GWT	Google Web Toolkit
HTML	Hypertext Markup Language
IT	Information Technology
IaaS	Infrastructure-as-a-Service
JDO	Java Data Objects
JPA	Java Persistence API
JRE	Java Runtime Environment
JSNI	Javascript Native Interface
JSON	Javascript Object Notation
MVC	Model-View-Controller
MVP	Model-View-Presenter
NIST	National Institute of Standards and Technology
POJO	Plain old Java Object
PaaS	Platform-as-a-Service
RDBMS	Relational Database Management System
REST	Representational State Transfer
RPC	Remote Procedure Call
SDK	Software Development Kit
SLA	Service Level Agreement
SQL	Structured Query Language
SSL	Secure Sockets Layer
SaaS	Software-as-a-Service
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

Abbildungsverzeichnis

Literaturverzeichnis

[Bro11a] Joe Brockmeier. Google app engine pricing angers developers, kills plusfeed. http://www.readwriteweb.com/hack/2011/09/google-app-engine-pricing-ange.php, 2011. Zugriff am 03.09.2011.

[Bro11b] Thomas Broyer. Gwt 2.1.1 requestfactory. http://tbroyer.posterous.com/gwt-211-requestfactory, 10. Januar 2011. Zugriff am 05.08.2011.

[CCW06] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. Multi-tenant data architecture. http://msdn.microsoft.com/en-us/architecture/aa479086, 2006. Zugriff am 19.07.2011.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, November 2006. Abrufbar unter http://labs.google.com/papers/bigtable-osdi06.pdf.

[Cha10] David Chandler. Generic dao for objectify 2. http://turbomanage.wordpress.com/2010/02/09/generic-dao-for-objectify-2/, 09. Februar 2010. Zugriff am 19.07.2011.

[Com] JBoss Community. Hibernate validator. http://www.hibernate.org/subprojects/validator.html. Zugriff am 05.08.2011.

[Dau10] Markus Dauberschmidt. Use of the google app engine for business web applications: potentials & restrictions. Master's thesis, TU München, 2010.

[DDM06] Alexander Deiss, Bettina Druckenmüller, and Antje Melle. Soa vs. rest basierend auf google, ebay, amazon, yahoo! http://elib.uni-stuttgart.de/opus/volltexte/2006/2618/pdf/FACH_0056.pdf, 2006. Zugriff am 05.08.2011.

[Dis11] Reslet Discuss. Incompatible classchange error with concurrentmap. `http://restlet.tigris.org/ds/viewMessage.do?dsForumId=4447&dsMessageId=2745795`, 2011. Zugriff am 05.08.2011.

[F+] Roy Fielding et al. Method definitions. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`. Zugriff am 05.08.2011.

[FE10] Borko Furht and Armando Escalante, editors. *Handbook of Cloud Computing*. Springer, 2010.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. Kapitel 5, Abrufbar unter `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`, Zugriff am 05.08.2011.

[Goo11a] Google. App engine product roadmap. `http://code.google.com/intl/de-DE/appengine/docs/roadmap.html`, 2011. Zugriff am 27.08.2011.

[Goo11b] Google. Choosing a datastore (java). `http://code.google.com/intl/de-DE/appengine/docs/java/datastore/hr/index.html`, 2011. Zugriff am 12.07.2011.

[Goo11c] Google. Coding basics - deferred binding. `http://code.google.com/intl/de-DE/webtoolkit/doc/latest/DevGuideCodingBasicsDeferred.html`, 2011. Zugriff am 05.08.2011.

[Goo11d] Google. Communicating with the server. `http://code.google.com/intl/de-DE/webtoolkit/doc/latest/tutorial/clientserver.html`, 2011. Zugriff am 05.08.2011.

[Goo11e] Google. Datastore overview (java). `http://code.google.com/intl/de-DE/appengine/docs/java/datastore/overview.html`, 2011. Zugriff am 12.07.2011.

[Goo11f] Google. Declarative layout with uibinder. `http://code.google.com/intl/de-DE/webtoolkit/doc/latest/DevGuideUiBinder.html`, 2011. Zugriff am 05.08.2011.

[Goo11g] Google. The deployment descriptor: web.xml. http://code.google.com/intl/de-DE/appengine/docs/java/config/webxml.html, 2011. Zugriff am 05.08.2011.

[Goo11h] Google. Entities, properties, and keys. http://code.google.com/intl/de-DE/appengine/docs/java/datastore/entities.html, 2011. Zugriff am 05.08.2011.

[Goo11i] Google. Getting started with requestfactory. http://code.google.com/intl/de-DE/webtoolkit/doc/latest/DevGuideRequestFactory.html, 2011. Zugriff am 05.08.2011.

[Goo11j] Google. Google web toolkit overview. http://code.google.com/intl/de-DE/webtoolkit/overview.html, 2011. Zugriff am 05.08.2011.

[Goo11k] Google. Gwt development with activities and places. http://code.google.com/intl/de-DE/webtoolkit/doc/latest/DevGuideMvpActivitiesAndPlaces.html, 2011. Zugriff am 05.08.2011.

[Goo11l] Google. Implementing multitenancy using namespaces. http://code.google.com/intl/de-DE/appengine/docs/java/multitenancy/multitenancy.html, 2011. Zugriff am 05.08.2011.

[Goo11m] Google. Java application configuration. http://code.google.com/intl/de-DE/appengine/docs/java/config/appconfig.html, 2011. Zugriff am 05.08.2011.

[Goo11n] Google. Making remote procedure calls. http://code.google.com/intl/de-DE/webtoolkit/doc/latest/tutorial/RPC.html, 2011. Zugriff am 05.08.2011.

[Goo11o] Google. Organize projects. http://code.google.com/intl/de-DE/webtoolkit/doc/latest/DevGuideOrganizingProjects.html, 2011. Zugriff am 05.08.2011.

[Goo11p] Google. Using jdo with app engine. http://code.google.com/intl/de-DE/appengine/docs/java/datastore/jdo/overview.html, 2011. Zugriff am 05.08.2011.

[Goo11q] Google. What is google app engine? `http://code.google.com/intl/de-DE/appengine/docs/whatisgoogleappengine.html`, 2011. Zugriff am 05.08.2011.

[Goo11r] Google. The year ahead for google app engine! `http://googleappengine.blogspot.com/2011/05/year-ahead-for-google-app-engine.html`, 10. Mai 2011. Zugriff am 02.09.2011.

[Gro09a] Bean Validation Expert Group. *JSR 303: Bean Validation*. Red Hat, 12. Oktober 2009. Abrufbar unter `http://download.oracle.com/otndocs/jcp/bean_validation-1.0-fr-oth-JSpec/`.

[Gro09b] JSR 311 Expert Group. *JAX-RS: Java API for RESTful Web Services*. Sun Microsystems, 17. September 2009. Abrufbar unter `http://download.oracle.com/otndocs/jcp/jaxrs-1.1-mrel-eval-oth-JSpec/`.

[Haz10] Vikas Hazrati. Multitenancy in google app engine: Scope of namespacemanager. `http://thoughts.inphina.com/2010/09/16/multi-tenancy-in-google-app-engine-scope-of-namespacemanager/`, 16. September 2010. Zugriff am 05.08.2011.

[HLBA11] E. Hammer-Lahav, A. Barth, and B. Adid. Http authentication: Mac access authentication (draft). `http://tools.ietf.org/pdf/draft-hammer-oauth-v2-mac-token-05.pdf`, 2011. Zugriff am 05.08.2011.

[HLRH11] E. Hammer-Laha, D. Recordon, and D. Hard. The oauth 2.0 authorization protocol (draft). `http://tools.ietf.org/pdf/draft-ietf-oauth-v2-21.pdf`, 2011. Zugriff am 06.09.2011.

[JA111] Jsonfilter: add global filters that apply to all beans during serialization. `http://jira.codehaus.org/browse/JACKSON-501`, 2011. Zugriff am 05.08.2011.

[Jer] Jersey. Jersey 1.9 user guide. `http://jersey.java.net/nonav/documentation/latest/user-guide.html`. Zugriff am 05.08.2011.

[KE11] Alfons Kemper and Andre Eickler. *Datenbanksysteme*. Oldenbourg Verlag, 8 edition, 2011.

[MG11] Peter Mell and Timothy Grance. The nist definition of cloud computing (draft). http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf, 2011. Zugriff am 05.08.2011.

[Mil10] Damien Miller. jbcrypt. http://www.mindrot.org/projects/jBCrypt/, 2010. Zugriff am 05.08.2011.

[Obja] Objectify. Best practices. http://code.google.com/p/objectify-appengine/wiki/BestPractices#Automatic_Scanning. Zugriff am 05.08.2011.

[Objb] Objectify. Introductionto objectify. http://code.google.com/p/objectify-appengine/wiki/IntroductionToObjectify. Zugriff am 05.08.2011.

[Objc] Objectify. Objectify project home. http://code.google.com/p/objectify-appengine/. Zugriff am 05.08.2011.

[O'C09] John O'Conner. Creating extensible applications with the java platform. http://java.sun.com/developer/technicalArticles/javase/extensible/, September 2009. Zugriff am 05.08.2011.

[Res] Restlet. Security package. http://wiki.restlet.org/docs_2.1/13-restlet/27-restlet/46-restlet.html. Zugriff am 05.08.2011.

[Sal11] Tatu Saloranta. Feature: Json filter. http://wiki.fasterxml.com/JacksonFeatureJsonFilter, 2011. Zugriff am 05.08.2011.

[Sch09] Erick Schonfeld. The efficient cloud: All of salesforce runs on only 1,000 servers. http://techcrunch.com/2009/03/23/the-efficient-cloud-all-of-salesforce-runs-on-only-1000-servers 23. März 2009. Zugriff am 05.08.2011.

[Sho11] Shopify. Api docs introduction. http://api.shopify.com/, 2011. Zugriff am 05.08.2011.

[Sli] Slim3. Slim3 home. https://sites.google.com/site/slim3appengine/. Zugriff am 05.08.2011.

[TP10] Twig-Persist. Overriding default behaviour. http://code.google.com/p/twig-persist/wiki/Cookbook, 2010. Zugriff am 05.08.2011.

[TP11] Twig-Persist. Comparison to other datastore interfaces. `http://code.google.com/p/twig-persist/wiki/Comparison`, 2011. Zugriff am 05.08.2011.

[Tur10] Garry Turkington. Re: Comprehensive example of authentication in restlet 2.0? `http://www.mail-archive.com/discuss@restlet.tigris.org/msg10449.html`, 28. Januar 2010. Zugriff am 05.08.2011.

[Wika] Reslet Wiki. Restlet edition for google app engine. `http://wiki.restlet.org/docs_2.1/13-restlet/275-restlet/252-restlet.html`. Zugriff am 05.08.2011.

[Wikb] Wikipedia. Cloud computing. `http://de.wikipedia.org/wiki/Cloud_Computing`. Zugriff am 05.08.2011.

[Wikc] Wikipedia. Representational state transfer. `http://en.wikipedia.org/wiki/Representational_state_transfer`. Zugriff am 05.08.2011.