Transaktionen im Kontext von REST-basierten Web-Diensten

Martin Mensing-Braun

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2015

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 19. Juni 2015

Martin Mensing-Braun

Inhaltsverzeichnis

Kurzfassung

Reisebuchungen über sogenannte Single-Page-Web-Applikationen die ihre Daten von verschiedenen von einander unabhängigen Servern mit Hilfe von REST-Anfragen beziehen können schnell zu Problemen führen. Diese Applikationen senden verschiedene asynchrone Anfragen an ihre Server, diese sind schwer konsistent zu Halten, da im Fehlerfall einer Anfrage keine Möglichkeit besteht die restlichen bereits abgesetzten Anfragen rückgängig zu machen, da REST zustandslos ist und somit unabhängig von den anderen Anfragen.

Eine solide Lösung ist daher die Implementierung von REST-Transaktionen auf der Client-Seite mit Hilfe von verschiedenen Pattern, wie das TCC (Try-Catch-Cancel-Pattern) oder anderen Workflowengines welche die Kommunikation zwischen dem Client und den Servern abwickeln. Diese Implementierungen soll in der Form einer JavaScript-API auf der Client-Seite realisiert werden und hauptsächlich mit Standard-REST-Schnittstellen auf der Server-Seite kommunizieren. Diese Lösung soll es dem Entwickler einfach machen, REST-Transaktionen zu implementieren.

In einem abschließenden Vergleich werden die selbst gefundenen und vorhandenen Lösungen gegenübergestellt.

Abstract

Booking a journey on a single-page-web-application including different backendserver using RESTful-Requests, which one are not connected to each other, can be a tricky thing. The web-application is sending different asynchronous requests to the backends, which is hard to keep consistent, because if one server-request fails, there is no way to cancel the request to the other server. The main issue is that REST is stateless and isn't aware of any other Requests happening before or afterwards.

A solid solution could be to implement RESTful-transactions on the client side using different patterns, like TCC (Try-Catch-Cancel Pattern) or other workflowengines to manage the communication between the client and the different servers. These implementations should offer a JavaScript-API on the clientside and some standard-REST-interface on the backendside. The solution found should make it easy for the developer to implement these transactions.

In a final comparison all own and existing solutions will be analysed.

Kapitel 1

Einleitung

Moderne Webapplikationen setzen im Moment immer mehr auf sogenannte Single-Page-JavaScript Applikationen. Diese Applikationen besitzen den Vorteil, dass die Applikation selbst im Browser ausgeführt wird und somit in der Lage ist, Daten von dezentralisierten Webservices zu beziehen. Die Kommunikation zu diesen Servern wird über zustandslose REST-Schnittstellen abgewickelt. Durch die Zustandslosigkeit der Anfragen kann es bei Fehlern jedoch schnell zu Dateninkonsistenzen auf einem oder mehreren der angesteuerten Servern kommen.

Diese Arbeit soll sich daher mit Lösungen beschäftigen, die es ermöglichen REST-Services über mehrere Server innerhalb von REST-Transaktionen oder etwas vergleichbaren zu Verwalten, um im Fehlerfall die Sicherheit der Datenbasis auf verschiedenen Systemen zu gewährleisten.

Da der Vorteil dieser JavaScript-Applikationen schon lange bekannt ist, existieren bereits einige Lösungen. Der Nachteil dieser Lösungen ist jedoch, dass sie meist nur für einen Webserver funktionieren oder extrem eingeschränkt anwendbar sind. Die Realisierung für die Abwicklung von REST-Transaktionen stellt sich daher als interessant dar, da gute Lösungen dringend benötigt werden und stetig auf diversen Konferenzen diskutiert werden und somit eine gute Basis für Weiterentwicklung bieten. Trotz des hohen Aufmerksamkeitsgrades wurden bis zum heutigen Tage fast keine brauchbaren Lösungen gefunden, vor allem Implementierungswege die komplett ohne Unterstützung des jeweiligen Webservices auskommen.

In Kapitel 2 dieser Arbeit werden die technischen Grundlagen der verwendeten Technologien beschrieben, um eine fundamentale Wissensbasis für die beschriebenen Ansätze zu schaffen. Im anschließenden Kapitel werden die Rahmenbedingungen für valide Transaktionen erläutert, bereits vorhandene Client- und Server-Basierte Lösungen abgesteckt und auf ihre Brauchbarkeit analysiert. Anhand dieser Ansätze wird in Kapitel 4 einerseits ein dezentraler Weg mit Serverunterstützung und andererseits eine Lösung ohne Unterstützung des Webservices gefunden. Diese bilden dann die theoretische

Basis der nachfolgenden eigenen Implementierungen für clientseitige Transaktionsmanager in Kapitel 6, die das Herzstück dieser Arbeit bilden. Nach den Erläuterungen der Umsetzungen werden diese gründlich im Kapitel 7 durchleuchtet, auf ihre Verlässlichkeit geprüft und verglichen.

Kapitel 2

Technischer Hintergrund

Single-Page-Applikationen werden aufgrund der gut skalierbaren Architektur immer populärer. Diese Applikationen beziehen oftmals ihre Ressourcen mit Hilfe von REST-Schnittstellen von ihren Web-Services. Daher handelt dieses Kapitel REST mit seinen wichtigsten Funktionsweisen, Beschränkungen und Richtlinien ab. Darüber hinaus wird in der Sektion *Cross-Domain-REST* besprochen, welche Möglichkeiten es gibt die *Same-Origin-Policy* zu umgehen, um die Kommunikation zwischen den Instanzen möglich zu machen. Da die meisten gängigen Webframeworks asynchrone Serverabfragen mit REST als sogenannten *Promises* zurückliefern, werden diese am Kapitel-Ende im Detail beschrieben.

2.1 REST

Representational State Transfer, kurz *REST*, ist ein flexibles und gut skalierbares Programmierparadigma, anwendbar für Client-Server Kommunikationen und ist somit ein fester Bestandteil in vielen Client und Serverbasierenden Webapplikationen. Die Grundidee dieses Konzepts ist es eine Ansammlung von Ressourcen auf der Serverseite zur Verfügung zu stellen, die mit verschiedenen Methoden von einem Client aus behandelt werden können [1, Kap. 5].

Jeder einzelne dieser Methodenaufrufe wird als zustandslos bezeichnet und jeder Aufruf ist somit komplett unwissend und unabhängig von seinem Vorgängern oder Nachfolgern. Ein weiterer wichtiger Fakt ist, dass alle Ergebnisse asynchron zurückgeliefert werden.

Jede dieser Ressourcen kann theoretisch auf einem anderen Server liegen und mit Hilfe von *fetch*, *modifiy* oder *delete* Operationen manipuliert werden. Diese Operationen werden in einem CRUD-Interface (*create, read, update und delete*) zusammengefasst. Diese Schnittstelle ist zwar kein Teil eines offiziellen Standards, jedoch wird der beschriebene Weg in den meisten Implementationen erläutert und verwendet. Je nach Operation werden verschiede-

Tabelle 2.1: Übersicht HTTP-Methoden mit Muster-URLs [4, S. 331].

HTTP-Method	**URI**	**Aktion**
GET	URL/resourceName URL/resourceName/id	Abfragen
POST	URL/resourceName	Erzeugen
PUT	URL/resourceName/id	Manipulieren
PATCH	URL/resourceName/id	Manipulieren
DELETE	URL/resourceName/id	Löschen

ne HTTP-Methoden wie *GET*, *POST*, *PUT*, *PATCH* und *DELETE* herangezogen. Dieses Interface ist somit in der Lage vielen Backend-Frameworks in verschiedenen Programmiersprachen eine gemeinsame Schnittstelle zu bieten die es ermöglicht auf diese Ressourcen zu zugreifen [12, 4, S. 329–330].

Im allgemeinen wird die HTTP-Methode *GET* verwendet, um eine Lese-Abfrage durchzuführen und somit alle in Frage kommenden Einträge einer Quelle abzurufen. Diese Abfrage wird mit dem URL-Schema *URL/resourceName* für alle Objekte und *URL/resourceName/id* für einzelne Objekte angewandt [4, S. 331].

Für neue Einträge (*Create*) wird mit HTTP-*POST* oder auch *PUT* realisiert, hierbei wird das selbe URL-Schema (*URL/resourceName*) herangezogen und die Daten des neuen Objekts im Body des Aufrufs gesendet. Um Datensätze zu aktualisieren wird HTTP-*PUT* verwendet, wieder wird der aktuelle Datensatz mit allen Parametern, egal ob verändert oder nicht, in den Body inkludiert und der Aufruf an den Server mit der entsprechenden ID geschickt (*URL/resourceName/id*). Als Alternative für *PUT* gibt es *PATCH*, hierbei müssen nur im Gegensatz zu PUT die geänderten Parameter übertragen werden und nicht der komplette Datensatz [14, 9, S. 97–105].

Der *Destroy*-Command wird für das Löschen von Datensätzen angewandt, wird über die HTTP-*DELETE* realisiert und wieder mit der ID in der URL spezifiziert [15].

2.1.1 Einschränkungen (Constraints)

Um REST-Schnittstellen und Applikationen richtig gestalten und somit eine leicht zu verwendende Plattform zur Verfügung stellen zu können, sollen sogenannte REST-Beschränkungen berücksichtigt werden. Jede Beschränkung

ist eine vorgegebene Design-Entscheidung die positive und negative Effekte an der Schnittstelle mit sich zieht [1, 19, S. 76].

Client-Server

Diese Beschränkung beschreibt die Trennung zwischen der Client-Logik und der Server-Logik. Wie bereits in der Einleitung kurz beschrieben, bringt diese Einschränkung den Vorteil, dass ein Client mit mehreren Servern kommunizieren kann. Wobei der Client für jeden Aufruf eine Antwort bekommen muss. Diese Aufrufe werden auf der Server-Seite durch eine Logik abgearbeitet [1, S. 78].

Alle aufkommenden Ausnahmen die den Aufruf stören, müssen zurück an den Client gebracht werden und dieser muss entsprechende Maßnahmen treffen. Ein weiterer Vorteil ist, dass durch diese Architektur der Client nicht alle Informationen des Servers erfahren muss und schmälert somit den Mehraufwand auf der Client-Seite [19].

Zustandslosigkeit (Statelessness)

Wie bereits beschrieben, sind alle REST-Requests zustandslos. Dies bedeutet das jede Anfrage alle Informationen enthalten muss, sodass der Server sie vollständig ohne weitere Requests verarbeiten kann. Daher muss der Service bei Erfolg alle sitzungs-relevanten Daten an den Client zurückgeben.

Um zustandslos zu bleiben ist es dem Server nicht erlaubt den aktuellen Status während der Abfrage zu teilen . Ist die Request abgeschlossen, kennt der Service keine requestspezifischen Informationen mehr. Die Serverlogik darf keine Runtime spezifischen Daten speichern, jedoch aber Daten die für die Serverlogik von Wichtigkeit sind [1, 19, S. 78–79].

Cache

Antworten zwischen Server und Client können und sollen auch zwischengespeichert werden. Dies wird durch das Label *cacheable* und *non-cacheable* erreicht. Caching hat den Vorteil, dass Ergebnisse von Requests die regelmäßig abgesetzt werden, von einer Middleware zwischengespeichert werden und somit die Möglichkeit bietet Server und Netzwerk dauerhaft zu entlasten. Dies wird mit Hilfe des *Stateless-Constraints* ermöglicht [1, S. 79–81].

Interface / Uniform Contract

Mit der Definition von einer REST-Schnittstelle soll die Erreichbarkeit und die Weiterverwertbarkeit von Services sichergestellt werden. Services sollen funktions- und medientyp-basierend auf dem HTTP-Protokoll implementiert werden. Dies ermöglicht es, ein generisches und extrem starkes standardisiertes Interface zur Verfügung zu stellen.

Die Client-Logik soll vom Server abgekapselt sein und kann alle mögli-

chen Ressourcen vom Server verarbeiten. Der Service kann auch Links auf weitere Ressourcen oder Services zur Verfügung stellen [1, S. 81–82].

Layered System

In einigen Fällen kann es vorkommen, dass die Architektur einer Applikation auf mehrere Schichten aufgeteilt wird, sprich ein Client kann wieder als Service für einen anderen Client dienen und so weiter. Hier ist es besonders wichtig, um REST-konform zu bleiben, dass keiner dieser Schichten über den eigenen Tellerrand sieht und somit nicht über vielleicht verbundene Services oder Clients Bescheid weiß [19].

2.1.2 Design-Goals

Mit den Punkten die in den REST-Constraints besprochenen wurden, ergeben sich einige Codedesign-Ziele für den Entwickler, die beachtet werden sollen, um den Gedanken von REST zu erfüllen und die Schnittstelle richtig zu formen [19].

Leistungsfähigkeit

Im Rahmen einer dezentralen Architektur sind Leistungsfähigkeit und Zuverlässigkeit wichtige Eckpfeiler. Früher wurde eine Architektur nur auf einzelnen Servern gehostet und eine Optimierung der Aufrufe war notwendig, um den Server nicht zu überlasten. Diese Verbesserungen sind auch heute extrem wichtig, da die zu übertragenen Datenmengen rasant wachsen. Daher sollte sich der Entwickler Gedanken machen, wie er im Stande ist den Overhead eines Aufrufes und somit den Payload zu optimieren, ohne die Interaktionen zu komplex zu gestalten und dennoch jeden Aufruf so simpel wie möglich und unabhängig von anderen zu strukturieren [19].

Skalierbarkeit

Skalierbarkeit ist wichtig um eine große Anzahl an Instanzen bzw. Anfragen abzuarbeiten. Skalierbarkeit ist in mehreren Dimensionen möglich:

- **Scaling up** – Erhöhen der Kapazitäten der Services, Clients und Netzwerk-Devices.
- **Scaling out** – Verteilen des Workloads über die verschiedenen Server und Programme.
- **Smoothing out** – Wenn möglich, verteilen der Abfragen auf Zeiten wo der Server nicht ausgelastet ist um den Server bei Topzeiten zu entlasten.
- **Decoupling the consumption of finite resources** – Caching von Anfragen die öfter mit den selben Parametern abgesetzt werden, um so die Logik und die Datenbasis zu entlasten.

Ebenfalls ist es möglich den Workload mit Hilfe des Schichtensystems auf mehrere Servereinheiten zu verteilen [19].

Simplizität
Ebenso wichtig für dezentrale Architekturen ist es die Komplexität zu senken indem die Funktionen so geteilt werden, dass sie leicht unterscheidbar und zu verstehen sind. Jede Funktion sollte seine eigenen einzigartigen Fähigkeiten haben und über ein generisches Interface verfügbar sein [19].

Modifizierbarkeit
Im Laufe der Zeit ändern sich meist Anforderungen und Services müssen sich daher leicht ändern und erweitern lassen. Modifizierbarkeit ist vor allem wichtig für größere Services die im laufenden Betrieb erweitert werden müssen. Daher sollten generell bei der Entwicklung folgende Punkte in Betracht gezogen werden [19].

- **Evolvierbarkeit** – Die Fähigkeit von Refactoring und Redeploying eines Services ohne den Rest der Architektur zu beeinflussen.
- **Erweiterbarkeit** – Die Möglichkeit neue Funktionalitäten zu einer Architektur hinzuzufügen.
- **Anpassbarkeit** – Die Möglichkeit vorübergehend Funktionen für spezielle Aufgaben anzupassen.
- **Konfigurierbarkeit** – Die Möglichkeit das Verhalten von Funktionen dauerhaft zu verändern ohne die Implementierung adaptieren zu müssen.
- **Wiederverwendbarkeit** – Die Möglichkeit Funktionen ohne Aufwand für andere Applikationen wieder zu verwenden.

Portabilität
Beschreibt, wenn die Services und Lösungen leicht von einem auf den anderen Server portiert werden können [19].

2.2 Cross Domain REST

In vielen Fällen wird REST verwendet, um die Kommunikation zwischen einem JavaScript-Frontend und einem Server mit Hilfe von sogenannten *AJAX (Asychronous JavaScript and XML)* herzustellen. In Fällen wo sich JavaScript-Applikation und Backend nicht am selben Host befinden, kann die Verbindung aufgrund der *Same-Origin-Policy*, kurz SOP, nicht hergestellt werden. Die SOP ist eine Sicherheitsvorkehrung von Webbrowsern, die clientseitige Scripts, wie JavaScript oder ActionScript daran bindet, nur innerhalb des eigenen Ursprungs-Hosts zu kommunizieren. Daher wird in Fällen in denen die Webapplikation versucht Daten von einem anderen Host abzugreifen, geblockt. Um dieses Problem zu umgehen gibt es in der Regel zwei gängige Wege, Standard CORS und den Weg von JSONP. In der

Vergangenheit wurden auch noch andere Methoden wie iFrames und Browser Plugins eingesetzt. Da diese bereits sehr veraltet oder umständlich zu Verwalten sind, werden sie meist nicht mehr verwendet [2, 20, S. 1].

2.2.1 CORS

Cross-Origin-Resource-Sharing ist ein Mechanismus der es ermöglicht sogenannte *Cross-Origin-Requests* via *AJAX* abzusetzen, die im Normalfall von der *Same-Origin-Policy* geblockt werden. Diese Technik verwendet HTTP-Headers auf der Serverseite [16].

Hier werden Fälle definiert die für Cross-Domain-REST-Anfragen erlaubt sind, alle anderen werden automatisch geblockt. Dies wird mit dem nachfolgenden Kommando in der Serverimplementierung realisiert.

```
Access-Control-Allow-Origin: http://www.theDomain.me
```

Es wird dem Entwickler ebenfalls eine Reihe Features, wie etwa Events, zur Verfügung gestellt. Im Normalfall wenn CORS aktiviert ist, sind im Standard die HTTP-Methoden HEAD, GET und POST zugelassen, um jedoch PUT und DELETE verwenden zu können, muss der *Authorization-Header* erweitert werden [16].

```
Access-Control-Allow-Methods: GET, POST, DELETE, PUT
```

CORS wird mit Hilfe einer neuen erweiterten *XMLHttpRequest*-Funktion im Rahmen des HTML5-Standards auf der Client-Seite zur Verfügung gestellt. Im Internet Explorer wird die Funktion *XDomainRequest* genannt, in den übrigen Browsern Firefox, Chrome und Safari *XMLHttpRequest*. Diese beiden Funktionen erweitern die alt bewährte *XMLHttpRequest*, die für AJAX-Aufrufe verwendet wird und sind in der Lage CORS-Anfragen durchzuführen.

Ein weiterer Vorteil dieser Methoden sind die integrierten Möglichkeiten für die Authentifizierung mit Parametern die mitgesendet werden können. Der einzige Nachteil der sich mit CORS ergibt, ist das nicht alle Content-Types unterstützt werden, erlaubte Typen sind *application/x-www-form-urlencoded*, *multipart/form-data* oder *text/plain* [16].

2.2.2 JSONP

JSON with Padding ist eine sehr verlässliche Möglichkeit eine Verbindung zwischen einer Client-Applikation und einem Host herzustellen [21]. Indem es, mit Hilfe von in Callback-Funktionen verkapselte Daten auf der Serverseite die SOP umgehen kann.

Im Normalfall wenn der Browser die Anfrage mit Hilfe der *XMLHttpRequest*-Methode tätigt, returniert der Server bei Erfolg eine Antwort in Form von JSON oder XML an den Browser. Dies würde ohne extra Vorkehrungen auf beiden Seiten von der SOP blockiert werden.

Um die Kommunikation erfolgreich abwickeln zu können, müssen alle Daten auf der Client-Seite URL-encoded werden und zur Anfrage-URL angehängt werden, da nur reine GET-Anfragen vom Server zugelassen werden und es sich um einen simplen Funktionsaufruf handelt, werden alle Body-Elemente geblockt.

Wird die Anfrage erfolgreich am Server abgearbeitet und es wird kein JSONP verwendet, würde der Browser das Ergebnis als normalen Block interpretieren und den Transfer blocken. Um diesen Vorgang zu umgehen, wird das JSONP-Pattern herangezogen und das Ergebnis in eine Callback-Funktion gepackt, die so aufgebaut ist, dass die Daten als Rückgabewert zurück gegeben werden.

Der Aufruf erfolgt nicht mehr über XMLHttpRequest, sondern über ein dynamisch erzeugtes Script-Element, das nicht der SOP unterliegt. JSONP ist jedoch nicht auf JSON limitiert, es ist ebenfalls möglich Datenobjekte im Form von Javascript, Variablen oder anderen Datenformen zurückzugeben, der Standard ist jedoch trotzdem JSON.

Dieser Weg der Datenabfrage bringt dennoch einige Nachteile mit sich, der Größte ist das lediglich nur GET-Anfragen zulässig sind und alle Daten die es vom Client zu übermitteln gilt, müssen in die URL codiert werden. Alle POST und PUT Anfragen und deren Raw-Bodies werden weiterhin geblockt [21].

2.3 Promise

Promises beschreiben ein eventuelles Ergebnis als ein Teil einer asynchronen Operation [18]. Der primäre Weg für die Interaktion mit Promises ist durch die *.then* Methode, diese registriert Callbacks, die Werte zurückliefert wenn die dazugehörige Abfrage erfolgreich war oder der Grund warum das Promise nicht erfüllt werden kann.

Promises basieren auf dem *Future*-Design-Pattern [18]. Futures oder auch Versprechen gelten als Platzhalter für Ergebnisse die noch nicht bekannt sind, weil sie zum Zeitpunkt des Aufrufs nicht existieren. Der Vorteil von Promises besteht darin, dass Futures als Argumente an andere Prozeduraufrufe weitergereicht werden können und somit die Logik die Abarbeitung beginnen kann, bevor das ganze Promise verfügbar ist, somit ist es möglich Parallelismen zu maximieren.

Die in dieser Arbeit verwendete Implementation von Promises ist die *PromiseA+*-Variante [18]. Dieser in JavaScript realisierte und verwendete Standard ist anwendbar in den meisten JavaScript Frameworks, wie Ember.js [1] oder Angular.js [2]. Die Spezifikation wird als sehr stabil gesehen, da sie hauptsächlich die *.then*-Methode verwendet, um eine gemeinsame Ba-

[1] http://emberjs.com/
[2] https://angularjs.org/

sis zu bieten um alle Promises/A+ konformen Promises zu verbinden. Der Schwerpunkt der Spezifikation liegt nicht an der Realisierung der *Create*, *Fulfill* oder *Reject*-Promises, sondern am Fokus einer interoperativen *.then*-Funktion.

Die Terminologie von Promises wird wie folgt beschrieben. Ein Promise ist ein Objekt oder eine Funktion in JavaScript mit einer *.then*-Methode. Ein Objekt ist *thenable* wenn es eine *.then*-Methode besitzt. Ein *Value* ist jeder erlaubter JavaScript-Wert, ebenso die Werte *undefined*, *thenable* oder ein *Promise*. Eine *Exception* ist ein Wert, der im Falle eines Fehlers vom *throw*-Statement geworfen wird. Ein *Reason* ist ein Wert der zeigt das ein Promise abgelehnt wurde [18].

2.3.1 Einfache Verwendung von Promises

Die Verwendung von Promises muss von zwei Seiten betrachtet werden. Einerseits die Seite auf der das Promise seine Bedingungen und Verhalten definiert und andererseits die Seite des Aufrufs die dann zum Beispiel mit *.then*-Methode aufgerufen werden kann (Implementierung nach [18]).

```
1 //Defining Promise
2 var prom = new Promise(function(fullfill, reject){
3   var dataFetch = doFetch();
4   if(dataFetch.success){
5     fullfill(dataFetch.data);
6   }
7   reject(dataFetch.errorMessage);
8 });
```

new Promise
: Hiermit wird eine neue Promise-Instanz erstellt, die unabhängig von den vorangegangenen Versprechen ist.

fullfill
: Wird verwendet um den Erfolg des Versprechens zu signalisieren, der im Aufruf übergebene Parameter kann dann in der *.then* abgefragt und weiterverwendet werden.

reject
: Übermittelt nach außen die Meldung, dass das Ergebnis des Versprechens negativ ausgefallen ist, auch hier ist es möglich einen Parameter, in der Fehlernachricht nach außen zu übertragen.

Ist das Promise definiert, kann es nun entsprechend verwendet werden.

```
1 //Using Promise
2 prom.then(mySuccess, myError);
3
4 function mySuccess(data){
5   // Do something with Data returned
6 }
7
```

```
 8 function myError(data){
 9   //Do Error Handling
10 }
```

Wie es das Beispiel zeigt, müssen auch hier, wie bei vielen anderen asynchronen Funktionsaufrufen, Callback-Funktionen definiert werden. Im Fall des Beispiels repräsentiert *mySuccess* den Fall des Erfolgs und *myError* den Fehlerfall.

Kapitel 3

State of the Art

Im nachfolgenden Kapitel werden vorhandene Technologien für die Abhandlung von Transaktionen in Verbindung mit RESTful-Services beschrieben. Transaktionen kommen ursprünglich aus dem Bereich der Datenbanken und sollen sicherstellen, dass alle Aktionen die gegen eine Datenquelle abgegeben werden, in einem kohärenten und verlässlichen Weg, unabhängig von anderen Transaktionen stattfinden. Transaktionen werden unter anderen in transaktionalen Datenbanken, Objektdatenbanken, dezentralisierten Transaktionen und transaktionalen Dateisystemen verwendet. Transaktionen sind somit für das Verwalten der Zustände verantwortlich. REST-Transfers sind wie im Abschnitt 2.1 beschrieben zustandslos und voneinander unabhängig und somit konträr zu Transaktionen.

Um diese Tatsache besser erläutern zu können, wird folgendes Beispiel herangezogen. Es beschreibt eine Web-Applikation zuständig für die Sitzplatzreservierung von Flügen, wie es bei vielen Online-Reiseanbietern üblich ist. Mit dieser Anwendung können Flüge und Transfers von einem oder mehreren Anbietern, die in verschiedenen Fällen jeweils ihre eigenen Webservices besitzen können, gebucht werden.

Im Falle des Beispiels würde das bedeuten, dass ein Kunde einen Flug *F1* von Linz nach Wien und einen Anschlussflug *F2* von Wien nach Dubai bucht, genauso wie einen anschließenden Flughafentransfer *T1* vom Flughafen in Dubai zum Hotel wie es Abbildung 3.1 illustriert.

Diese Buchungen klingen in der Theorie ziemlich simpel, doch müssen auch Fälle behandelt werden, in denen eine der Anfragen zu einer Airline, also einem Webservice, fehlschlägt. Um diese Situationen behandeln zu können gibt es verschiedene Lösungsansätze die nachfolgend erläutert werden.

Der Anfang dieses Kapitels wird den Begriff von *Atomaren Transaktionen* und deren Richtlinien behandeln, anschließend im zweiten Teil des Kapitels werden Lösungen beschrieben, in denen Transaktionen hauptsächlich auf der Server-Seite abgehandelt werden. Der dritte Teil beschäftigt sich mit Möglichkeiten, in denen Transaktionen in Zukunft auf der Client-Seite

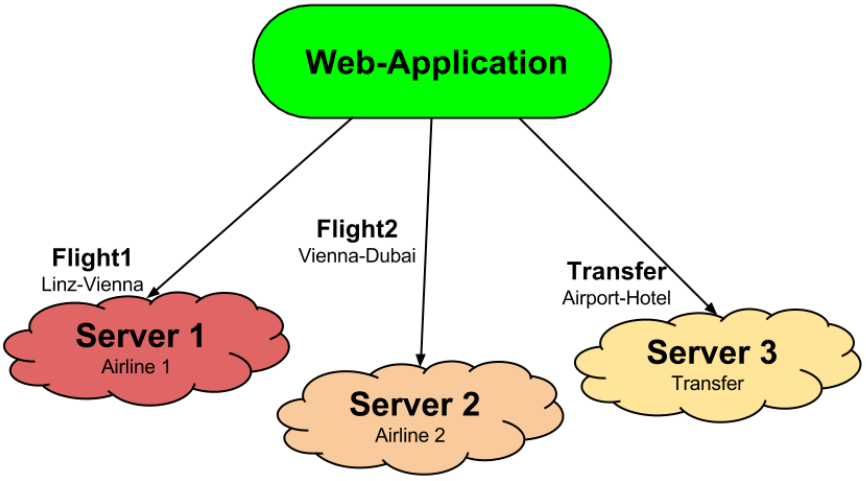

Abbildung 3.1: Übersicht Beispiel Reiseplattform.

verwaltet werden können, somit die Server nur entsprechende Webservices zur Verfügung stellen müssen und dem Server schlussendlich Rechenlast abgenommen wird.

3.1 Atomic Transactions

Unter *Atomic Transactions* versteht man Transaktionen die in kleinere Transaktionen als eigenständige und isolierte Arbeitseinheiten aufgeteilt werden [22]. Diese Einheiten verwenden ein eigenes *Scope*-Konstrukt in dem die Variablen mit transaktionaler Semantik abgearbeitet werden. Diese Einheiten arbeiten mit einem Zustand (State) der ebenfalls abgekapselt von anderen atomaren Transaktionen ist. Atomare Transaktionen garantieren das jedes partielle Update automatisch zurückgerollt wird, sollte während dem Update eine *Exception* geworfen werden [17]. Das bedeutet das alle eingeworfenen Updates wieder auf den Stand zum Beginn der Transaktion rückgängig gemacht werden. Diese Transaktionen sind generell sehr kurzlebig und besitzen die vier ACID-Attribute.

Die ACID-Attribute sind generell wichtig für diese Arbeit und bilden die Basis für Transaktionen. Die ACID-Attribute wurden in den späten 1970er Jahren von Jim Gray für verlässliche Transaktionssysteme aufgestellt. Das Acronym *ACID* wurde dann von Andreas Reuter und Theo Härder 1983 geformt [22].

Atomicity – Atomarität

Eine Transaktion representiert ein Arbeitspaket, entweder jede oder

keine Änderung in diesem Paket wird ausgeführt [17, 23].

Consistency – Konsistenz
Wenn die Transaktion commited wird, muss die Datenintegrität gewährleistet werden. Wenn eine Transaktion Daten in einer Datenbank die intern konsistent sind manipuliert, bevor die Transaktion gestartet wurde und müssen die Daten auch noch am Ende der Transaktion konsistent sein. Die Sicherstellung der Datenintegrität ist hauptsächlich die Verantwortung des Entwicklers [17, 23].

Isolation – Isolierung
Änderungen innerhalb einer Transaktion müssen isoliert sein und dürfen nicht von anderen Transaktionen beeinflusst werden. Isolierte Transaktionen müssen ihre Änderungen so der Datenbasis anfügen, dass die interne Datenintegrität gewährleistet ist, als ob die Transaktionen seriell ausgeführt werden [17, 23].

Durability – Langlebigkeit
Wenn die Transaktion commitet ist, sind alle Änderungen permanent der Datenbasis hinzugefügt. Die Änderungen müssen ebenfalls bestehen wenn ein Systemfehler auftritt [17, 23].

Atomic Transactions können auch mit einem Timeout ausgestattet werden, dies macht Sinn wenn Ressourcen nicht zu lange blockiert werden sollen. Ebenso sinnvoll ist es, Retries einzubauen, um eventuelle Fehler abzufangen, die durch Serverprobleme oder geblockten Ressourcen durch andere Transaktionen entstehen können [17, 22].

3.2 Serverseitige Lösungen

Um einen guten Überblick über die Wichtigkeit für eine clientseitig verwaltete REST-Transaktionen darzustellen, gibt der nachfolgende Teil einen kurzen Überblick über die gängigsten Server-Implementierungen. Generell gibt es ca. fünf verschiedene Varianten die sich im Laufe der Zeit auf diesem Gebiet etabliert haben, wie es in der nachfolgenden Tabelle 3.1 ersichtlich ist und aus [7] entnommen wurde.

3.2.1 Batched Transactions mit überladenen POST

Eine viel verwendete Methode um REST-Transaktionen zu Verwalten ist ein Überladen der POST-Operation. Das Überladen wird mit Hilfe einer *Transaction-Factory*, die mit Hilfe von speziellen URL-Patterns ausgelöst werden können, erreicht. Alle nachfolgenden POST, PUT oder DELETE die dann über das Pattern laufen sind ein Bestandteil der Transaktion. Während die Transaktion aktiv ist, werden alle eingehenden Änderungen serialisiert, damit sie dann einfach am Ende durch den Commit der Datenbasis übergeben werden können [3, S. 1-2].

Tabelle 3.1: Übersicht vorhandener Lösungen [7]

Jahr	Methode
2000	Batched Transactions mit überladenen POST
2007	Transaktionen als Quelle
2009	Optimistische Technik für REST verwendende Transaktionen
2009	RETRO - Konsistentes und wiederherstellbares RESTful-Transaktions-Model
2010	Zeitstempel basiertes zwei-Phasen Protokoll für RESTful-Transaktionen

GET-Operationen würden natürlich Echtzeitdaten liefern. Ein Read-Kommando (über die Factory) während der Transaktion würde den Datenstand von der Transaktion zurückgeben und somit eine Referenzpunkt für spätere Vergleiche zur Verfügung stehen. Würden Werte geändert werden bevor die Transaktion beginnt, würde die Transaktion abgebrochen und eine neue gestartet werden. Dieser Mechanismus ist ähnlich der *Lock-Free*-Datenstruktur [3, S. 3].

Lock-Free erlauben multiple Threads um Updates durchzuführen ohne die Hilfe von Sperren wie *Mutexes* oder *Spinlocks* in Anspruch zu nehmen. Grundlegend garantiert der Mechanismus, dass Lese-, Update- und Schreibe-Versuche entweder fehlschlagen oder erfolgreich abgewickelt in *transactional*-Sinne.
Das Ende der Transaktion, also der Commit, wird durch einen *commit=true* Parameter mit Hilfe der PUT HTTP-Methode übertragen. Der Server ist dann für die entsprechende Abwicklung der Transaktion zuständig [3, 7, S. 2–4].

3.2.2 Transaktionen als Quelle

Die Ressourcen orientierte Architektur bedient sich an der REST-Struktur in der jede eingehende HTTP-Request eine Ressource als Ziel besitzt. Dies wird für die Datenkonsistenz herangezogen um ähnliche Ergebnisse zu erzielen wie es normaler Weise die Aufgabe einer Transaktion auf einem Server ist. Hierfür werden die Transaktionen selbst als Ressource herangezogen [15].

Um die Funktionsweise besser erklären zu können wird als Beispiel eine Umbuchung von Konto A auf Konto B verwendet. Im Normalfall würden die Konten mit */accounts /A11* und */account /B11* erreichbar sein. Um die Transaktion zu Starten wird eine Transaction-Factory verwendet, diese

startet die Transaktion in diesem Fall mit */transactions /account-transfer*, und erstellt somit die Ressource und das Ergebnis könnte wie folgt aussehen */transactions /account-transfer /A11T1* [15].

Als nächstes können die Kontostände mit den entsprechenden PUT Befehlen mit */transactions /account-transfer /A11T1 /accounts /A11* und */transactions /account-transfer /A11T1 /accounts /B11* abgeändert werden, die neuen Kontostände werden mit Hilfe eines Parameters übertragen. Die Transaktion kann jeder Zeit mit Hilfe des DELETE HTTP-Kommandos auf der URL */transactions /account-transfer /A11T1* zurück gerollt werden und es wird sichergestellt das keine Inkonsistenzen auftreten [15].

Bei jedem Commit muss geprüft werden, dass keine Daten verloren gehen, im Falle des Beispiels natürlich das kein Geld zerstört wird. Um den Commit durchzuführen wird der HTTP-PUT auf */transactions /account-transfer /A11T1* verwendet und der Service sollte die Änderungen zurückgeben. Der Vorteil an dieser Variante ist, dass gleichzeitig aufgrund der Ressource ein Log erstellt wird und somit jede Aktion zurückverfolgbar ist [9, S. 231–232].

3.2.3 Optimistische Technik für REST verwendende Transaktionen

Dieser Ansatz versucht eine REST-Transaktionsunterstützung für Datenkonsistenz und Integrität ohne dem Hinzufügen komplexer Datenstrukturen zu erreichen. Die Technik sieht vor einen Algorithmus ohne Datensatzsperren zu verwenden um somit den technischen Aufwand zu verringern [11, S. 1]. Diese Lösung baut auf dem vorangegangenen Ansatz, *Transaktionen als Quelle* auf. Mit der Basis kann der Client eine Transaktion mit einer POST-Anfrage an eine Transaction-Factory erstellen. Die Antwort dieser Anfrage liefert eine URL zurück, die auf die erstellte Ressource zeigt. Nun kann der Client alle Daten absetzten die in der erstellten Transaktion verwaltet werden müssen. In dieser Zeit ist es jederzeit möglich einen Rollback mit einem HTTP-DELETE an die übertragene URL zu erzielen. Wenn der Client alle relevanten Daten abgesetzt hat, kann die Transaktion commited werden. Der Server muss in dieser Zeit sicherstellen das alle Daten konsistent gehalten werden [11, S. 2].

Wie schon erwähnt versucht diese Methode ohne Datensperren auszukommen, dafür wird davon ausgegangen, dass keine Probleme während der Transaktion auftreten, im Gegenzug dafür ist diese Variante Deadlockfree. Anhand der Annahme, dass keine Fehler auftreten, wird sie als optimistisch bezeichnet [11, S. 2-3].

Dafür werden die nachfolgenden Regeln aufgesetzt. Eine Leseanfrage kann nicht zu einem Verlust der Datenintegrität führen und besitzt daher keine Einschränkung. Schreiboperationen sind relativ eingeschränkt, indem jede Transaktion aus einem Zwei- oder Drei-Phasenprotokoll besteht. Dieses

Protokoll besteht aus einer Lese-, Validierungs- und möglichen Schreibphase. Während der Lesephase werden auf der Client-Seite Sicherungen abgelegt, um auf diese die gewünschten Änderungen anwenden zu können und während der Validierungsphase sicherstellen zu können, dass keine Datenverluste oder Inkonsistenzen auftreten, sind die Validierungen erfolgreich, werden die lokalen Änderungen auf die globale Datenbasis kopiert [11, S. 3].

3.2.4 RETRO – Konsistentes und wiederherstellbares RESTful-Transaktions-Model

Eine weitere serverseitige Möglichkeit für die Verwaltung von Transaktionen ist das *RETRO*-Model. Dieser Mechanismus erfüllt alle ACID-Eigenschaften und arbeitet komplett innerhalb von REST mit HTTP.

Für die Realisierung operiert das System Ressourcen, die über eine homogene Schnittstelle manipuliert werden können. Um Zustandslosigkeit zu garantieren, werden die Ressourcen seriell gesperrt und Kopien werden für Änderungen innerhalb der Transaktion erstellt. Der Client manipuliert dann nur die Kopien und ist somit von der Datenbasis abgekapselt [6, S. 3].

Jede Ressource, die im Rahmen der Manipulation erstellt wird, kann ebenfalls innerhalb des Transaktionsbereichs generiert werden und ist mit Hyperlinks von der manipulierten Ressource erreichbar. Die Anfragen werden als Ressource gespeichert und werden ein Teil der History. Beim Speichern der Transaktion, wird die Ressource und ein Teil der Datenbasis kopiert [6, S. 3].

Um im *RETRO*-Modell eine neue Transaktion anzulegen, muss die entsprechende Representation mit POST an den Service gesendet werden. Der Server liefert dann einen *Code 201 Created* und eine URL zur neuen Transaktions-Ressource zurück. Eine Transaktion mit *RETRO* kann über *GET, PUT und DELETE* modifiziert werden und beinhaltet eine URL des Besitzers, einen Link zu den aktuellen Sperren, einen Link zur History der Transaktion, eine Commit-Url für den Abschluss und eine, die bei Aufruf den aktuellen Status der Transaktion zurück gibt [6, S. 3–5].

3.3 Client-Seitige Lösung

Die nächsten zwei Lösungen die in diesem Kapitel behandelt werden, besitzen ihren Transaktions-Koordinator auf der Clientseite und sind daher Lösungen die es ermöglichen REST-Transaktionen über dezentralisierte Webservices abzuwickeln.

3.3.1 Zeitstempel basiertes Zwei-Phasen-Protokoll für RESTful-Transaktionen

Das zeitstempel-basierte Protokoll für REST-Services sieht einen speziellen Algorithmus für den Support eines Transaktionsverwaltungs-Mechanismus vor. Dieser nimmt wie schon in Lösungen zuvor, Gebrauch am Überschreiben der verschiedenen HTTP-Methoden, die auf verschiedene REST-Ressourcen zugreifen. Hierfür soll der HTTP-GET weiterhin für das Auslesen der Ressourcen aus der Datenbasis verwendet werden. Für das Anlegen eines neuen Datensatzes hingegen wird in dieser Variante anstatt einem POST ein überschriebenes PUT herangezogen. PUT wird bei dieser Lösung, ebenso wie im Standard-REST, für Updates am Datensatz verwendet. Für das Entfernen von Daten bleibt dem REST-Standard entsprechend das DELETE-Kommando erhalten [10, S. 1–3].

Soll nun eine neue Transaktion gestartet werden, sendet der Client als erstes einen sogenannten *Prewrite* an alle für die Transaktion relevanten Webservices aus. Anhand der zurückkommenden Antworten wird dann wiederum ein *Write*-Kommando abgesetzt, dieses beinhaltet entweder einen *Commit* oder *Abort*-Befehl, der an die Webservices gesendet wird [10, S. 3].

Der Nachteil an diesem System besteht darin, dass wenn ein *Prewrite* an den Server abgesetzt wurde, dieses nicht mehr aktualisierbar ist. Der einzige Weg um somit ein *Prewrite* zu ändern, besteht darin, dieses zu Löschen und dann die Änderungen in einem neuen abzusetzen [10, S. 3].

Natürlich kann es geschehen, dass in dieser Zeit andere Requests auf die gleiche Ressource zugreifen wollen. Um auch in diesen Fällen konstistent zu bleiben, wird ein Zeitstempelsystem hinzugefügt. Dieses System beschreibt einen Satz an Zeitstempelregeln die eingehalten werden müssen, um einen Datensatz erfolgreich in die Datenbasis aufnehmen zu können. Um dieses System unterstützen zu können müssen die Datensourcen mit Felder für die Zeitstempelverwaltung erweitert werden. Anhand der Struktur kann jedoch der Tranaktionmanager am REST-Client angesiedelt werden und ermöglicht somit die Verwaltung über dezentralisierte Webservices [5, S. 1–3].

3.3.2 Try-Cancel-Confirm-Pattern

Das *Try-Cancel-Confirm-Pattern* kurz TCC ist ein Design-Pattern das großes Potential für die Realisierung von *RESTful-Transactions* besitzt. Hierbei ist es wichtig, dass die verwendete Implementierung sicher und verlässlich ist, vor allem auch bei lang laufenden Transaktionen die Datenintegrität sicherstellt.

Um das TCC-Pattern in diesem Punkt besser erklären zu können, wird das schon vorher verwendete Beispiel der Sitzplatzreservierung herangezogen und der Ablauf des Patterns in der Abbildung 3.2 dargestellt.

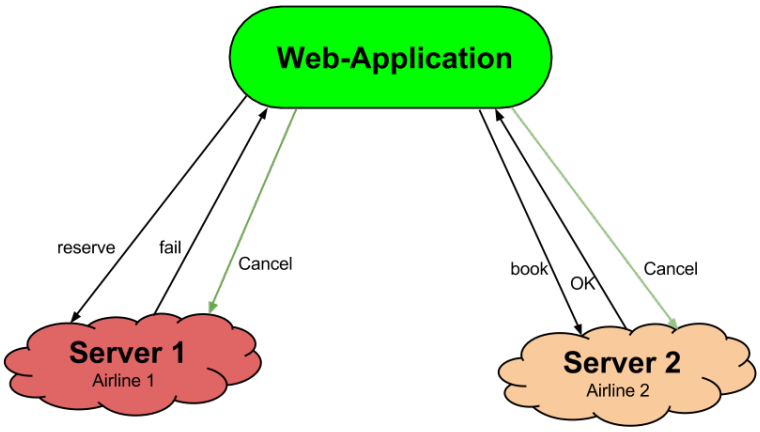

Abbildung 3.2: Übersicht Beispiel Reiseplattform.

Der grundlegende Ablauf des Try-Cancel-Confirm-Patterns sieht in diesem Fall vor, dass Sitzplätze nach und nach bei den verschiedenen Airlines reserviert werden, ist jede Reservierung erfolgreich, je nach Implementierung die Transaktion erfolgreich beendet oder durch Absetzen eines zusätzlichen Confirm-Commands erfolgreich abgeschlossen. Komplizierter werden jedoch Fälle in denen eine oder mehrere Reservierungen fehlschlagen und dann die weitere Handhabung mit den bereits abgeschlossenen Reservierungen gehandhabt wird.

Eine Möglichkeit wäre die vorhandenen Reservierungen wieder zu stornieren oder noch einmal versuchen die Fehlgeschlagenen abzusetzen oder einen alternativen Flug zu finden und zu buchen. Für die Handhabung dieser Transaktionen wird hierfür ein Transaction-Service in Form einer Workflow-Engine herangezogen, der auf dem Try-Cancel-Confirm-Pattern aufbaut. Der Service ist dafür verantwortlich die Buchungen zu Verwalten und den aktuellen Status der Anfragen auf jedem einzelnen Servern zu protokollieren [13].

Ein weiterer notwendiger Schritt für die Realisierung des TCC-Patterns ist jede Anfrage als eigene Transaktion zu sehen die dem ACID-Konzept untersteht. Diese Transaktionen werden dann unter dem TCC-Pattern wieder in einer größeren Transaktion zusammengefasst.

ACID (Atomicity, Consistency, Isolation, Durability) beschreibt dass alle Daten, die während einer Transaktion verwendet werden, gesperrt sind und sich nicht ändern dürfen so lange bis die Transaktion Commited (Confirm) wird oder ein Rollback (Cancel) veranlasst wird [8, S. 3].

Der Ablauf des Workflows ist in drei grundlegende Phasen aufgeteilt. Die

erste Phase (TRY) repräsentiert die normalen Anfragen von der Business-Logik abgearbeitet wird, in diesem Fall die Sitzplatzreservierung.

Die zweite Phase ist der *CANCEL*-Mechanismus der ausgelöst wird, wenn eine oder mehrere Anfragen fehlschlagen, das Rollback soll und kann ebenfalls ausgelöst werden wenn ein Timeout erreicht wird. Generell soll jede gestartete *Mini*-Transaktion mit dem Status *PENDING* welcher jederzeit von der Workflowengine abgefragt werden kann, versehen werden. Service-Logik kann diese Stati auf Timeouts prüfen und abbrechen wenn die Workflowengine keinen CANCEL oder CONFIRM Befehl sendet. Die dritte und letzte Phase ist CONFIRM, diese wird an die Services ausgesendet um die Transaktion zu bestätigen und die Reservierungen zu fixieren [8, S. 4].

3.4 Vergleich der Lösungen

Bei den beschrieben Lösungsansätzen muss in erster Linie zwischen server- und client-basierten Lösungen unterschieden werden. Bei den Serverlösungen befindet sich die Transaktionsverwaltung auf der Serverseite und bei den Client-Lösungen werden diese im Browser abgehandelt.

Serverlösungen sind meistens nur auf einem Webservice beschränkt und können daher keine dezentralen Architekturen unterstützen, jedoch meist besser die ACID-Eigenschaften für Transaktionen erfüllen und somit für eine bessere Datenintegrität garantieren. Jedoch kommt dieser Vorteil meist mit Einschränkungen da mit dem Überladungen von diversen HTTP-Methoden, wie zum Beispiel bei *Batched Transactions*, das volle REST-Spektrum nur begrenzt oder mit einem erhöhten Aufwand verfügbar ist.

Client-Lösungen stellen sich als etwas komplizierter heraus, da diese nicht auf die Verlässlichkeit und Ausfallsicherheit eines Servers setzten können und somit auf den Faktor User angewiesen sind. Jedoch unterstützt auch bei den hier aufgezeigten client-basierten Lösungen nur das TCC-Muster das volle REST-Spektrum. Aber auch diese Ansätze benötigen trotzdem noch Unterstützung vom Server um Commit und Rollback Befehle auszuführen und andere eventuelle Fehlerfälle wie Deadlock, Zeitüberschreitungen oder Abstürze der Transaktionskoordinatoren zu kompensieren.

Kapitel 4

Umsetzungsvarianten

Im vorangegangen Kapitel wurden bereits vorhandene Lösungen für REST-basierte Transaktionen erläutert. Dieses Kapitel beschäftigt sich mit den theoretischen Lösungsansätzen für die Realisierung REST-basierter Transaktionen. Das Hauptaugenmerk liegt hier bei Lösungen in denen die Transaktionslogik beziehungsweise die Workflow-Logik für die Abhandlung der Transaktionen auf der Client-Seite platziert ist. Die Platzierung der Logik am Client ermöglicht es Transaktionen über mehrere dezentrale Webservices abzuwickeln. Anhand der beschriebenen Lösung wirkt das TCC-Pattern als sehr zuverlässig und wird daher in diesem Kapitel näher beschrieben und im nachfolgenden implementiert. Zusätzlich wird noch ein Konzept für eine *ClientOnly*-Lösung in Form einer Workflow-Engine erläutert, die komplett ohne Server-Unterstützung auskommt.

4.1 Server-Lösung

Ein vielversprechender Ansatz für RESTful-Transaktionen ist das *Try-Cancel-Confirm-Pattern* basierend auf der Veröffentlichung von ATOMIKUS wie bereits näher im Abschnitt 3.3.2 beschrieben. In diesem Kapitel wird ein eigener Implementierungsansatz des Patterns in Form einer API (*Application-Programming-Interface*) erklärt. Die Aufgabe dieser API ist es eine Schnittstelle in JavaScript zur Verfügung zu Stellen die im Hintergrund die Fähigkeit besitzt, RESTfull-Calls zu Servern zu tätigen und diese in Form von Transaktionen verwalten zu können. Die Hauptaufgabe der Transaktionsverwaltung besteht darin den aktuellen Status der Anfragen zu Verwalten und im Fehlerfall dafür zu sorgen, dass die Datenintegrität über alle kontaktierten Webservices gewährleistet ist.

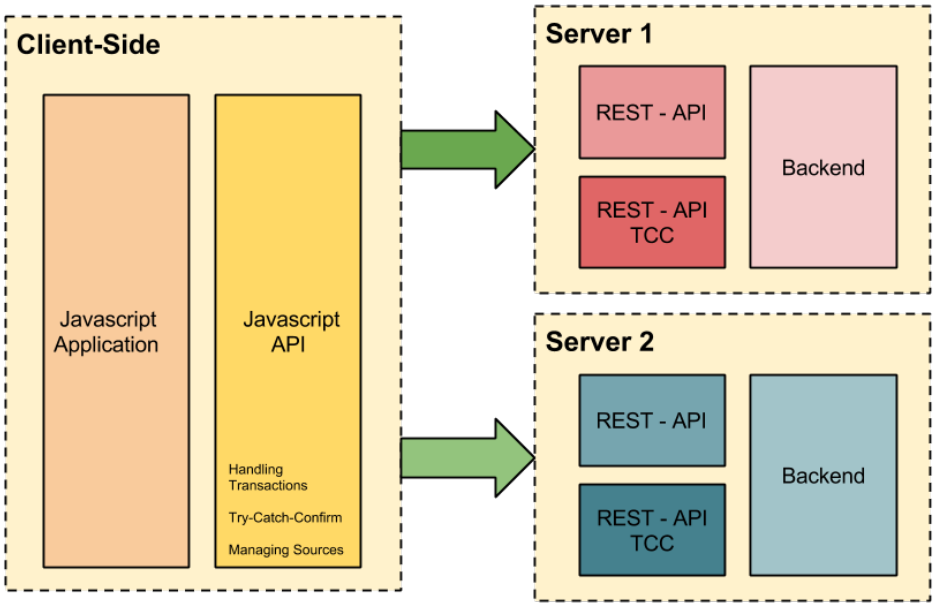

Abbildung 4.1: Grundarchitektur von Client und Server.

4.1.1 Architektur und Aufbau

Die grundlegende Architektur der API wird, wie in Abbildung 4.1 dargestellt, auf zwei Teile aufgeteilt, die Client- und die Backend-Seite.

Die Backend-Seite kann sich aus einer Sammlung von Webservices auf einander unabhängigen Systemen zusammensetzen. Diese Webservices stellen eine Standard-REST-Schnittstelle zur Verfügung, diese wird mit Logik erweitert, um *Cancel*- und *Confirm*-Befehle im Sinne des TCC-Patterns realisieren zu können.

Die Frontend-Seite beinhaltet die API, diese besitzt zwei wesentliche Komponenten, einen *TransactionHandler* und eine oder mehrere *TransactionFunctions*. In dieser Implementierung werden eine oder mehrere *TransactionFunctions* dem *TransactionHandler* zur Verwaltung übergeben. Dieser ist für die Fehlerbehandlung mit Hilfe von Mitprotokollieren aller Aktionen und das Verwalten der Resultate verantwortlich.

Die Verwendung der API soll durch das nachfolgende Beispiel veranschaulicht und die weitere Funktionsweise der API im Abschnitt 4.1.3 im Detail beschrieben werden.

```
1    //Start Transaction
2    var transHandler = new TccTransactionHandler();
3
4    //Set TCC Stuff
5    transHandler.setCancelUrl(urlTccCancel);
```

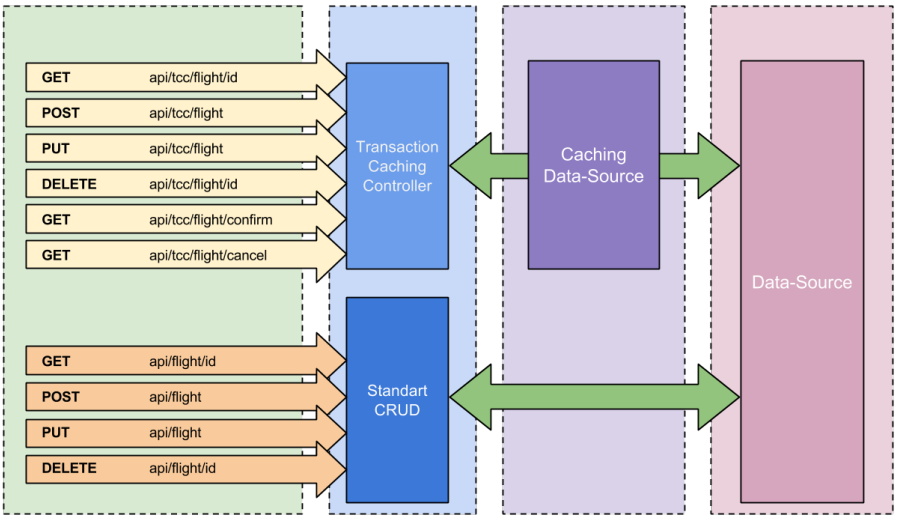

Abbildung 4.2: Grundarchitektur Server

```
6     transHandler.setConfirmUrl(urlTccConfirm);
7
8     //Add TransactionFunction to Handler
9     var f = new TransactionFunction("f1", urlFlights);
10    f.setTransactionHandler(transHandler);
11
12    //Do Function Call
13    var prom = f.doCall();
```

4.1.2 Backend

Die grundlegende Idee für die Backend-Implementierung besteht darin so wenig Abweichung wie möglich vom Standard-REST-Interface zu generieren. Die grundsätzliche Idee der Backend-Architektur ist in der Abbildung 4.2 illustriert und anhand des Beispiels in Kapitel 3 beschrieben. Wie bereits in der Einleitung erwähnt, soll der Server eine normalisierte Schnittstelle für *Create*, *Read*, *Update* und *Delete*-Befehle für RESTful-Requests zur Verfügung stellen, sodass Datenressourcen mit oder ohne Transaktionen manipuliert werden können.

Dafür werden zwei Interfaces angeboten, die erste Schnittstelle bietet die Möglichkeit alle Anfragen innerhalb einer Transaktion abzuwickeln, das Zweite bietet den REST-Standard sollte keine Transaktionsverwaltung benötigt werden. Das Standard-REST-Interface greift direkt auf die Datenbasis zu und ist strikt von den Daten in den Transaktionen getrennt.

Das erste Interface für Transaktionen besitzt einen Controller der die An-

fragen der JavaScript-API und somit von den *TransactionFunctions* entgegennimmt. Mit Hilfe eines spezifischen URL-Schemas und einer im Frontend generierten Transaktionsnummer, die auf diesen Controller zeigt, werden alle Anfragen die in einer Transaktion verwaltet werden, weitergeleitet. Der Controller ist dafür zuständig das er die übermittelten Daten validiert und anhand der Transaktions-ID überprüft, ob sie wirklich Teil einer Transaktion sind.

Um Datenobjekte innerhalb einer Transaktion zu verwalten, wird für jedes Objekt eine Caching-Tabelle parallel zur Datenbasis angelegt. In dieser Tabelle werden alle Entitäten innerhalb einer Transaktion gespeichert, ebenfalls wird die Caching-Tabelle um eine weitere Spalte für die Transaktions-ID erweitert, um die Einträge einer Transaktion eindeutig zuordnen zu können. Um auf bestehende Objekte referenzieren zu können, wird ein Fremdschlüssel auf den entsprechenden Eintrag der Datenbasis hinzugefügt und eine Spalte mit dem Typ (POST, UPDATE oder DELETE) eingefügt, um die Aktion festhalten zu können.

Um eine neue Entität zum Speichern zu übermitteln, wird diese im Body eines *HTTP-POST* im Sinne von CRUD an den Webservice übertragen. Die Transaktionsnummer wird in der URL als Parameter übergeben. Das Objekt wird dann schlussendlich in der entsprechende Caching-Tabelle abgespeichert.

Soll das Objekt manipuliert werden, wird mit Hilfe von *HTTP-UPDATE* das komplette geänderte Objekt transferiert, wird dieses wieder über das Transaktions-URL-Pattern und mit der Transaktions-ID übertragen, zieht das eine Änderung des entsprechenden Eintrags in der Caching-Tabelle mit sich. Soll innerhalb der Transaktion ein Objekt der Datenbasis geändert werden, wird eine Kopie in der Caching-Tabelle angelegt, die übermittelten Änderungen auf das Objekt angewandt und auf den Quell-Eintrag in der Datenbasis referenziert.

Um Einträge zu Löschen wird *HTTP-DELETE* herangezogen, hierbei wird der Aufruf an die URL mit der ID des zu löschenden Eintrags und der Transaktionsnummer getätigt. Ist der gewünschte Eintrag innerhalb der Transaktion verfügbar, wird dieser gelöscht, ist der zu löschende Eintrag nur in der Datenbasis erreichbar, wird eine entsprechende Referenz in der Caching-Tabelle gespeichert.

Um ein oder mehrere Objekte abfragen zu können, wird *HTTP-GET* verwendet. Auch bei der Datenabfrage gilt das die ID der Entity und die Transaktions-ID als Parameter mit übertragen werden. Ist das Objekt ein Teil der Transaktion oder der Datenbasis wird es zurückgegeben, ist es jedoch ein Teil einer anderen nicht abgeschlossenen Transaktion, kann das Objekt nicht zurückgeben werden.

Um nun Transaktionen beenden zu können, wird eine *Cancel*-Methode und eine *Confirm*-Methode im Sinne des *Try-Cancel-Confirm-Patterns* erstellt. Beide Methoden werden mit Hilfe eines *HTTP-GET*-Aufrufs ausge-

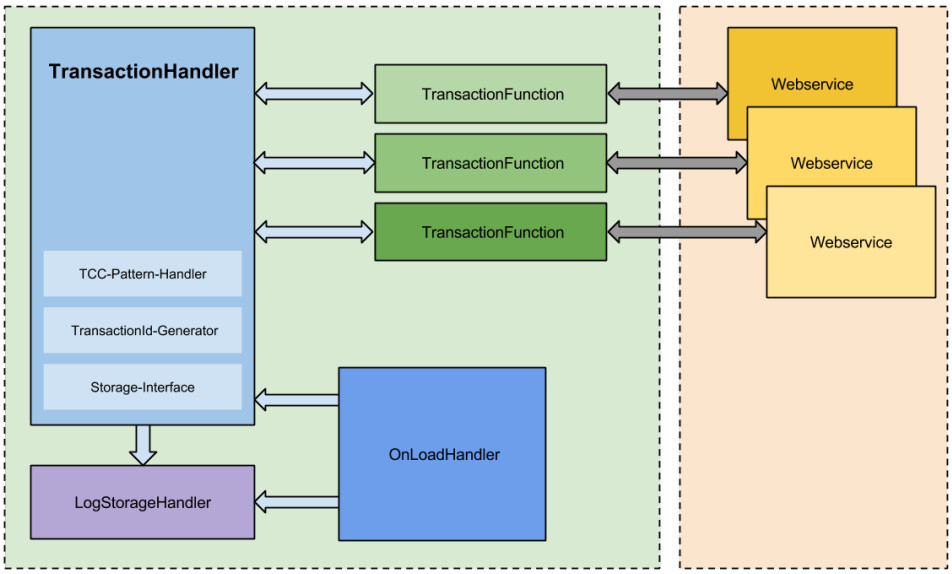

Abbildung 4.3: Grundarchitektur Frontend der TCC-Lösung.

löst, auch hier muss wieder die Transaktions-ID als Request-Parameter mitgesendet werden, um die Referenz zu der Transaktion herzustellen.

Wird der *Cancel*-Befehl aufgerufen werden alle Elemente mit der entsprechenden Transaktions-ID aus allen Caching-Tabellen gelöscht. Im Falle des *Confirm*-Befehls werden alle mit der Transaktion verbundenen Objekte und deren Änderungen in die entsprechende Tabelle in der Datenbasis übertragen.

4.1.3 Client-Seite

Die Client-Seite soll das Nervenzentrum des Transaktionsmanagement in dieser Implementation werden. Die Hauptaufgabe der API ist es eine Schnittstelle zu bieten die RESTfull-Calls an Webservices absetzen kann. Diese Aufrufe sollen dann auf der Client-Seite innerhalb einer Transaktion abgewickelt werden können. Die API soll in JavaScript implementiert werden und jQuery für die AJAX-Aufrufe die im Hintergrund die Requests an die Webservices senden, verwenden.

Die Struktur der API ist auf drei wichtige Teile aufgeteilt, die TransactionFunction, der TransactionHandler und der Logger. Die Abbildung 4.3 soll die Zusammenhänge der einzelnen Komponenten veranschaulichen.

TransactionFunction

Die *TransactionFunction* repräsentiert jeweils eine Ressource im Backend und kann alle REST-Operationen wie Create mit POST, Upda-

te mit UPDATE, Delete mit DELETE und Query mit GET ausführen. Die Funktion kann auch als alleinstehende Funktion ohne Transaktions-Handler verwendet werden. Soll die Funktion und deren Aufrufe ein Teil der Transaktion sein, muss der Funktion eine Handler-Instanz übergeben werden.

TransactionsHandler
Der *TransactionsHandler* protokolliert alle Aktionen der Funktionen mit Hilfe des TransactionLogger-Instanz die beim Instanzieren des Handlers erstellt wird. Der Handler ist ebenfalls dafür zuständig, dass nach Beendigung der Transaktion der entsprechende *Cancel-* oder *Confirm-*Befehl zu den Webservices gesendet wird. Dafür können entsprechende URLs generell im Handler oder in den einzelnen Funktionen definiert werden.

TransactionLogger
Der *TransactionLogger* ist die Ausfallssicherung für den *TransactionHandler*, mit Hilfe des Loggers ist es möglich im Falle eines Absturzes der Applikation, wenn noch nicht eingetroffene Anfragen vorhanden sind, diese noch Rückgängig zu machen und somit die Integrität der Daten zu gewährleisten.

4.2 Client-Only

Alle bisher behandelten Lösungen benötigen zusätzliche Implementierungen auf der Serverseite um eine Transaktionsunterstützung generieren zu können. Im Gegensatz zu den vorhandenen Serverlösungen soll eine ClientOnly-Lösung gefunden werden, die komplett ohne Serverunterstützung operieren kann. Die einzige vom Server benötigte Unterstützung ist somit die zur Verfügungstellung einer Standard-REST-Schnittstelle. Wenn dieses Kriterium erfüllt ist, kann eine gesamte client-basierte Lösung zur Transaktionsabwicklung realisiert werden.

Die Abbildung 4.4 soll die Architektur und Kommunikation zwischen Frontend und Backend veranschaulichen. Für die ClientOnly-Lösung wird eine API implementiert die Transaktionen mit Hilfe von speziellen Workflows abarbeiten und mögliche Fehlerfälle abzufangen. Der nachfolgende Code soll die geplante Verwendung der API veranschaulichen.

```
1 //Start Transaction
2 var transHandler = new TransactionHandler();
3
4 //Add TransactionFunction to Handler
5 var f1 = new TransactionFunction("f1", urlFlights);
6 f1.setTransactionHandler(transHandler);
7
8 //Do Function Call
9 var prom1 = f1.doCall();
10 prom1.then(onSuccess, onFail);
```

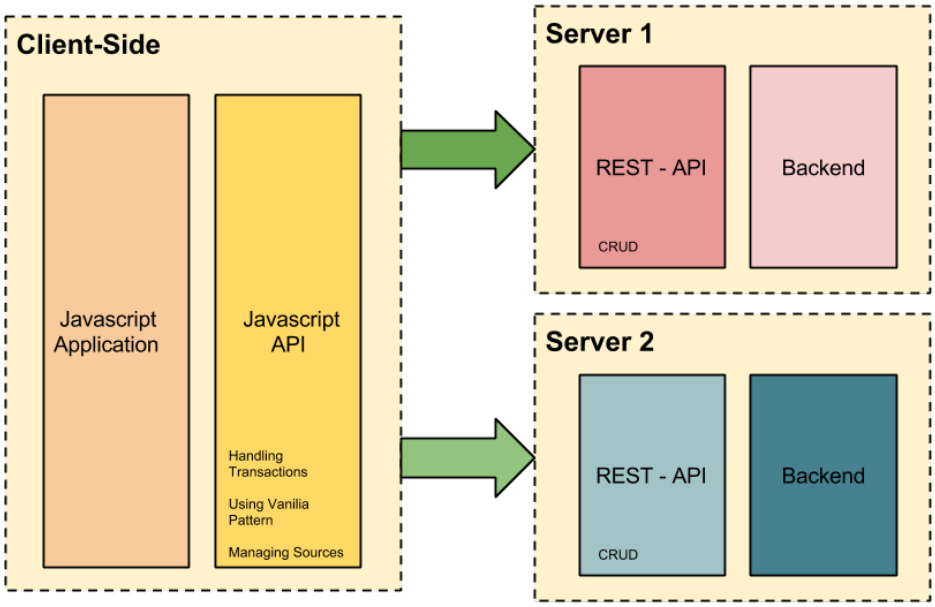

Abbildung 4.4: Grundarchitektur Client-Server der ClientOnly-Lösung.

4.2.1 Backend

Die Idee hinter der Backend-Implementierung in diesem Lösungsansatz ist das keine Anpassungen an der REST-Schnittstelle am Webservice notwendig sind. In dieser Beispiel-Implementierung wird zwar weiterhin Grails verwendet werden, jedoch unterstützen die gängigsten Backendframeworks ein REST-Interface, wie in Abschnitt 2.1 beschrieben, das ohne viel Aufwand auf die Datenbasis zugreifen kann. Diese Frameworks sind in verschiedenen Programmier- und Script-Sprachen verfügbar. In PHP sind in etwa Laravel und Zend sehr verbreitet, in JavaScript besteht die Möglichkeit NodeJs zu verwenden und in Java ist SpringIO eine der verbreitetsten Frameworks. All diese Frameworks unterstützen eine einfache Implementierung einer REST-Schnittstelle. Dies ermöglicht es das Backend leicht austauschbar zu machen, ohne dass ein großer Mehraufwand für den Entwickler entsteht.

Die Abbildung 4.5 zeigt die Änderungen in der Backend-Architektur. Wie zu sehen ist, steht im Vergleich zur vorangegangenen Architektur nur noch das Standard-REST-Interface zur Verfügung, ist eine Logik für die Manipulation der Daten notwendig, wird diese an diesem Interface angebunden. Die Struktur der Implementierung des Backends wird in der Abbildung 4.5 aufgezeigt und anhand des Beispiels in Kapitel 3 abgewickelt. Da im letzten Kapitel im Bereich Backend sehr tief auf die Transaktionsabwicklung eingegangen worden ist, behandelt dieser Absatz einerseits die

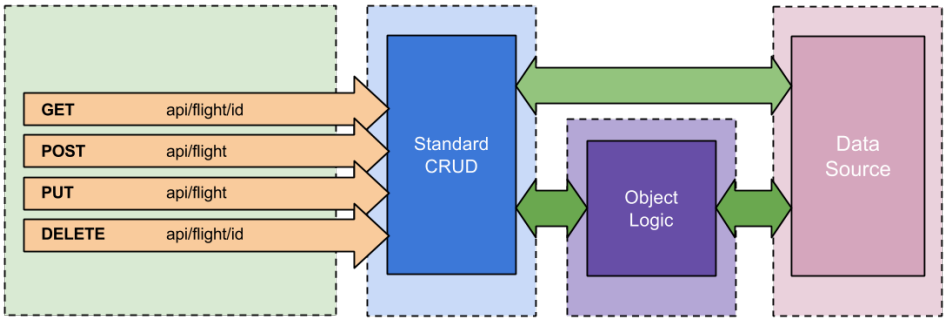

Abbildung 4.5: Grundarchitektur ClientOnly Backend.

Handhabung der Standard-REST-Schnittstellen für *Create, Read, Update, Delete*-Operationen und anderseits die Möglichkeit Logik zwischen Interface und Datenbasis hinzuzufügen.
Die einfachste Variante für die Implementierung des Backends besteht darin, dass das Interface bei validen Kommandos direkt auf die Datenbasis zugreift, dies soll mit Hilfe des jeweiligen Frameworks in Verwendung realisiert werden. Meistens werden für diese Verwendungen Routen eingerichtet die über gewisse URL-Pattern angesteuert werden und dann zu den entsprechenden Aktionen weiterleiten. In den meisten Frameworks werden im Datenmodell Eigenschaften oder Annotations gesetzt die spezifizieren das dieses Model über eine REST-Schnittstelle zu erreichen ist und somit einen direkten Zugriff auf die Datenbasis bietet. Je nach Framework werden Validierungen anhand des Datenmodels auf die eingehenden Daten durchgeführt, bevor diese Änderungen auf die Datenbasis angewandt werden können.
Die zweite Variante der Backend-Implementierung sieht vor, dass vor dem Speichern, Update oder Löschen Prüfungen durchgeführt werden. Diese Prüfungen können im Falle des Beispiels der Flugbuchungen auf vorhandene freie Sitze, auf gültige Berechtigungen oder andere Konstellationen prüfen. Dafür werden im ersten Schritt wieder Routen gesetzt, die dann in die entsprechende Businesslogik weiterleiten. Diese Logik prüft dann die Daten entsprechend und übernimmt auch die Weiterleitung an die Datenbasis oder die Abfrage aus dieser und die Interpretation der Daten, die dann an den Requestor weitergeleitet werden. Dieses Standard-Interface muss von jedem angesteuerten Webservice zur Verfügung gestellt werden um eine reibungslose Kommunikation zwischen Client und Webservices zu garantieren.

4.2.2 Client-Seite

Die Struktur der Client-API wird in der Abbildung 4.6 gezeigt. Diese Architektur ist wie bei der TCC-Lösung auf drei bzw. vier Hauptteile unterteilt.

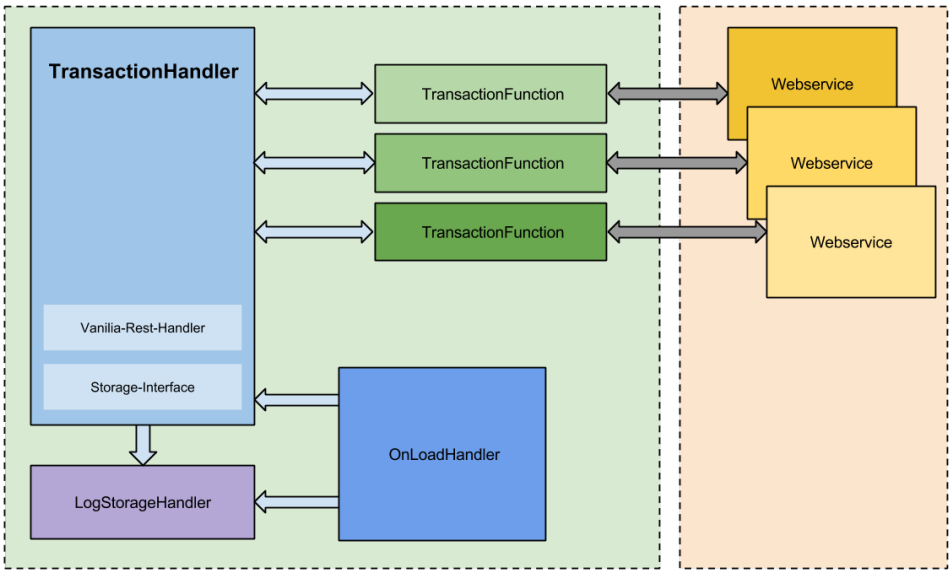

Abbildung 4.6: Grundarchitektur Client.

Die Hauptarchitektur ist daher sehr ähnlich, jedoch muss die Funktionsweise der einzelnen Komponenten angepasst werden. Der *OnLoadHandler* unterscheidet sich kaum von dem Ansatz im TCC-Pattern, wobei der *TransactionLogger* mit seinen detailreichen Loggins bei diesem Ansatz noch wichtiger ist. Der *TransactionLogger* und der *TransactionHandler* sind zwar von der Architektur her ähnlich, von der Funktionalität jedoch unterschiedlich zur TCC-Implementierung, ebenso musst die *TransactionFunction* erweitert werden.
Anhand der notwendigen Erweiterungen der vorhandenen Komponenten und dem erhöhten Aufwand der Workflowengine wird die Komplexität des Ansatzes im Vergleich zum TCC-Ansatzes erhöht. Ebenfalls werden mehrere Anfragen benötigt um die Datenintegrität zu gewährleisten. Um die verschiedenen CRUD-Operationen innerhalb einer Transaktion verwalten zu können, muss für jede Aktion eine entsprechende Gegenaktion vorbereitet werden, um deren Effekte auf den Stand des Transaktionsbeginns zurücksetzen zu können.

Create

Wenn ein neuer Eintrag während einer Transaktion erstellt wird, ist der erste Schritt der *TransactionFunctionInstance.create(dataObject)*-Befehl um die Ressource im Backend anzulegen. Nach dem erfolgreichem Anlegen des Datensatzes, wird per Standard die gespeicherte Instanz, inklusive der ID in der Datenbasis, zurückgegeben. Diese wird dann mit Hilfe des TransactionLoggers im LocalStorage abgelegt. Im

Falle eines Rollbacks wird dann die ID verwendet, um die Instanz mit Hilfe der DELETE-Methode in der Datenbasis wieder löschen zu können.

Update

Updates in der Datenbasis werden über *TransactionFunctionInstance .update(dataObject)* veranlasst. Im Hintergrund löst der Update-Command zuerst auf der Client-Seite eine Abfrage an den entsprechenden Webservice aus, um eine Sicherungsinstanz im Transaction-Logger ablegen zu können. Anschließend wird der HTTP-Update an das Backend abgesetzt. Wenn ein Rollback über den *TransactionManager* veranlasst und wird die im *TransactionLogger* abgelegte Instanz mit einem weiteren Update im Backend ausgetauscht.

Delete

Im Falle eines DELETE-Aufrufs mit einem *TransactionFunctionInstance .update(idOfObject)* wird wieder mit Hilfe eines Fetch-Aufrufs eine Instanz für Sicherungszwecke mit Hilfe des *TransactionLoggers* im LocalStorage abgelegt. Im Anschluss kann die Instanz in der Datenbasis mit einem HTTP-DELETE gelöscht werden. Für den Rollback wird die im LocalStorage abgelegte Instanz mit Hilfe eines HTTP-POST wieder in der Datenbasis angelegt.

Read

Für das Read-Kommando wird kein Ausfallszenario benötigt, ebenso wenig ein eigenes Read-Kommando wie bei der TCC-Implementierung benötigt wurde, um die in der Transaktion geänderten Daten abfragen zu können, da alle Änderungen in dieser Lösung sofort in der Datenbasis angewandt werden.

Tabelle 4.1 zeigt die Aktionen und deren notwendigen Gegenaktionen in einer Übersicht.

Tabelle 4.1: Übersicht Aktionen

Aktion	**Schritt 1**	**Schritt 2**	**Rollback**
Read	Datensatz am Server abfragen	-	-
Create	Datensatz am Server abspeichern	Instanz am Client sichern	Datensatz am Server löschen
Update	Instanz in Local-Storage sichern	Datensatz mit HTTP-PUT aktualisieren	Instanz via Update aus LS wiederherstellen
Delete	Instanz im Local-Storage sichern	Datensatz löschen in Datenbasis	Instanz wiederherstellen aus LS

Kapitel 5

Implementierung – Serverlösung

In diesem Kapitel wird die Implementierung der Serverlösung mit Hilfe des TCC-Patterns beschrieben. Der theoretische Ansatz für den Ablauf der Logik wird dem vorangegangenen Kapitel entnommen. Der erste Teil des Kapitel beschäftigt sich mit der Implementierung des Backends mit Grails. Anschließend wird die Realisierung der Client-API in JavaScript für den Transaktions-Koordinator beschrieben. Am Ende des Kapitels wird diese Lösung analysiert und mögliche Fehlerfälle mit deren Abhandlung beschrieben.

5.1 Backend-Umsetzung

Für die Implementierung für das Beispiel-Backend werden in dieser Arbeit Grails [1] verwendet. Grails ist ein auf Java basierendes Webframework welches gängige Javaframeworks wie Hibernate, Spring und Groovy ineinander vereint. Somit werden einfache ORM-Mappings mit allen möglichen Datenbanktypen, zur Laufzeit mögliche Meta-Programmierung, sowie asynchrone Programmierung ermöglicht, ebenfalls ist der Aufbau *Ruby on Rails* [2] relativ ähnlich.

Eine besondere Stärke des Frameworks ist, dass die Entwicklung dank *Convention- over- Configuration* vereinfacht und somit effizienter wird. Mit Hilfe von Grails lassen sich grundlegende REST-Schnittstellen, aufbauend auf einem vorgegebenen Datenmodell, sehr einfach generieren. Es bietet ebenfalls die Möglichkeit, mit Hilfe von sogenannten Services Businesslogik zu implementieren, die von den Controllern aufgerufen werden können, die wiederum eingehende Anfragen annehmen und verwalten können.

[1] https://grails.org/
[2] http://rubyonrails.org/

Dennoch wird versucht die Erläuterung des Codes so allgemein wie Möglich zu Halten um eine Implementierung in einem anderen Framework zu erleichtern.

5.1.1 Url-Mapping

Url-Mappings werden in Grails verwendet um alle eingehenden Requests anhand des Quell-Url-Patterns auf verschiedene Funktionen, bzw. im Falle von Grails, entsprechende Controller zu zuordnen.

```
1 //Mappings for TCC
2 "/api/tcc/flight?"(controller: "flightCacheApi", action: "index", method
      : "GET")
3 "/api/tcc/flight?"(controller: "flightCacheApi", action: "save", method:
       "POST")
4 "/api/tcc/flight/${id}?"(controller: "flightCacheApi", action: "show",
      method: "GET")
5 "/api/tcc/flight/${id}?"(controller: "flightCacheApi", action: "update",
       method: "PUT")
6 "/api/tcc/flight/${id}?"(controller: "flightCacheApi", action: "delete",
       method: "DELETE")
7
8 //TCC Confirm and Cancel Methods
9 "/api/confirm/${transactionId}"(controller: "TCC", action:"confirm",
      method: "GET")
10 "/api/cancel/${transactionId}"(controller: "TCC", action:"confirm",
      method: "GET")$
```

Die Zuordnung in Zeile 2 ist für die GET-Route und somit für die Bulk-Abfrage zuständig. Das Fragezeichen am Ende der Route signalisiert, dass auch Parameter in der URL angehängt werden können, in diesem Fall wird das Anhängen für die Transaktionsnummer benötigt. In der Klammer nach dem Muster wird dann der zu verwendende Controller, in *action* die Methode im Controller und mit *method* die entsprechende HTTP-Methode definiert.

Zeile 3 realisiert mit einem HTTP-POST das Abspeichern neuer Datensätze. Die Zeilen 4 bis 6 besitzen noch zusätzlich zu den Fragezeichen für die Transaktion-ID noch ein *${id}*, dies soll es ermöglichen dem URL-Pattern noch IDs für das Manipulieren und Abfragen spezieller Datensätze mitzugeben. Jede eingehende Abfrage, die nicht einem aufgelisteten Pattern entspricht, wird nicht verarbeitet und liefert einen Fehler zurück.

Die letzten zwei Mappings leiten die Anfragen für *Cancel* und *Confirm* an den *TCC-Controller* weiter.

5.1.2 Controller

Der Controller ist das Nervenzentrum des Backends, er ist dafür zuständig dass eingehende Requests validiert werden, die Daten an die dafür vorgesehene Businesslogik weitergibt und dann das Ergebnis wieder im richtigen Format zurück an den Requestor gibt.

```
1 class FlightCacheApiController extends RestfulController{
2   static responseFormats = ["json"]
3
4   def index(Integer max) {
5       doParamValidation()
6       respond flightService.fetch(params), [status: OK]
7   }
8
9   def save() {
10      doParamValidation()
11      flightCacheInstance.save flush:true
12      respond flightCacheInstance, [status: CREATED]
13  }
14
15  def update(FlightCache flightCacheInstance) {
16      doParamValidation()
17      respond flightService.update(params), [status: OK]
18  }
19
20  def delete(FlightCache flightCacheInstance) {
21      doParamValidation()
22      render flightService.delete(params): NO_CONTENT
23  }
24 }
```

Wichtig bei diesem angegebenen Beispiel ist die Zeile 2, in dieser wird das Rückgabeformat für den ganzen Controller auf JSON gesetzt, natürlich sind auch andere Formate wie XML möglich, jedoch sieht die Transaktions-API auf der Client-Seite ausschließlich die Verarbeitung von JSON vor. In den Zeilen 4, 9, 15 und 20 werden die Funktionen, die über URL-Mappings aufgerufen werden definiert.

Die Funktionen *Index*, *Save* und *Update* verwenden *respond* um die Antwort zu generieren und an den Client zurück zu liefern. *Respond* benötigt zwei Rückgabe-Parameter, der Erste ist das im gewünschten *ResponseFormat* zurück geliefert wird. Der zweite Parameter ist der HTTP-Status mit der die Antwort zurück gegeben werden soll.

Die Methode *Save* benötigt als Einzige keine spezielle Businesslogik, da sie lediglich das übertragene Objekt in den Cache legen muss.

Der *TCC-Controller* ist der Beschaffenheit des *FlightCacheApiController* ähnlich.

5.1.3 Businesslogik – Service

Services sind dafür zuständig Daten zu verarbeiten, sie greifen auf die Datenbasis zu und entscheiden was mit den jeweiligen Einträgen geschieht. In den nachfolgenden Punkten wird genau beschrieben wie die einzelnen Methoden agieren und im Sinne der Transaktionen handeln.

Fetch

Die *Fetch*-Methode bekommt entweder eine Transaktionsnummer über-

geben oder eine Objekt-ID und eine Transaktionsnummer. Wenn nun eine Abfrage der Daten auf ein spezielles Objekt eingeht, wird zuerst im Transaktions-Cache, geprüft ob es ein entsprechendes Objekt gibt, wird kein Objekt gefunden, wird noch in der Datenbasis gesucht. Wenn die Methode nicht auf ein spezielles Objekt abzielt, werden alle der Transaktions-Objekte und die der Datenbasis in eine Liste zusammen gefasst und an den Controller zurückgegeben.

Update
Der Update-Vorgang ist relativ ähnlich zur *Fetch*-Methode. Als Erstes wird in der Transaktion nachgesehen, ob es das zu ändernde Objekt gibt, wenn ja, wird dieses bearbeitet. Ist es nicht vorhanden, wird es aus der Datenbasis in den Cache kopiert und der Typ auf Update gesetzt und der Caching-Eintrag mit der Basis verlinkt, danach werden die Änderungen auf die Kopie angewandt.

Delete
Die Auswahl der Datenobjekte funktioniert wie bei dem Update, der einzige Unterschied besteht darin, dass der Typ im Cache auf *delete* gesetzt wird.

Confirm
Wenn der Confirm-Befehl aufgerufen wird, werden anhand der übergebenen Transaktionsnummern alle Einträge in die Datenbasis übertragen, nach erfolgreicher Übertragung, werden sie aus dem Cache entfernt. In welcher Art und Weise die Einträge übertragen werden, wird durch den Typ festgelegt.

- **Update** – Sucht den Eltern-Eintrag anhand des hinterlegten Schlüssels und wird dann mit den Parametern des Caching-Eintrags abgeglichen.
- **New** – Wird umgesetzt indem der Eintrag aus dem Cache einfach eins zu eins in die Datenbasis transferiert wird.
- **Delete** – Der Elterneintrag wird anhand des Schlüssels in der Datenbasis gesucht und aus der Basis entfernt.

Cancel
Wird der Befehl aufgerufen, werden alle Einträge die mit der Transaktion in Verbindung stehen, in der Caching-Tabelle gelöscht.

5.2 Client

Die Clientseite ist für die Abwicklung der Transaktionen in dieser Lösung verantwortlich. Die Implementierung in diesem Punkt ist analog zu dem Ansatz in Abbschnitt 4.1.3 und wird in JavaScript realisiert. Die API soll nach dem selben Schema wie in der nachfolgenden Implementierung verwendbar sein.

```
1 //Define Function
2 var f = new TransactionFunction("f1", urlFlights);
3 //Alternative mit TCC
4 var f = new TransactionFunction("f1", urlFlights, urlTccConfirm,
      urlTccCancel);
5 //Set TransactionHandler
6 f.setTransactionHandler(transHandler);
7 f.setData(data);
8
9 //Function Possibilities
10 promise = f.query();
11 promise = f.queryById(15);
12 promise = f.queryByParams({start: "12-01-2014", name: "flug"});
13 promise = f.update(14, dataObject);
14 promise = f.create(dataObject);
15 promise = f.delete(15);
```

Zeile 2 zeigt die Definition einer *TransactionFunction*. Jede Funktion muss mit mindestens einer eindeutigen ID innerhalb der Applikation und einer URL die auf den Webservice verweist, übergeben werden, weichen *Cancel*- und *Confirm*-URLs von den URLs des Transaktions-Handlers ab, können diese auch im Konstruktor definiert werden, wie es in Zeile 4 aufgezeigt wird.

In Zeile 6 wird der Funktion der *TransactionManager* übergeben in dem die Funktion abgearbeitet werden soll. Zeile 7 beschreibt wie der Funktion Datensätze für die Funktionen *.create* und *.update* gesetzt werden können, alternativ dazu können die Daten auch im entsprechenden Funktionsaufruf mit übergeben werden. Dies kann den Code übersichtlicher gestalten.

Die Zeilen 10 bis 15 zeigen die von der API zur Verfügung gestellten Aufrufe um mit dem Interface am Webservice alle CRUD-Methoden ansteuern zu können. Wichtig dabei ist, dass jede Anfrage ein sogenanntes Promise (wie in Abschnitt 2.3 beschrieben) zurückgibt und somit ermöglicht die asynchronen Antworten mit Hilfe von Callback-Funktionen leicht abzuarbeiten.

Die Funktion *.query* ermöglicht es alle verfügbaren Einträge im Bulk vom Webservice abzufragen, für das Abfragen von Datensätzen bei denen die ID bekannt ist, wird *.queryById* verwendet. Auch das Filtern nach einer Liste von Parametern ist mit *.queryByParams* möglich.

```
1 var transHandler = new TccTransactionHandler();
2
3 transHandler.setCancelUrl(urlTccCancel);
4 transHandler.setConfirmUrl(urlTccConfirm);
5
6 transHandler.startTransaction();
7 var f = new TransactionFunction("f1", urlFlights);
8 f.setTransactionHandler(transHandler);
9 f.setData(getRandomData());
10 f.create().then(successFunction1, myError);
11
```

```
12 function successFunction1(data) { transHandler.endTransaction(); }
13 function myError(data) { transHandler.endTransaction(); }
```

Um nun Funktionen innerhalb einer Transaktion mit Hilfe des TCC-Patterns abhandeln zu können, muss eine neue Instanz des *TccTransactionHandler* (Zeile 1) erstellt werden. Ebenso müssen wie in den Zeilen 3 und 4 gezeigt, die *Cancel* und *Confirm*-URLs gesetzt werden. Anschließend kann die Transaktion gestartet werden und wie schon im vorhergehenden Beispiel zu Funktionen hinzugefügt werden. Alle Funktionen die nun abgefeuert werden, wie in Zeile 10, sind ein Teil der Transaktion.

Im Falle des Beispiels wird eine neue Ressource angelegt und ein Promise zurückgegeben. Das Promise kann nun mit Hilfe von *.then()* mit den Callback-Funktionen verknüpft werden. Diese beinhalten dann die vom Service zurückgegebenen Daten. Da es sich um eine asynchrone Anfrage an den Webservice handelt, macht es Sinn das Ende der Transaktion mit *endTransaction* auszulösen, wenn auch wirklich alle Abfragen abgeschlossen sind und dies somit in den Callbacks (Zeilen 12 und 13) erledigt wird.

5.2.1 TransactionHandler

Wie bereits erwähnt ist der TransactionHandler für das Verwalten der Transaktion zuständig und wird auch oft als Transaktions-Koordinator bezeichnet, um das Konzept und die Funktionsweise besser erläutern zu können, wurden nachfolgend die wichtigsten Funktionen aufgeführt.

```
 1 function TransactionHandler() {
 2   this.startTransaction = function () {};
 3   this.endTransaction = function () {};
 4
 5   this.isTransactionRunning = function(){};
 6   this.addFunctionFired = function (functionFired, action) {};
 7
 8   this.functionFailed = function (functionFailed) {};
 9   this.functionSuccess = function () {};
10
11   this.isTransactionRunning = function () {};
12
13   this.setConfirmUrl = function (url) {};
14   this.setCancelUrl = function (url) {};
15   this.addConfirmUrl = function (url) {};
16   this.addCancelUrl = function (url) {};
17
18   function getCancelConfirmRequest(url) {};
19   function fetchNewTransactionId() {};
20 }
```

StartTransaction

Die Funktion *startTransaction* startet die Transaktion, dafür werden alle relevanten Parameter von vorangegangenen Transaktionen zurückgesetzt und die aktuelle Transaktionsnummer mit Hilfe der Metho-

de *fetchNewTransactionId* (Zeile 17) neu generiert. Ebenso wird ein Tracking-Parameter auf *true* gesetzt, um die Transaktion als offen zu deklarieren und somit den aktuellen Status verfügbar zu machen. Die letzte Aufgabe der Funktion ist es dem *StorageHandler* die Transaktionsnummer zu übergeben und somit die Transaktion als begonnen zu setzten.

EndTransaction

EndTransaction ist für das Beenden der Transaktion zuständig. Der erste Schritt besteht darin zu Prüfen, ob bereits jede Funktion ein Ergebnis, egal ob positiv oder negativ, geliefert hat, ist dies nicht der Fall, wird mit einem JavaScript-Intervall die Funktion so lange aufgerufen bis alle Ergebnisse zurück gekommen sind.

Kann die Transaktion beendet werden, prüft die Logik ob eine *TransactionFunction* auf Fehler gelaufen ist, in diesem Fall wird der *Cancel*-Befehl an die verwendeten Webservices abgesetzt.

Besitzt eine der Funktionen eine eigene TCC-URL wird diese auch aufgerufen. Ist der Aufruf abgeschlossen wird dem *StorageHandler* mitgeteilt, dass die Transaktion abgeschlossen ist und alle mit der Transaktion verbundenen Einträge gelöscht.

FunctionFailed und FunctionSuccess

FunctionFailed und *functionSuccess* werden von den *TransactionFunctions* aufgerufen wenn diese ein Ergebnis vom jeweiligen Webservice zurückbekommen, die Funktionen erhöhen dann entsprechende Zähler um bestimmen zu können, ob schon alle abgefeuerten Funktionen ein Ergebnis zurück bekommen haben.

IsTransactionRunning

IsTransactionRunning gibt den aktuellen Status der Transaktion zurück, mögliche Werte für den Status sind *NOT_STARTED*, *PENDING*, *SUCCESSFUL* oder *FAILED*.

SetConfirmUrl und SetCancelUrl

SetConfirmUrl und *setCancelUrl* werden verwendet um die URLs für die allgemeinen *Cancel*- und *Confirm*-Befehle zu setzen. Alternativ gibt es noch die Möglichkeit mehrere URLs über die *addCancelURL* bzw. *addConfirmUrl* hinzufügen.

GetCancelConfirmRequest

Die *GetCancelConfirmRequest* setzt die AJAX-Request mit Hilfe von jQuery zusammen und gibt die Request-Instanz zurück.

5.2.2 Tracking

Der *LogStorageHandler* ist dafür zuständig, dass jede Aktion die von einer *TransactionFunction* oder einem *TransactionManager* getätigt wird, solange dokumentiert ist, solange sie ein Teil einer offenen Transaktion ist. Jeder

Schritt wird mit Hilfe des LocalStorage in den Browser-Cache gespeichert. Der LocalStorage ist ein Teil des HTML5-Standards, dieser ermöglicht es Daten nach einem Key/Value Prinzip im Speicher des Browsers abzulegen.

```
1 LogStorageHandler = function () {
2   var mStorage = window.localStorage;
3
4   this.addTransaction = function (id) {};
5   this.setTransactionFinished = function (id) {};
6
7   this.addFunction = function (transactionId, functionInstance) {};
8   this.areThereUnfinishedTransactions = function () {};
9
10  function _getTransitionsFunctionArray(transId) {};
11  function _setTransitionsFunctionArray(transId, array) {};
12  function _getFunctionStorageIdentifier(transId, functionInst) {};
13 };
```

In der Zeile 2 wird die Instanz des LocalStorage des Objekts in eine Variable gespeichert um den Zugriff auf dieses Objekt zu erleichtern.

AddTransaction

AddTransaction wird vom *TransactionHandler* aufgerufen wenn eine neue Transaktion gestartet wird. Diese legt die übergebene Transaktionsnummer im LocalStorage ab, dazu muss als erstes das JSON-Array *openTransactions* abgerufen und in ein JavaScript-Objekt geparst werden. Diesem Objekt wird dann die ID beigefügt und wieder als JSON-String in den LocalStorage gespeichert.

SetTransactionFinished

SetTransactionFinished liest wie bei *AddTransaction* die offenen Transaktionen aus dem Storage, nur das die übergebene Transaktionsnummer aus dem Array entfernt wird und somit bei einem Applikationsabsturz, diese Transaktion nicht mehr manipuliert werden kann.

AddFunction

AddFunction wird aufgerufen wenn eine *TransactionFunction* abgefeuert wird, sie speichert anhand der Transaktionsnummer eine Instanz der abgefeuerten Funktion im LocalStorage ab. Die Organisation ist wieder den Vorgängermethoden angelehnt, die Transaktionsnummer wird als *Key* im LocalStorage verwendet, als *Value* wird ein JSON-Array mit den Funktionsinstanzen verwendet. Dieses Objekt wird gelöscht, wenn die Funktion *setTransactionFinished* aufgerufen wird.

StartTransaction

AreThereUnfinishedTransactions überprüft ob es im LocalStorage laufende Transaktionen gibt und liefert sie ein *true* zurück, ist dies der Fall.

Die letzten drei Funktionen sollen nicht von außen erreichbar sein und helfen bei der Verwaltung der Objekte im LocalStorage.

5.2.3 Error Handling – Rollback

Um die Integrität einer Transaktion zu gewährleisten, ist es wichtig das alle möglichen Fehlerfälle von dem verantwortlichen *TransactionsManager* abgefangen werden. Der *TransactionManager* entscheidet daher wann ein *Rollback* der Transaktion über die Webservices veranlasst werden soll.

Eine dieser möglichen Fehlerfälle tritt auf, wenn eine von mehreren Anfragen mit einem negativen Ergebnis zurückkommt, dass kann aus mehreren Gründen geschehen, sei es das wie es im Falle des Beispiels ist, kein Sitzplatz-Kontingent mehr vorhanden ist, dass eine Authentifizierung bei einem einzelnen von mehreren Webservices fehlschlägt oder das ein Webservice schlichtweg nicht erreichbar ist. Ebenso können auch Timeouts oder Konvertierungsfehler auftreten. In all diesen Fällen heißt es, alles oder gar nichts und somit müssen alle bereits abgesetzten Requests zurückgenommen werden.

Eine weitere Fehlermöglichkeit ist es, wenn die Applikation während einer Transaktion beendet wird oder Aufgrund eines Fehlers festsitzt. In diesem Fall wäre der Transaktions-Manager nicht in der Lage die restlichen Antworten der verschiedenen *TransactionFunctions* entgegen zu nehmen und somit ist es die Aufgabe des *TransactionMangers* einen Rollback zu veranlassen. In diesen Fällen wird davon ausgegangen das der Benutzer die Applikation wieder startet und somit wird der *TransactionManager* neu gestartet. In diesem Fall springt ein *onLoadHandler* ein, der beim Starten der Applikation den LocalStorage prüft, ob noch offene Transaktionen vorhanden sind. Ist dies der Fall ruft der *onLoadManager* ein Rollback aller abgesetzten Anfragen und und im LocalStorage gespeicherten *TransactionFunction*-Instanzen auf.

Der einzige Fall in dem die clientseitige Ausfallsicherung nicht funktioniert ist wenn die Applikation nicht mehr gestartet wird, dann würden bereits erstellte aber noch nicht bestätigte Einträge im Caching am Webservice hängen bleiben. Für diese Fälle gibt es mehrere Möglichkeiten diese Einträge zu löschen, eine Lösung wäre es am jeweiligen Server einen Cronjob einzurichten, der prüft ob es Einträge gibt die älter sind als ein bestimmter Zeitraum oder die Confirm/Cancel-Logik prüft bei jedem Aufruf ob es Einträge gibt in denen der Zeitraum überschritten wurde.

5.3 Analyse

Die Implementation des TCC-Pattern in JavaScript ist relativ simpel. Die wichtigsten Punkte die hierbei zu beachten sind, ist die Asynchronizität der verwendeten AJAX-Aufrufe zu den Servern. Ebenso das jeder getätigte Schritt im LocalStorage mitgeloggt werden muss um eventuelle Ausfälle des *TransactionManagers* abzufangen. Ein Nachteil dieser Implementierung ist jedoch, das eine extra Implementation auf der Server-Seite benötigt wird

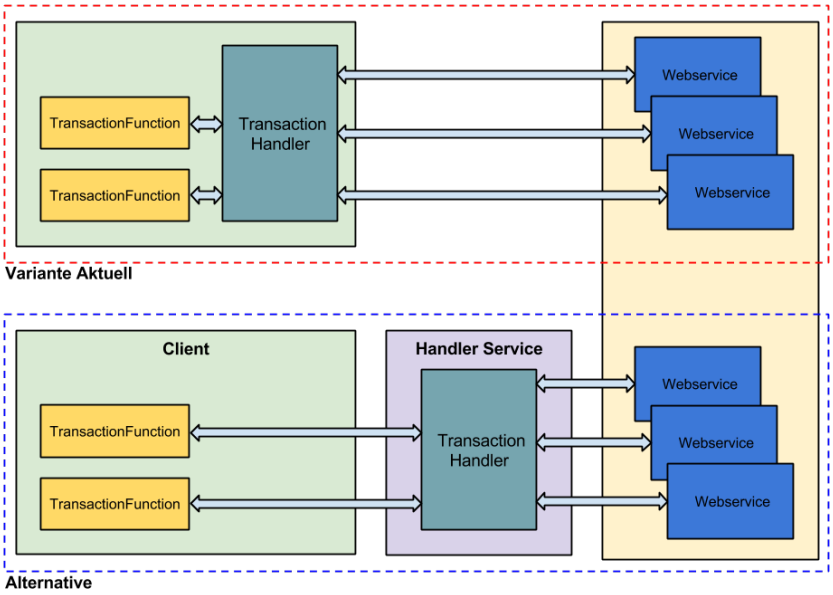

Abbildung 5.1: Alternative Platzierung des Service-Handlers.

um die Daten zu Verwalten die sich innerhalb einer Transaktion befinden. Ebenso ist es bei diesem Pattern die Aufgabe des Servers, Interfaces bereit zu stellen um *Confirm*- und *Cancel*-Kommandos entgegen zu nehmen und zu verarbeiten. Jedoch liegt der Auslöser und die Auslöselogik die sich diesen Befehlen betätigt auf der Seite des Clients.

Eine Möglichkeit um den Teil einer abstürzenden Applikation im Frontend abzufangen ist, den *TransactionManager* auf eine andere zentrale Stelle auszulagern, die Aufgabe dieser Stelle ist es dann alle getätigten Funktionen mitzuloggen und somit einen Router zwischen den Ressourcen-basierten Webservices, wie es in der Abbildung 5.1 zu sehen ist, zu bieten.

Kapitel 6

Implementierung – ClientOnly

In diesem Kapitel wird eine reine client-basierte Lösung zur Transaktionsabwicklung beschrieben. Der theoretische Ansatz für die ClientOnly-Lösung und deren Implementierung entspricht dem Ansatz aus Abschnitt 4.1.3.

Der erste Teil der Implementierung beschäftigt sich mit dem Backend und beschreibt mögliche Wege für das Bereitstellen von REST-Schnittstellen. Der zweite Teil des Kapitels beschäftigt sich mit der Implementierung der API für die clientseitige Transaktionsabwicklung. Da einige Teile der API der ClientOnly-Implementierung mit denen der TCC-Lösung gleich oder ähnlich sind, liegt das Hauptaugenmerk der Implementierungs-Beschreibung in diesem Kapitel bei den Unterschieden zur TCC-Implementierung und den ClientOnly spezifischen Methoden. Im letzten Teil des Kapitels wird der Ansatz kurz auf seine Schwachstellen und Stärken analysiert.

6.1 Backend-Umsetzung

Die Schnittstelle des ClientOnly-Ansatzes wird auf der Backend-Seite ebenfalls wie im Abschnitt 4.1.2 in Grails realisiert. Der erste wichtige Teil der Implementation ist das Routing bzw. in Grails das *URL-Mapping*. Das URL-Mapping wie es im nachfolgenden Code-Beispiel gezeigt wird, unterstützt zwei verschiedene Mapping-Varianten.

```
1 //Mapping for using Ressource on Domain-Object (Variante I)
2 "/api/reservation/"(resources: "reservation")
3
4 //Mappings via Extended RESTfull Controller (Variante II)
5 "/api/reservation"(controller: "reservationApi", action: "index", method
      : "GET")
6 "/api/reservation"(controller: "reservationApi", action: "save", method:
       "POST")
7 "/api/reservation/${id}"(controller: "reservationApi", action: "show",
      method: "GET")
```

```
8 "/api/reservation/${id}"(controller: "reservationApi", action: "update",
      method: "PUT")
9 "/api/reservation/${id}"(controller: "reservationApi", action: "delete",
      method: "DELETE")$
```

Das in Zeile 2 angeführte Mapping leitet zum eindeutigsten Ressourcen-Controller weiter, der über *Convention-over-Configuration* spezifiziert wird, das bedeutet, wenn für die Ressource ein Controller definiert ist, wird zu diesem weitergeleitet, alternativ kann ein Domain-Objekt ebenfalls mit Hilfe einer Annotation *@resource()* direkt zum Ressourcen-Controller deklariert werden, in diesem Fall wird ein Standard-RESTful-Controller verwendet, der von Grails zur Verfügung gestellt wird.

Variante II stellt die Routen zu dem eigenen REST-Controller her in denen es noch notwendig ist eine Logikebene zwischen dem Controller und der Datenbasis einzuziehen. Wie schon im Abschnitt 5.1.2 beschrieben, können im Controller, wenn erforderlich, die Requests an einem Service weitergeleitet werden.

Um nun als erstes Variante I mit dem Domain-Objekt als Ressource zu beschreiben zu können, wird das folgende Codebeispiel herangezogen.

```
1 //Mapping a
2 @Resource(uri = '/api/reservations', formats=['json'], readOnly=false)
3 //Mapping b
4 @Resource(formats=['json'])
5 class Reservation {
6     String username
7     Date reservationDate = new Date()
8
9     static belongsTo = [flight : Flight]
10
11    static constraints = {
12      flight nullable:false
13    }
14 }
```

Als erstes wird *Mapping b* erläutert, mit der Annotation wird dem System mitgeteilt das dieses Domain-Objekt als Ressource verfügbar ist, ebenfalls können hier die Rückgabe-Formate, wie in diesem Fall JSON, definiert werden. In denn darauffolgenden Zeilen (5 bis 7) wird die grundlegende Klasse mit ihren Eigenschaften definiert, diese bietet die Grundlage für den ORM-Mapper für die dahinterstehende Datenbank.

In den *Constraints* wird festgelegt unter welchen Kriterien ein Datenobjekt valide ist, hier kann angegeben werden ob etwa ein Objekt *null* sein darf, ebenso ist es möglich hier Größen und Muster der validen Datenstrukturen zu definieren.

Die zweite Variante, *Mapping a*, definiert in Zeile 2, gibt die Möglichkeit das Datenobjekt als Ressource zu verwenden ähnlich wie bei der ersten Variante. Der Unterschied besteht darinnen das diese Möglichkeit das URL-Mapping aufhebt und die hier definierte Variante in Kraft setzt, die

restlichen Eigenschaften sind gleich denen in der vorhergehenden Variante.

Für die Variante II wird als erster Schritt nach den Mappings der Controller angesteuert.

```
1 class ReservationApiController extends RestfulController<Reservation>{
2   def responseFormats = ["json"]
3   def reservationService
4
5   ReservationApiController(){
6       super(Reservation)
7   }
8
9   def save(){
10    Reservation reservationInstance = new Reservation(request.JSON)
11    reservationInstance.doValidation(.....)
12    resond reservationService.doReservation(reservationInstance)
13  }
14 }
```

Dieser wird wieder dem Standard-Restfulcontroller mit dem Typ Reservation abgeleitet (Zeile 1) und muss im Konstruktor (Zeilen 5 bis 7) ebenfalls entsprechend definiert werden. Nachfolgend wird für die Verwendung der Buisnesslogik der entsprechende Service (Zeile 3) injiziert. Nun können alle Methoden des CRUDs überschrieben werden, in denen es notwendig ist. In diesem Fall muss die Anzahl der verfügbaren Sitze überprüft werden, bevor eine neue Reservierung gespeichert wird. Dafür wird als erstes eine Reservierungs-Instanz aus den übertragenen Daten erstellt, anschließend werden diese validiert ob sie weiterverarbeitbar sind. Ist dies der Fall wird die Instanz dem Service mit dem Aufruf *doReservation* übergeben und das Ergebnis des Services an den Requestor geparst und zurückgegeben.

```
1 class ReservationService {
2   def doReservation(Reservation reservation){
3       def sum = 0;
4       Reservation.findAllByFlight(reservation.flight).each{sum += it.
      noSeats}
5
6       if(sum+seats <= flight.seats){
7           return reservation.save()
8       }
9       return false
10  }
11 }
```

Die Funktion soll nun prüfen ob noch Sitzplätze verfügbar sind, dafür werden alle Reservierungen in Verbindung mit diesem Flug abgefragt und alle bereits vergebenen Plätze zusammengezählt, sind noch Kapazitäten verfügbar, darf die Reservierung gespeichert werden.

Der Reservierungsvorgang ist in diesem Fall der einzige der mit Hilfe von Logik abgesichert werden muss, für eine Stornierung oder Abfrage einer Reservierung ist keine zusätzliche Logik notwendig und kann somit vom

Client gesteuert werden.

6.1.1 CORS

Ein Problem, dass sich bei beiden Implementierungen ergibt, sind die Sperren die sich durch die *Same-Origin-Policy* ergeben. Um dies zu Umgehen, bietet Grails mit der Konfiguration die Möglichkeit *Cross-Origin-Resource-Sharing* zu etablieren. Um nun alle notwendigen URL-Patterns für die Applikation freigeben zu können, muss der *api/* Prefex angegeben werden, wie es im nachfolgenden Code gezeigt wird. Mit Hilfe dieser Freigabe werden alle Requests die mit *api/* beginnen erlaubt und somit nicht geblockt.

```
cors.url.pattern = 'api/*'
```

6.2 Client

6.2.1 TransactionFunction

Wie bereits in Abschnitt 5.2 aufgeführt, ist die *TransactionFunction* eine Repräsentation eines Server-Aufrufs und kann mit Hilfe der übergebenen *TransactionHandler*-Instanz innerhalb einer Transaktion verwaltet werden. Für die Realisierung der *TransactionFunction* sieht die Struktur des Objekts wie folgt aus.

```
1  function TransactionFunction(t_id, t_url, t_tccConfirm, t_tccCancel) {
2      //Getter & Setter
3      this.setTransactionHandler = function (transHandler) {...};
4
5      //Request Calls from Outside
6      this.queryById = function (id) {...};
7      this.query = function () {...};
8      this.queryByParams = function (params) {...};
9      this.create = function (t_data) {...};
10     this.update = function (t_data) {...};
11     this.delete = function (id) {};
12
13     //Private Functions
14     var doCallwithBackupBefore = function (method, id) {...};
15     var doCallwithBackupAfter = function (method, id) {...};
16     this.doCall = function (t_method, t_params) {...};
17     this._doRollback = function(method, backupInstance) {...};
18
19     //Helpers
20     function getRequest(tm_method, tm_params) {...};
21 }
```

Die Herzstücke von diesem Objekt sind die Funktionen die in den Zeilen 14 bis 16 untergebracht sind. Die Aufgabe dieser Funktionen besteht darin, je nach Methodenart, entsprechende Requests mit der Funktion *getRequest()* in Zeile 20 zusammenzusetzen, diese anschließend in Promises abzusetzen

und alle für die Transaktion relevanten Werte dem *TransactionManager* zu übergeben.

```
1 var doCallwithBackupBefore = function (method, id) {
2   return new Promise(function (fullfill, reject) {
3     var backupRequest = getRequest("GET", id);
4     backupRequest.success(function (data) {
5       transactionHandler.addFunctionFired(this, method, data);
6       var realRequest = getRequest(method, id);
7       realRequest.success(function (data) {
8         transactionHandler.functionSuccess();
9         fullfill({"data": data, "id": id});
10      });
11      realRequest.error(function (data) {
12        transactionHandler.functionFailed();
13        reject({"error": error, "id": id});
14      });
15    });
16    backupRequest.error(function (error) {
17      transactionHandler.functionFailed();
18      reject({"error": error, "id": id});
19    });
20  });
21 };
```

doCallwithBackupBefore

Die Funktion *doCallwithBackupBefore* wird von den Funktionen *update* und *delete* aufgerufen. Der erste Schritt dieser Funktion ist es eine neue Promise-Instanz anzulegen, in dem alle abgesetzten Requests verwaltet werden können. Schlägt eine Anfrage innerhalb des Promises fehl, wird sofort die reject-Methode aufgerufen und die Transaktion abgebrochen.

Der nächste Schritt im Funktionsablauf besteht darin eine Backuprequest (Zeile 4) zusammenzusetzen und die Abfrage für die Backup-Instanz abzusetzen. Anschließend wird im *TransactionHandler* die Funktion *addFunctionFired* mit der Funktionsinstanz, der Methodenart und der Backup-Instanz als Übergabeparameter aufgerufen. Danach kann die wirkliche Abfrage abgefeuert und das Resultat im Promise mit Hilfe der *fullfill*-Funktion zurückgegeben werden.

DoCallwithBackupAfter

DoCallwithBackupAfter wird hauptsächlich von der Funktion *create* benötigt und arbeitet die Requests in umgekehrter Reihenfolge als *doCallwithBackupBefore* ab, daraus folgt als Schritt eins *addFunctionFired* für den *TransactionManage*, Schritt zwei, ausführen des Create-Kommandos und anschließendes Speichern des Ergebnisses für Backup-Zwecke.

DoCall wird für Standardabfragen verwendet, die nicht in einer Transaktion verwaltet werden müssen.

DoRollback

DoRollback wird vom TransactionManager oder dem *onLoadHandler* aufgerufen und mit dem Aufruf wird die Methode und das Sicherungsobjekt übergeben. Anhand von der Methode wird dann der Rollback durchgeführt. Wie in der Tabelle zusammengefasst, wird für ein Create ein Delete als Rollback durchgeführt, für ein Update wird wiederum ein Update und für Delete ein Create verwendet.

6.2.2 TransactionHandler

Der TransactionHandler ist dafür zuständig im Falle eines Fehlers während einer laufenden Transaktion alle abgesetzten Requests wieder rückgängig zu machen. Wie bereits erwähnt ist die Struktur des Objekts ähnlich der des TCC-Patterns, daher werden in diesem Punkt nur die abweichenden Komponenten näher erklärt und in der nachfolgenden Outline aufgeführt.

```
1 this.startTransaction = function () {...};
2 this.endTransaction = function () {...};
3 function doNormalRollback() {...};
4 function resetTransaction() {...};
5
6 this.addFunctionFired = function (functionFired, action, backup) {...};
```

StartTransaction

Wird zum Start der Transaktion angewendet. Die Aufgabe der Funktion besteht darin alle Transaktions relevanten Parameter zurück zu setzen und einen entsprechenden Eintrag im *TransactionLogger* zu setzen.

EndTransaction

EndTransaction wird aufgerufen wenn alle *TransactionFunctions* die dem *TransactionHandler* zugeordnet wurden, abgefeuert sind. Die erste Aufgabe besteht darin zu überprüfen ob bereits jede *TransactionFunction* ein Ergebnis, sei es positiv oder negativ, zurückgeliefert hat. In diesem Fall wird überprüft ob eine oder mehrere Anfragen fehlgeschlagen sind, existieren keine Fehler, kann die Transaktion abgeschlossen werden. Im Fehlerfall jedoch müssen alle erfolgreichen Anfragen durch einen Aufruf der *DoNormalRollback*-Funktion mit Hilfe der *TransactionFunction* zurückgerollt werden. Nachdem der Rollback geglückt ist, wird die Transaktion im *TransactionLogger* als abgeschlossen gesetzt.

DoNormalRollback

Die Funktion *DoNormalRollback* ist dafür zuständig das alle abgespeicherten *TransactionFunction*-Instanzen, die im LocalStorage mit Hilfe des *TransactionLoggers* abgelegt wurden, aufgerufen werden. Die zurückgelieferten Funktionen werden in einem Array retuniert und können so relativ einfach durchlaufen werden. Bei jedem Durchlauf kann

somit die schon vorher beschriebene Methode *_doRollback*, die ein Teil der *TransactionFunction* ist, aufgerufen werden.

AddFunctionFired

AddFunctionFired wird automatisch beim ausführen der *TransactionFunction* aufgerufen, wenn diese ausgeführt. Der Funktion wird eine Instanz der abgefeuerten Funktion, die Art der HTTP-Methode und der Backup-Datensatz übergeben. Die übergebenen Parameter werden in ein Objekt zusammengefügt das dann an den *TransactionLogger* weitergegeben wird.

6.2.3 TransactionLogger

Für die Vanilia-REST Implementierung muss der TransactionLogger erweitert werden. Im TCC-Pattern musste der Logger nur wissen ob Transaktionen noch aktiv sind und welche Funktionen noch betroffen sind, nicht benötigt wurden jedoch die Funktionsinstanzen und die dazugehörigen Backupdatensätze. Daher wurde der Logger mit der Funktion *getFunctions* hinzugefügt.

```
this.getFunctions = function(transactionId){...};
```

Die Methode ruft als erstes alle TransactionFunction-Ids der übergebenen Transaktions-ID ab, um dann anhand der Funktions-Ids die Funktions-Instanzen aus dem LocalStorage abzufragen und dann gesammelt in Form eines Arrays an den TransactionManager zurückzuliefern.

6.2.4 Fehlermöglichkeiten

Fälle in denen die Integrität der Datenbasis gefährdet sein könnte sind denen des vorangegangenen Kapitels sehr ähnlich. Mögliche Fehlerfälle die hierbei entstehen können. Sind wenn ein oder mehrere Request negativ zurückkommen, die Applikation während einer Transaktion beendet wird und/oder nie wieder geöffnet wird. Diese Fälle werden nach dem selben Prinzip behandelt.

Der einzige Unterschied besteht darin das es keine Möglichkeiten für Rollbacks gibt die auf der Serverseite ausgelöst werden, da die Transaktionsbehandlung rein auf der Clientseite getätigt wird.

6.3 Analyse

Die Implementierung der API in JavaScript ist im Vergleich zum TCC-Pattern komplexer, da für jede Aktion eine Backup-Aktion benötigt wird, um im Fehlerfall alle Aktionen zurücknehmen zu können. Die Backup-Instanzen sind notwendig da auf der Serverseite überhaupt kein Cashing oder keine Transaktionsabarbeitung passiert und somit wirklich alles auf der

Client-Seite abgewickelt wird. Der Komplexitätsgrad wird ebenfalls durch das Warten der Sicherungsinstanzen erhöht.

Grundsätzlich gesehen ist diese Lösung für eine reine clientseitige Transaktionsverwaltung besser als die des TCC-Pattern, da wirklich direkt mit einem Standard-REST-Interface gearbeitet werden kann. Jedoch werden bei Rollbacks von Create-Kommandos die Einträge wieder gelöscht und somit IDs in der Datenbasis vergeudet.

Kapitel 7

Vergleich der Lösungen

In dieser Arbeit wurden viele Lösungen und Möglichkeiten für die Realisierung von Transaktionen über REST behandelt. Um nun einen genaueren Überblick über die Lösungen geben zu können werden in diesem Kapitel im ersten Punkt die Serverlösungen aus Kapitel 3 zu den zwei eigenen Lösungen. Nachfolgend werden die zwei eigenen Lösungen analysiert. Aufgrund der Struktur der eigenen Implementierungen ergeben sich Analysepunkte wie Fehlerbehandlung, Parallelismus, Performance und zusätzlicher Implementierungsaufwand um die Vor- und Nachteile der jeweiligen Implementierung herausfiltern zu können.

7.1 Vergleich eigene Lösungen zu Vorhandenen

Der Vergleich der eigenen Lösungen mit den bereits gut Etablierten in Kapitel *State of the Arts*, muss im ersten Schritt zwischen serverseitigen und clientseitigen Lösungen unterschieden werden und somit der Standort der verwendeten Transaktionslogik.

Als serverseitige werden die Punkte *Batched Transactions* (Abschnitt 3.2.1), *Transaktion als Quelle* (Abschnitt 3.2.2), *Optimistisches REST* (Abschnitt 3.2.3) und *RETRO* (Abschnitt 3.2.4) gesehen und sind somit nicht 1:1 vergleichbar mit den implementierten Lösungen, da diese nur für jeweils einen Webservice gültig sind.

Clientseitige Lösungen sind somit das *Zeitstempel basierte System*, das *Try-Cancel-Confirm-Pattern* und Client-Only.

7.1.1 TCC im Vergleich zu den vorhandenen Serverlösungen

Batched Transactions

Batched Transactions mit überladenen POST im Vergleich zur TCC-Transaktion unterscheidet sich hauptsächlich darin, dass sich die Komplette Transaktionslogik im Backend befindet, somit wird die Trans-

aktion mit ihrer ID im Backend erstellt und an den Client übergeben. Somit unterscheidet sich die Erstellung und die Verwaltung der Transaktion in diesem Punkt beim TCC lediglich durch den Ort der Erstellung. Die Kommunikation zwischen Server und Client der beiden Methoden ist beinahe gleich. Ebenso wird bei beiden Methoden jeweils eine URL für den Commit bzw. Rollback zur Verfügung gestellt.

Transaktionen als Quelle

Transaktionen als Quelle und die *optimistische Technik für REST-Transaktionen* bedienen sich der REST-Struktur und realisieren die Transaktions-Verwaltung mit zusätzlichen Transaktions-Ressourcen auf der Server-Seite, jedoch kann die Verwaltung der Transaktion nur bedingt auf der Clientseite realisiert werden, da Transaktionsstart und Ende über den Server abgewickelt werden müssen, somit ist diese Methode wieder nur an einen Webserver gebunden und bietet keine Möglichkeit für eine dezentrale Architektur. Die TCC-Implementierung erstellt vergleichsweise die Transaktion auf der Clientseite, benötigt keine verschiedenen Ressourcen-URIs für die Aktionen und benötigt lediglich eine Realisierung von Cancel und Confirm auf der Serverseite.

RETRO

Das *RETRO*-Model ist eine gute Mischung aus den Methoden Batched Transactions mit überladenen POST, *Transaktionen als Quelle* und die *optimistische Technik für REST-Transaktionen*. RETRO ist eine reine Serverseitige Lösung, die eine Verwendung des gesamten Create, Read, Update und Delete-Spektrum ermöglicht. Ähnlich wie bei der TCC-Implementierung erstellt RETRO Kopien der Datensätze innerhalb der Transaktion und kapselt diese somit von der Datenbasis ab. Ebenso werden URLs für die Commit und Rollback-Befehle zur Verfügung gestellt. Somit besteht der Unterschied zwischen RETRO und TCC darin das die Verwaltung der Transaktionen bei TCC in der Verantwortung des Clients liegt und bei RETRO auf der Serverseite, somit ist RETRO nicht für eine Dezentralisierung geeignet.

Zeitstempel basiertes zwei Phasen-Protokoll

Das Zeitstempel-System beinhaltet die komplette Transaktionsabwicklung auf der Clientseite, jedoch wird wie bei den vorangegangenen Implementierungen eine Logik auf der Serverseite verwendet um die Transaktionsdaten von der Datenbasis abzukapseln und das Zeitstempelregelwerk abzuarbeiten. Ebenso arbeitet dieses Protokoll nur mit GET und PUT Operationen und ermöglicht somit nicht das komplette CRUD-Spektrum und kein Standard-REST-Interface wie die TCC-Lösung. Für den Commit oder den Rollback wird ein write-Kommando zur Verfügung gestellt, alternativ zu den Kommandos werden auch Zeitstempel für das Festlegen der Reihenfolge des Abarbeitens verwendet.

Das TCC-Pattern vereint alle Vorteile der aufgeführten Lösungen. Es ermöglicht das volle CRUD-Spektrum als REST-Interface und ist somit universell einsetzbar. Da der Transaktions-Manager auf der Clientseite platziert ist, können multiple Webservices mit in die Transaktion einbezogen und adressiert werden. Da das Abhandeln der Transaktion auf der Clientseite geschieht, muss der Server lediglich ein Kommando für den Commit und Rollback zur Verfügung stellen und ist somit leicht zu Warten. Da die Transaktionsdaten wie die Transaktionsnummer am Client erzeugt werden, ist die am Server notwendige Implementierung nur für die Abkapselung der Transaktionsdaten von der Datenbasis und den Transfer von der Transaktion in die Basis zuständig.

7.1.2 ClientOnly

Die ClientOnly-Implementierung ist die einzige von den behandelten Varianten, die wirklich komplett ohne extra Serverunterstützung funktioniert. Wird die *Batched Transactions*-Variante verglichen, unterstützt diese zwar das volle REST-Spektrum, nur wird für das Commit und Rollback die POST-Methode überschrieben und die Abkapselung auf der Server-Seite realisiert.

Die mit REST als Ressourcen arbeiteten Varianten lehnen übergeben zwar schon einen Teil der Verwaltung der Transaktion an den Client, jedoch wird der Server für die *TransactionFactory* benötigt, die für das Anlegen der Transaktionen zuständig ist.

RETRO unterstützt wie die ClientOnly-Implementierung zwar das volle REST-Spektrum, jedoch verwendet auch diese Variante Serverlogik für die Abhandlung der Transaktion und ist nicht für die Dezentralisierung geeignet.

Das Zeitstempelsystem besitzt ein ähnliches Problem da es nicht das volle REST-Spektrum unterstützt. Ebenso wird auch hier Logik auf der Serverseite benötigt um die Commit und Rollback-Aktionen durchführen zu können.

ClientOnly ist somit die einzige Möglichkeit REST-Transaktionen zu verwirklichen ohne zusätzlichen Aufwand auf der Serverseite zu erzeugen. Die mit dieser Variante kommenden Nachteile wurden einerseits im letzten Kapitel erläutert und werden ebenfalls im nachfolgenden Punkten analysiert.

7.2 Analyse und Vergleich der Fehlerbehandlungen

Datenintegrität ist der wichtigste Bestandteil von Transaktionen und daher einer der größten Bewertungskriterien für Transaktionsmanager. Bei den aufgeführten Serverlösungen stellt sich die Fehlerbehandlung im Vergleich zu den implementierten Client-Lösungen als relativ einfach dar. Serverlösun-

gen sind unabhängig von anderen Webservices und besitzen einen direkten Zugriff zur Datenbasis, ebenso sind Webservices unabhängig von anderen Webservices und eliminieren somit die Fehlermöglichkeit durch Verbindungsausfälle.

Die TCC-Implementierung schafft es sich der Vorteile von Server und Client zu bedienen. Da der Transaktions-Manager auf der Clientseite liegt, wird der Applikation ermöglicht Webservices auf verschiedenen Servern aufzurufen und zu Verwalten. Um volle Datenintegrität zu gewährleisten werden daher Vorkehrungen auf Server- und Clientseite vorgenommen. Die ClientOnly-Methode hingegen kann nur auf der Clientseite abgehandelt werden, da die Webservices lediglich über ein Standard-REST-Interface abgewickelt werden sollen.

Eine der wichtigsten Fehlerbehandlungen ist wie bereits in Abschnitt 6.2.4 erwähnt, wenn die Applikation während einer laufenden Transaktion unterbrochen wird oder abstürzt und somit die Transaktion nicht fertig abwickeln kann. Die Grundidee für die Abwicklung in solchen Fällen ist für die im LocalStorage mitprotokollierten Transaktionen mit deren Funktionen wiederherzustellen und anschließend ein Rollback aufzurufen. Um diesen Fall entsprechend Testen zu können wird das Beenden der Transaktion über den entsprechenden TransactionHandler (TCC oder ClientOnly) nicht aufgerufen und somit wird verhindert dass die Transaktionsdaten im LocalStorage nicht gelöscht werden. Somit kann mit einer Verzögerung die *onLoadHandler.onLoad()*-Methode aufgerufen werden, um somit den Neustart der Applikation bei einem Ausfall während einer laufenden Transaktion zu simulieren. Das nachfolgende Beispiel beschreibt den entsprechenden Testfall. Das Szenario ist von der Grundstruktur her für beide Implementierungen anwendbar.

```
1 var testApplicationCrash = function(){
2   var th = new TransactionHandler();
3   th.startTransaction();
4
5   var f1 = new TransactionFunction("f1", urlFlightsTCC);
6   f1.setTransactionHandler(th);
7
8   var f2 = new TransactionFunction("f2", urlFlightsTCCTomcat);
9   f2.setTransactionHandler(th);
10
11  var p1 = f1.create(getRandomData());
12  p1.then(onF1Success, onFail);
13  function onF1Success(data) {
14      var p2 = f2.create(getRandomData());
15      p2.then(onF2Success, onFail);
16  }
17
18  function onF2Success(data) { onLoadHandler.onLoad(); }
19  function onFail(data) { onLoadHandler.onLoad(); }
20 };
```

Am Anfang des angeführten Tests wird der TransactionHandler initialisiert und die Transaktion gestartet, anschließend werden zwei Funktionen zu verschiedenen Webservices erstellt und jeder wird die Instanz des Transactionsmanagers übergeben. Anschließend werden zwei Datenobjekte mit Hilfe der Funktionen an die Webservices zum Anlegen gesandt, sind die Objekte angelegt, kann der Rollbacktest über die Onload-Logik gestartet werden.

Da die genaue Funktionsweise der OnLoad-Logik in den vorangegangen Kapitel nie erläutert wurde, beschäftigen sich der nächste Absatz mit dem Ablauf der Logik.

```
var onLoadHandler = function(){

  this.onload = function(){
    var logger = new LogStorageHandler();
    var openTransactions = logger.getStartedTransactions();

    for(var key in openTransactions){
      var functions = logger.getTransitionsFunctionArray(
     openTransactions[key]);

      for(var key1 in functions){
          var fD = functions[key1];
          var transactionFunctionTemp = new TransactionFunction(fD.url);

          if(functionData.type === "vanila"){
              transactionFunctionTemp._doRollback(fD.method, fD.backup);
          }else{
              transactionFunctionTemp._doRollback(fD.tccCancelUrl);
          }
      }
    }
  };
};
```

Der onLoadHanlder soll einmalig beim Start der Webapplikation aufgerufen werden und wird daher in den meisten Fällen vom Entwickler mit dem onReady-Event aufgerufen. Der Handler ist dafür verantwortlich das er mit Hilfe des LogStorageHandlers eventuelle, auf der Strecke liegen gebliebene Transaktionen aus dem LocalStorage ausliest und die einzelnen Aufrufe rückgängig macht.

Dafür muss anfangs eine neue Instanz des StorageHandlers erzeugt werden. Diese bietet dann eine Abfrage, die alle nicht beendeten Transaktionen die von diesem Client ausgingen, in Form einer Liste zurückgibt. Mit Hilfe dieser Liste an Transaktionsnummern können dann alle abgefeuerten Requests ausgelesen werden. Wie in den vorhergehenden Kapiteln definiert, beinhalten diese, wie nachfolgend veranschaulicht, je nach Implementierung verschiedene Parameter, die für den Rollback von Nöten sind.

```
var tccBackup = { type: "tcc", id: functionFired.getId(), state: "
     started", url: functionFired.getUrl(), action: action, tccCancelUrl:
      tccCancelUrl};
```

```
2
3 var vanilaBackup = { type: "vanila", id: functionFired.getId(), state: "
      started", url: functionFired.getUrl(), action: action, backup:
      backup };
```

Anhand dieser Daten können somit neue TransactionFunctions erstellt werden die ihren Backups entsprechen und somit den TransactionFunctions die Teil der Transaktion waren bevor die Applikation beendet wurde. Mit diesen Funktionen können nun die entsprechenden Rollbacks veranlasst werden, die in jeder TransactionFunction definiert sind.

Sollten in diesem Stadium Fehler auftreten, wie ein erneuter Absturz der Applikation oder ein Timeout der Rollback-Requests, werden die Transaktionen nicht aus dem LocalStorage entfernt und es kann somit zu einem späteren Zeitpunkt ein erneuter Versuch gestartet werden.

Um nun wieder zu unseren Test zurückzukehren, muss festgestellt werden, dass diese Lösung wahrscheinlich in den meisten Fällen ausreichend ist, da der Normaluser nach einem Programmabsturz die Applikation wieder startet und den unterbrochenen Vorgang noch einmal durchführt um zu einen Erfolg zu kommen. Diese Fallback-Variante ist die einzige Möglichkeit für ClientOnly-Transaktionen nach einem Absturz rückgängig zu machen. Für die TCC-Implementierung können, wie im Abschnitt 5.2.3 erläutert, alternative und zusätzliche Maßnahmen auf der Serverseite getroffen werden.

7.2.1 Timeout

Ein weiterer Fehler der im Web oft auftritt sind Zeitüberschreitungen von Anfragen an den Webservice die bei den meisten Browsern ab 30 Sekunden oder mehr gesetzt werden. Im Falle von AJAX wird in solchen Fällen die onError-Funktion aufgerufen. Wenn das nun im Rahmen des Transaction-Handlers betrachtet wird, so sieht es die Logik vor das die gesamte Transaktion zurückgerollt wird.

Um diesen Fall testen zu können wird ein einfacher Test verwendet in dem zwei Ressourcen auf zwei verschiedenen Webservices angelegt werden, wie es das nachfolgende Beispiel für die Clientseite zeigt.

```
1 var testRequestTimeout = function(){
2   var th = new TransactionHandler();
3   th.startTransaction();
4
5   var f1 = new TransactionFunction("f1", urlFlightsTCC);
6   f1.setTransactionHandler(th);
7   var f2 = new TransactionFunction("f2", urlFlightsTCCTomcat);
8   f2.setTransactionHandler(th);
9
10  var p1 = f1.create(getRandomData());
11  p1.then(onF1Success, onFail);
12  function onF1Success(data) {
13      var p2 = f2.create(getRandomData());
```

```
14        p2.then(onF2Success, onFail);
15    }
16
17    function onF2Success(data) { th.endTransaction(); }
18    function onFail(data) {th.endTransaction(); }
19 };
```

Auf der Backendseite wird für diesen Test eine *Wait*-Aufruf mit einer Verzögerung von 45s eingebaut um sicher zu stellen, dass der Aufruf im Browser auch wirklich auf eine Zeitüberschreitung stößt. Dafür wird der Create-Controller im Grails-Backend wie folgt modifiziert.

```
1     def beforeInterceptor = {
2       wait(45000);
3     }
```

Um jede Anfrage zu Treffen wird im entsprechenden Controller der beforeInterceptor verwendet. Dieser wird bei jedem Controller-Aufruf angesprochen bevor die entsprechende Operation wie Save, Update oder Delete aufgerufen wird und ist somit wie geschaffen für jede eingehende Anfrage eine Verzögerung zu hinterlegen.

Die Timeout-Problematik lässt sich bei beiden Implementierungen gleich gut abfangen und wird daher nicht als großes Problem betrachtet.

7.3 Parallelismus

Ein weiteres wichtiges Kriterium für eine sichere Transaktionshandhabung ist, dass sichergestellt werden muss, dass auch mehrere Transaktionen parallel zu einander abgewickelt werden können ohne das die Datenintegrität gefährdet wird. Dies wird im TCC-Pattern mit Hilfe der Transaktionsnummern die jedem Datensatz zugeordnet wird, solange sie sich im Transit befindet realisiert.

Bei der ClientOnly-Variante wird dies nach dem *First-Come-First-Serve*-Prinzip abgearbeitet und die Änderungen an der Datenbasis in der Reihenfolge des Auftreffens, ähnlich zum zeitstempelbasierten System, auf das Backends festgelegt. Mit dieser Variante kann es natürlich passieren das bei einem späteren Rollback, wie zum Beispiel nach einem Systemabsturz, dass falsche Backup wieder eingespielt wird. Um dies zu verhindern kann die Datenbasis, je nach Sensibilität, mit einer Sperre erweitert werden, um diesen Fall abfangen zu können. Eine weitere alternative wäre die jeweiligen Ressourcen mit Zeitstempel zu erweitern und nur dann einen Rollback zuzulassen wenn der Eintrag in der Zwischenzeit nicht von einer anderen Transaktion und somit von einer anderen Anwendung geändert wurde. Nur mit diesen Hilfsmitteln kann für die ClientOnly-Implementierung eine Isolation für die Datensätze im Transit erreicht werden. Isolation ist aufgrund des zwei Tabellensystems in dieser TCC-Implementierung kein Problem.

Parallele Transaktionen innerhalb einer Applikation sind kein Problem, da für jede Transaktion ein neuer Handler angelegt werden muss und dieser dynamisch neue Transaktionsnummern erzeugt. Das nachfolgende Beispiel zeigt einen Test für zwei parallele und von einander unabhängige Transaktionen.

```
1 var testApplicationCrash = function(){
2     var th1 = new TransactionHandler();
3     th1.startTransaction();
4     var th2 = new TransactionHandler();
5     th2.startTransaction();
6
7     var f1 = new TransactionFunction("f1", urlFlightsTCC);
8     f1.setTransactionHandler(th1);
9
10    var p1 = f1.create(getRandomData());
11    p1.then(onF1Success, onFail);
12    function onF1Success(data) {
13        th1.endTransaction();
14    }
15
16    var f2 = new TransactionFunction("f2", urlFlightsTCCTomcat);
17    f2.setTransactionHandler(th2);
18
19    var p2 = f2.create(getRandomData());
20    p2.then(onF1Success, onFail);
21    function onF2Success(data) {
22        th2.endTransaction();
23    }
24
25    function onFail(data) {
26        th1.endTransaction();
27        th2.endTransaction();
28    }
29 };
```

Dieses Beispiel zeigt wie zuerst zwei Instanzen für des TransactionHandlers angelegt werden und diese jeweils einer Funktion zugewiesen werden. Anschließend werden mit Hilfe dieser Funktionen neue Ressourcen auf zwei verschiedenen Webservices erzeugt. Da die Aufrufe asynchron abgehandelt werden, können diese parallel und komplett unabhängig von einander innerhalb ihrer Transaktionen verarbeitet werden.

7.4 Performance

Effizienz und Performance werden im REST-Umfeld sehr groß geschrieben. Dank der REST-Architektur lassen sich entsprechende Applikationen meist gut skalieren und kommen somit leicht mit schnell steigenden Benutzerzahlen zurecht. Um die Applikation möglichst schnell und somit ein gutes Userexperience zu schaffen, aber auch die Anfragen an den Server so gering

wie möglich gestalten zu können, ist es wichtig die Anzahl der Anfragen und den damit verbundenen Payload so gut es geht zu minimieren.

Daher sollte auch die Beschaffenheit der verwendeten APIs bedacht werden. Im Falle der TCC-Implementierung bedeutet dass es für jede Transaktion mit n Anfragen an eine Anzahl an Webservices m eine totale Anzahl an Anfragen von $n + m$ ergibt.

Bei der ClientOnly-Implementierung ist die Rechnung etwas komplexer, diese setzt sich auf den Read-Anfragen n und den Datenbasis manipulierenden Anfragen m zusammen. Die Berechnung für den Fall das kein Rollback ausgelöst wird, lautet $n + m$. Für den Rollback muss zusätzlich noch die Anzahl der bereits abgesetzten manipulierenden Anfragen x mit in Betracht gezogen werden und ergibt als Formel für die Berechnung der Anfragen $n + 2*x$.

Um nun die Unterschiede in der Performance analysieren zu können wird der nachfolgende Test verwendet.

```
1 var th = new TransactionHandler();
2 th.startTransaction();
3
4 var testVanilaPerformance = function (rounds) {
5   var f = new TransactionFunction("f"+rounds, urlFlightsTCC);
6   f.setTransactionHandler(th);
7   var p = f.create(getRandomData());
8   p.then(function(data){
9     rounds--;
10    if(rounds >= 0){
11      testVanilaPerformance(rounds);
12    }else{
13      th.endTransaction();
14    }
15  }, function(data){
16    th.endTransaction();
17    console.log("Error on Call " + rounds);
18  });
19 };
```

Ebenfalls werden Hilfsmethoden und Variablen in den TransactionHandler hinzugefügt um die Zeitspanne vom Transaktionsstart bis zum Transaktionsende zu bekommen. Aber auch ein Zähler für die abgesetzten Requests ist notwendig.

Die Testmethode verwendet einen rekursiven Ansatz um eine dynamische Anzahl an Anfragen innerhalb einer Transaktion abzusetzen und somit eine Statistik erstellen zu können. Diese Methode ist für ClientOnly-REST implementiert und kann mit kleinen Änderungen auch für das TCC-Pattern verwendet werden.

In der nachfolgenden Tabelle 7.1 wird eine Übersicht gezeigt in der Anzahl an Requests und die benötigte Zeit für die zwei Implementierungen gegenüber gestellt wird. Wie die Tabelle zeigt bleibt die Anzahl der tatsächlich abgesetzten Abfragen bei der TCC-Implementierung relativ klein im Ver-

Tabelle 7.1: Übersicht Anzahl Anfragen je Implementierung

Anfragen	**1**	**3**	**5**	**10**	**100**
TCC (wirkl. Requests)	2	4	6	11	101
ClientOnly (wirkl. Requests)	1 – 2	3 – 6	5 – 10	10 – 20	100 – 200

gleich zur ClientOnly-Implementierung, bei dieser verdoppelt sich meist die Anzahl der tatsächlich abgesetzten Abfragen da für jede Aktion ein Backup gespeichert wird.

7.5 Schlussfolgerung

Beide verglichenen Client-Implementierungen in diesem Kapitel besitzen großes Potential für eine verlässliche Abwicklung von REST-Transaktionen innerhalb bestimmter Parameter. Jedoch zeigen die durchgeführten Tests, dass vor allem durch den Faktor Benutzer auf der Client-Seite nicht alle ACID-Eigenschaften erfüllt werden können und somit nicht alle Regeln für eine valide Transaktion eingehalten werden können.

Dieser Faktor wird vor allem bei der ClientOnly-Lösung tragend, da bei dieser Variante wirklich nur die im Browser laufende Workflowengine für die Transaktionsverwaltung zuständig ist und keine Sicherheitsvorkehrungen wie beim TCC auf der Server-Seite getroffen werden können. Ein weiterer Pluspunkt der TCC-Variante ist, dass diese anhand der Struktur um einiges weniger Anfragen an das Backend benötigt als die ClientOnly-Lösung.

Die ClientOnly-Lösung ist aus meiner Sicht eine ausreichende Lösung im Sinne des aufgeführten Beispiels um sicherzustellen, dass wenn eine aus mehren von einander abhängigen Requests auf Fehler läuft, die bereits abgesetzten wieder zurückgenommen werden können. Bei anderen auftretenden Fehlern versucht die Lösung zwar so gut es geht diese zu kompensieren, jedoch ist das nicht mehr möglich wenn die Applikation nach einem Absturz nicht mehr neu gestartet wird. Ebenso werden nach dem Rollback von *Create* und *Delete* Kommandos unnötige IDs in der Datenbasis vergeudet, da die Datensätze aus den Backup-Instanzen in der Datenbasis wieder neu angelegt werden. Da bei einem *Delete* die entsprechenden Einträge wirklich gelöscht werden und dann wieder neu abgelegt, kann der Rollback nicht als richtiger Rollback im Sinne von Transaktionen gesehen werden kann.

Der größte Vorteil dieser Lösung, ist wie bereits im Ansatz erklärt, das er mit jedem Standard-REST-Interface verwendbar ist und somit ohne zu-

sätzlichen Aufwand einsetzbar ist.

Die TCC-Lösung im Gegenzug benötigt einige Anpassungen am Backend. Dieses muss die Transaktionsabwicklung innerhalb der eigenen Datenbasis übernehmen und ebenso zwei Links für einen Commit oder einen Rollback zur Verfügung stellen, der die entsprechenden Aktionen am jeweiligen Webservice startet. Mit ein paar zusätzlichen Erweiterungen um eventuelle Datenleichen beim Ausbleiben der Commit oder Cancel Kommandos können die ACID-Eigenschaften für Transaktionen gut eingehalten werden.

Aus meiner Sicht sind beide Ansätze gut brauchbar, jedoch muss der Entwickler gut Bedenken für welche Anwendungen er sie einsetzt und ob er die gegebenen Beschränkungen in Kauf nehmen kann.

Kapitel 8

Schlusswort

Single-Page-Webapplikationen mit REST-Services im Hintergrund werden aufgrund ihrer Dynamik und guten Skalierbarkeit immer populärer. Ein weiteres Plus für clientseitige JavaScript-Applikationen ist die Möglichkeit auf Services von verschiedenen Servern zuzugreifen. Ebenso geben REST-Services anhand ihrer Zustandslosigkeit eine gute Abkapselung von parallel laufenden Anfragen an den Server.

Diese JavaScript-Frameworks wie AngularJs, EmberJs oder ExtJs setzen auf Datenübertragung basierend auf AJAX mit REST-Services im Hintergrund. Diese Frameworks entwickeln sich natürlich stetig weiter und sind im Großen und Ganzen sehr zuverlässig und ausfallsicher. Ebenso bieten JavaScript-Applikationen den Vorteil das vieles der für die Applikation benötigte Rechenleistung vom Client-Browser übernommen werden wird und der Server somit entlastet werden kann.

Ein größeres Problem wird weiterhin die Transaktionsabhandlung über JavaScript, wie in dieser Arbeit beschrieben, bleiben. Da es in einigen Fällen nach wie vor keine Sicherheit und Garantie gibt die Datenintegrität zu garantieren.

Da dieses Thema anhand des steigenden Bedarfs immer mehr Priorität bekommt und somit auf verschiedenen internationalen Entwickler- und Fachkonferenzen mehr Aufmerksamkeit erlangt, wird es wahrscheinlich nur eine Frage der Zeit sein bis es für dieses Problem eine adequate und professionelle Lösung geben wird, die eine reine clientseitige Transaktionsabhandlung unterstützt.

Für Anwendungsfälle wie das Beispiel in dieser Arbeit, ist der aktuelle Standard meiner Meinung nach im Moment ausreichend und gut funktionierende Lösungen lassen sich leicht implementieren.

Der Transaktionsproblematik beiseite, bieten diese Technologien meiner Meinung nach eine solide Basis für alle Arten von Webapplikationen verschiedenster Komplexitätsgrade. Sowohl für den normalen Heimanwender-Bereich wie Ticketbuchungen, aber auch für den Business-Bereich bei dem

Datensicherheit, Ausfallsicherheit und leichte Skalierbarkeit sehr groß geschrieben wird, einsetzbar.

Ein weiterer Punkt für eine wachsende Einsetzbarkeit dieser Applikation ist das sich leicht bestehende Serverlogik von älteren Systemen mit wenig Aufwand zu Erweitern, dass sie ein REST-Interface zur Verfügung stellen und für eine JavaScript-Applikation erreichbar werden.

Diese REST-Schnittstellen können natürlich nicht nur für die Kommunikation mit JavaScript-Applikationen verwendet werden, sondern bieten auch eine gute standardisierte Basis für die Kommunikation mit anderen Servern und bietet somit ein universelles Interface zur Außenwelt für viele Anwendungsfälle.

Anhang A

Inhalt der CD-ROM/DVD

A.1 PDF-Dateien

Pfad: /

Mensing-Braun_Martin_2015.pdf Masterarbeit (Gesamtdokument)

A.2 Literatur

Pfad: /Literatur

*.pdf . . . . . Kopien der verwendeten Literatur. PDFs sind immer nach dem Namen der Arbeiten benannt.

A.3 Literatur-Online

Pfad: /Literatur-Online

*.html . . . . . Kopien der verwendeten Online-Quellen.

A.4 Projekt

Pfad: /Projekt

instructions.txt . . . . . Installations-Anleitung für Frontend und Backend

frontend . . . . . Quellcode der TransactionManager-API in JavaScript.

backend . . . . . Quellcode des Beispiel-Backends in Grails.

backend/backend.war . Deploy-fähiges Packet für einen Tomcat-Server.

A.5 Abbildungen

Pfad: /Abbildungen

*.png . . . . . In der Arbeit verwendete Grafiken.

Quellenverzeichnis

Literatur

[1] Roy Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. Diss. Irvine: University of California, 2000 (siehe S. 3, 5, 6).

[2] Nam Giang, Minkeun Ha und Daeyoung Kim. „Cross Domain Communication in the Web of Things: A New Context for the Old Problem“. In: *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*. Seoul, Korea, 2014, S. 135–138 (siehe S. 8).

[3] Sebastian Kochman, PawełT. Wojciechowski und Miłosz Kmieciak. „Batched Transactions for RESTful Web Services“. English. In: *Current Trends in Web Engineering*. Bd. 7059. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 86–98 (siehe S. 14, 15).

[4] Peter Ledbrook und Glen Smith. *Grails in Action*. Greenwich, CT, USA: Manning Publications Co., 2014 (siehe S. 4).

[5] Luiz Alexandre Hiane da Silva Maciel und Celso Massaki Hirata. „Fault-tolerant timestamp-based two-phase commit protocol for RESTful services“. In: *Software: Practice and Experience* 43.12 (2013), S. 1459–1488 (siehe S. 18).

[6] A Marinos u. a. „RETRO: A Consistent and Recoverable RESTful Transaction Model“. In: *IEEE International Conference on Web Services (ICWS 2009)*. Los Angeles, CA, USA, 2009, S. 181–188 (siehe S. 17).

[7] Nandana Mihindukulasooriya, Miguel Esteban-Gutiérrez und Raúl García-Castro. „Seven Challenges for RESTful Transaction Models“. In: *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*. Seoul, Korea, 2014, S. 949–952 (siehe S. 14, 15).

[8] Guy Pardon und Cesare Pautasso. „Atomic Distributed Transactions: a RESTful Design“. In: *5th International Workshop on Web APIs and RESTful Design*. Seoul, Korea: ACM, Apr. 2014 (siehe S. 19, 20).

[9] Leonard Richardson und Sam Ruby. *Restful Web Services*. O'Reilly, 2007 (siehe S. 4, 16).

[10] Luiz Alexandre Hiane da Silva Maciel und Celso Massaki Hirata. „A Timestamp-based Two Phase Commit Protocol for Web Services Using Rest Architectural Style". In: *J. Web Eng.* 9.3 (Sep. 2010), S. 266–282 (siehe S. 18).

[11] Luiz Alexandre Hiane da Silva Maciel und Celso Massaki Hirata. „An Optimistic Technique for Transactions Control Using REST Architectural Style". In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. Honolulu, Hawaii, 2009, S. 664–669 (siehe S. 16, 17).

Online-Quellen

[12] Service Architecture. URL: http://www.service-architecture.com/articles/web-services/representational_state_transfer_rest.html (besucht am 25. 04. 2015) (siehe S. 4).

[13] Atomikos. *Try-Cancel/Confirm*. 2015. URL: http://www.atomikos.com/Publications/TryCancelConfirm (siehe S. 19).

[14] The RESTful CookBook. URL: http://restcookbook.com/HTTP%20Methods/patch/ (besucht am 25. 04. 2015) (siehe S. 4).

[15] Architects dZone. URL: http://architects.dzone.com/news/common-rest-design-pattern (besucht am 25. 04. 2015) (siehe S. 4, 15, 16).

[16] Mozilla. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS (besucht am 25. 04. 2015) (siehe S. 8).

[17] MSDN. URL: https://msdn.microsoft.com/en-us/library/aa560115(d=printer).aspx (besucht am 25. 04. 2015) (siehe S. 13, 14).

[18] PromisJS. URL: https://www.promisejs.org/ (besucht am 25. 04. 2015) (siehe S. 9, 10).

[19] WhatIsREST. URL: http://whatisrest.com/restconstraints/index (besucht am 25. 04. 2015) (siehe S. 5–7).

[20] Wikipedia. URL: https://en.wikipedia.org/wiki/Same-origin_policy (besucht am 25. 04. 2015) (siehe S. 8).

[21] Wikipedia. URL: https://en.wikipedia.org/wiki/JSONP (besucht am 25. 04. 2015) (siehe S. 8, 9).

[22] Wikipedia. URL: http://en.wikipedia.org/wiki/Linearizability (besucht am 25. 04. 2015) (siehe S. 13, 14).

[23] Wikipedia. URL: https://en.wikipedia.org/wiki/ACID (besucht am 25. 04. 2015) (siehe S. 14).