PirateBayTours

Projekt im Modul Verteilte

Informationsysteme WS2016/17

Jesko Appelfeller

Robin Naundorf

Frederik Broer

Jonas Droste

betreut durch

Prof. Dr.-Ing. Thomas Christian Weik

30. April 2017

Zusammenfassung

Abstrakt schreiben

Inhaltsverzeichnis

1 Motivation und Anforderungen **3**

2 Lösungskonzept **4**

2.1 Architektur 4

2.2 Backend 4

2.2.1 Django und Admin Interface 4

2.2.2 Database 5

2.3 Client 6

2.3.1 Java Client 6

2.3.2 Local Cache DB 10

2.4 Replizierung 11

2.4.1 Lesende Replizierung 11

2.4.2 Schreibende Replizierung 11

3 Geschäftslogik **14**

3.1 Buchungslogik 14

3.2 Replikationszeitpunkt 14

4 Fazit **16**

4.1 Replikation 16

Kapitel 1

Motivation und Anforderungen

Im Rahmen des Moduls 'Verteilte Informationssysteme' an der Fachhochschule Münster wird ein Semester begleitendes Datenbank Projekt durchgeführt. Ziel des Projektes ist es, ein Buchungssystem für Bootstouren zu implementieren. Die Firma Pirate-Bay-Tours beschäftigt mehrere Vertriebsmitarbeiter, die an Touristen Hotspots Tickets für Bootstouren vertreiben. Für diese Mitarbeiter soll ein neues Buchungssystem implementiert werden.
Wichtigstes Funktionsmerkmal des Systems ist die Offline-Fähigkeit. Das heißt, auch wenn der Mitarbeiter keine Verbindung zum zentralen Buchungsserver hat, muss die Buchung von Touren möglich sein. Dazu soll eine Replikationslogik entwickelt werden, die offline getätigte Buchungen bei verfügbarer Online-Verbindung mit dem Server synchronisiert. Weiterhin soll ein Quotensystem entwickelt werden, mit dem es möglich ist, verfügbare Tourtickets auf unterschiedliche Agenten zu verteilen. Zusätzlich muss eine Strategie entwickelt werden, wie mit Überbuchungen umgegangen werden soll.

Motivation des Projektes
Was waren seine Anforderungen?

Kapitel 2

Lösungskonzept

2.1 Architektur

Das Projekt PirateBayTours besteht aus einer Client-Server-Architektur. Die Agenten arbeiten mit eine Client auf Java-Basis. Eine Synchronisierung mit dem Server erfolgt mittels HTTP-Rest aufrufen. Der Server basiert auf Python 3[1] und dem Django Web-Framework[2] . Für die Implementierung der Rest-API wurde das Paket *django-rest-framework*[3] verwendet. Für das Load-Balancing der Postgres-Instanzen wurde ein Software-Loadbalancer auf Basis von VMware NSX[4] verwendet. Eine schematische Darstellung findet sich in Abbildung 2.1

2.2 Backend

2.2.1 Django und Admin Interface

Für das Backend wurde Django in der Version 1.10.5 verwendet. Django bietet als Framework unter anderem einen Object-Relational-Mapper (kurz: ORM) mit der bei der Abstrahierung der Datenbank-Schicht hilft. Django basiert auf dem MVC-Prinzip. Hierdurch ist es möglich in der Modell-Schicht ein Modell zu implementieren, dass dann mittels ORM auf die als Schema in der Datenbank abgelegt wird. Das Implementierte Modell ist in Abbildung 2.2 zu sehen.
Mit Hilfe des Django-Rest-Frameworks wird der Zugriff auf das Backend mittels HTTP-Rest-Aufrufen realisiert. Es ist z.B. möglich alle vorhanden Touren über einen

[1] https://www.python.org
[2] https://www.djangoproject.com
[3] https://www.django-rest-framework.org
[4] https://www.vmware.com/products/nsx.html

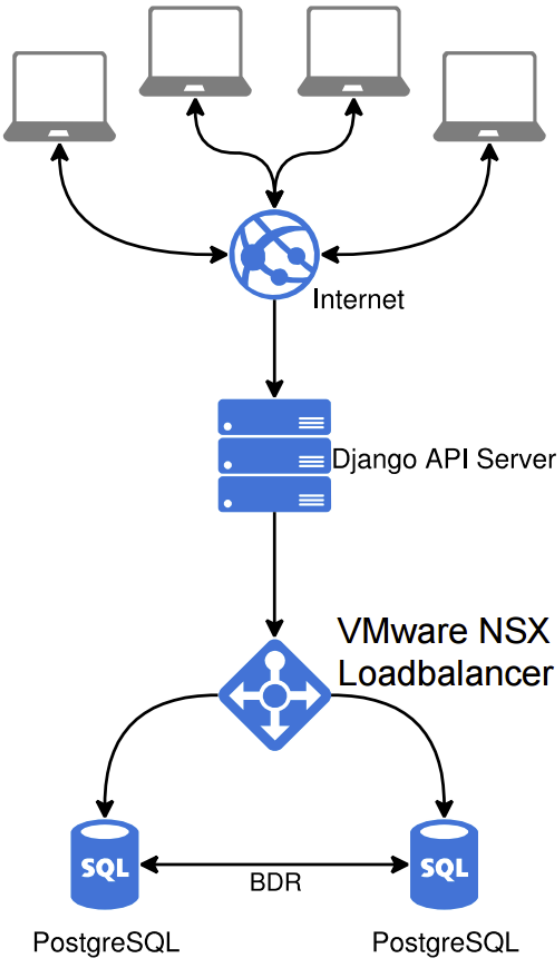

Abbildung 2.1: Architektur

HTTP-GET-Aufruf an http://SERVER/api/tours zu bekommen. Der Server liefert die angeforderten Daten im JSON-Format[5] zurück.
Für die Administration der Datenbestände (z.B. Hinzufügen neuer Touren/Schiffe) bietet Django ein integriertes Admin-Backend mit Authentifizierung über Benutzername/Passwort.

2.2.2 Database

Die Datenbank für das Projekt PirateBayTours basiert auf PostgreSQL. Die Replizierung der Datenbank serverseitig erfolgt mittels einer angepassten PostgreSQL-Version zur bidirektionalen Replikation *Postgres-BDR* der Firma 2ndQuandrant. Für die Hochverfügbarkeit der Datenbank wurden 2 Server mit Debian-Linux aufgesetzt auf denen jeweils eine Postgres-Datenbank in der BDR-Anpassung von 2nd-

[5]http://www.json.org/

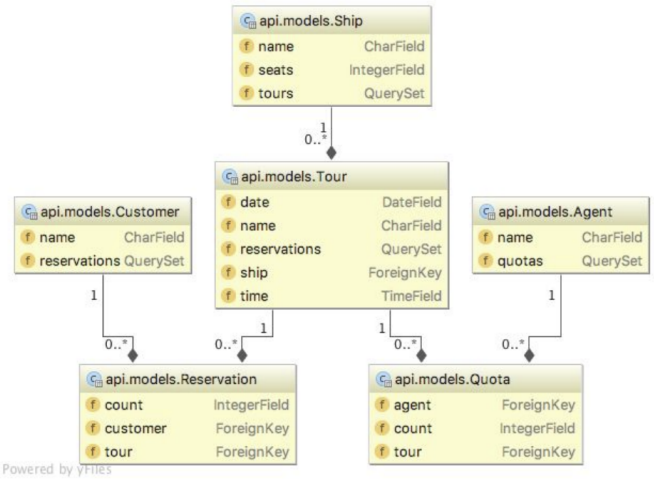

Abbildung 2.2: Django Modell

Quadrant läuft. Diese beiden Datenbanken synchronisieren sich ständig und sind beide sowohl lesend als auch schreibend benutzbar (sogenannte Multi-Master-Replikation).

Replikation

Eine vorgefertigte Replikationslösung zwischen SQLite und PostgreSQL gibt es nicht. Daher wird die Replikation in Eigenentwicklung umgesetzt. Für Details siehe Abschnitt 2.4.

2.3 Client

2.3.1 Java Client

Implementierungsmodell

Wir haben uns dazu entschieden den Client in der Programmiersprache Java zu schreiben, dadurch konnte unsere gesamte Gruppe gemeinsam an den Implementierungen für den Client arbeiten, weil Java eine Standartprogrammiersprache ist, die jeder aus unserer Gruppe beherrschte. Außerdem ist Java sehr praktikabel für die Implementierung einer Oberfläche, wie wir sie für den Client gebraucht haben.

Für die Implementierung haben wir das „Model-View-Controller“-Modell (MVC-Modell) verwendet, das sich für die Erledigung unserer Aufgabe sehr geeignet war. Das MVC-Modell teilt die Implementierung in drei unterschiedliche Komponenten ein:

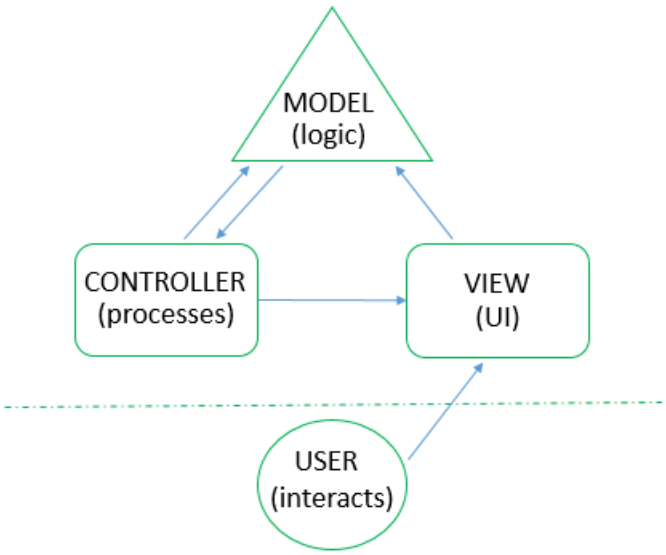

Abbildung 2.3: Darstellung des MVC Modells

- das Datenmodell (engl. model),
- die Präsentation (engl. view) und
- die Programmsteuerung (engl. controller).

Deswegen benötigen wir drei unterschiedliche Klasse, die in Abbildung 2.3[6] dargestellt sind. In der Klasse „model“ wurden daher die darzustellenden Daten gespeichert, wie beispielsweise der Routenname, in der die ausgewählte Route zwischengespeichert wird. In der Klasse „view“ wurden die benötigten Daten aus dem „model“ dargestellt und die Benutzerinteraktionen entgegengenommen. In der Klasse „controller“ wurde die Steuerung Implementiert, die für die Bearbeitung der Benutzerinteraktionen zuständig ist. Zusätzlich zu diesen drei Klassen des MVC-Modells, haben wir eine Klasse für die API-Zugriffe und eine Klasse für die SQLite Cache Datenbank Zugriffe erstellt.

Struktur

Nachdem im letzten Kapitel das Implementierungsmodell dargestellt wurde, folgt nun die Beschreibung der Struktur des Clients. Der Client ist wie in Abbildung 2.4 dargestellt. Diese sehr simple Darstellung der Oberflächenelemente und die dadurch entstehende hohe Benutzerfreundlichkeit ist ein großer Vorteil unseres Clients. Die einfache Bedienbarkeit entsteht durch die Bedienung mit Maus und Tastatur, wobei die Anwendung nur mit der Tastatur bedient werden kann. Dadurch können beispielsweise viele Tickets in kürzester Zeit verkauft werden.

[6] http://docs.sitefinity.com/sf-images/default-source/default-album/mvc.png

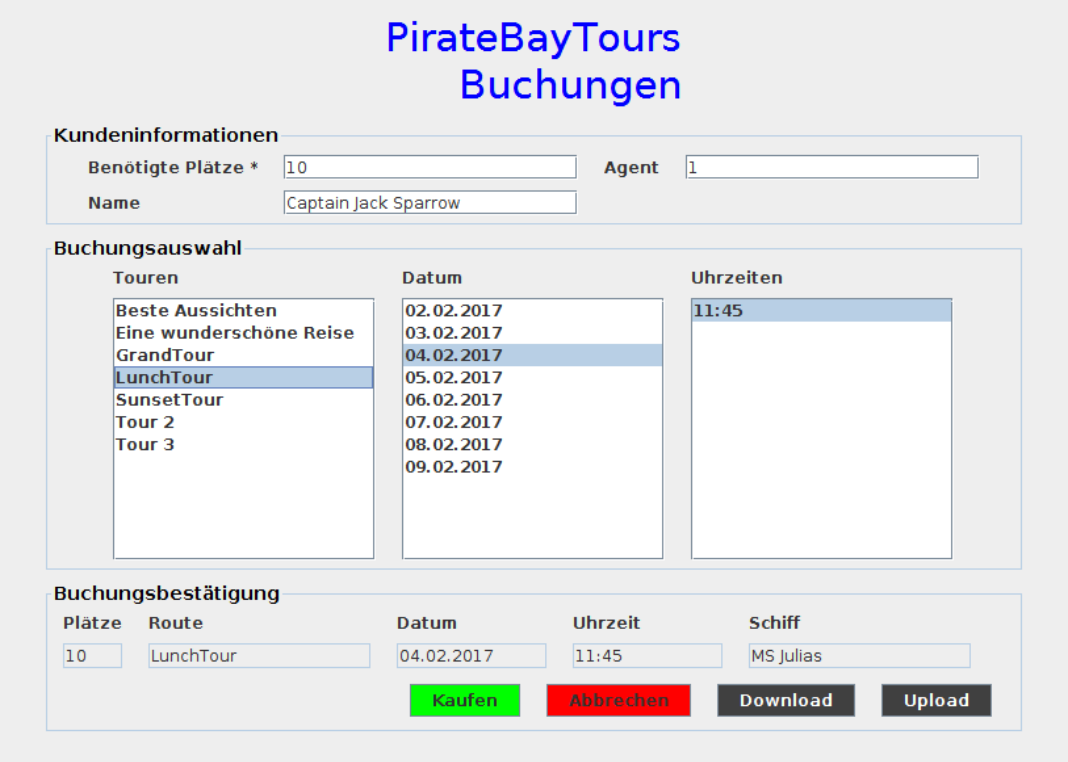

Abbildung 2.4: Darstellung des Clients

Beim Start der Anwendung müssen zunächst die Daten, also die bisher nicht verkauften Tickets sowie alle anderen notwendigen Daten aus den Datenbanken, vom Server geladen werden. Dafür wird die Schaltfläche „Download“ angeklickt. Anschließend liegen die aktuellen Daten in der lokalen Datenbank und können verwendet und bearbeitet werden. In das Textfeld „Agent“ muss die Personalnummer des jeweiligen Vertriebsmitarbeiters eingetragen werden.
Damit der Tickets für eine bestimmte Route gebucht werden können, müssen folgende Schritte nacheinander durchgeführt werden.

1. Die Anzahl der benötigten Tickets sowie der Name des Käufers müssen in die entsprechenden Textfelder eingetragen werden. Anschließend steht im Textfeld „Plätze“ der eingetragene Wert.

2. Nach der Bestätigung mit der „Enter“-Taste werden die Touren angezeigt, für die noch genügend Plätze vorhanden sind.

3. Sobald eine Tour, durch die Selektion einer Zeile, ausgewählt wurde, wird die ausgewählte Route im Textfeld „Route“ angegeben.

4. Nun werden in der Liste „Datum“ alle möglichen Datums angegeben, an denen die Tour gebucht werden kann.

5. Mit der Auswahl eines Datum und der zugehörigen Selektion einer Zeile wird im Textfeld „Datum“ das gewählte Datum angegeben und in der Liste „Uhrzeiten“ werden die möglichen Uhrzeiten für die gewählte Tour sowie Datum angezeigt.

6. Erst mit der Auswahl einer Uhrzeit kann ein Ticket, bzw. mehrere Tickets gebucht werden, weil erst dann die Schaltfläche „Kaufen“ anklickbar ist.

Mit der Schaltfläche „Abbrechen“ kann die Buchung jederzeit beendet werden und alle Textfelder, bis auf das Textfeld „Agent“, werden zurückgesetzt. Nachdem der Arbeitstag beendet ist, oder der Vertriebsmitarbeiter zwischenzeitlich eine Internetverbindung aufnehmen kann, können die bisher verkauften Tickets mit dem Server synchronisiert werden. Dafür muss lediglich die Schaltfläche „Upload“ angeklickt werden.

2.3.2 Local Cache DB

Um offline einsatzfähig zu sein, wurde eine lokale SQLite Datenbank genutzt. Während der Synchronisierung vom Server zum Client wird ein Abbild der Serverdatenbank heruntergeladen und in der lokalen Datenbank gecached. Wir haben uns entschieden eine SQLite Datenbank zu nutzen, da keine zusätzlich Datenbankserver installiert werden müssen und alle notwendigen Ressourcen gut in die Applikation eingebunden werden können. Weiterhin wird SQLite von einer Vielzahl an Programmiersprachen gut unterstützt und bietet eine ressourcensparende Möglichkeit Daten abzulegen.

Wie erwähnt, enthält die Datenbank nach einmalige Synchronisation ein komplett Abbild der Server-Datenbank. Das heißt, lokal sind folgende Tabellen verfügbar:

- `agents`
- `customers`
- `quotas`
- `ships`
- `tours`

Zusätzlich dazu werden zwei lokale Tabellen angelegt.

- `offline_bookings`
- `offline_customers`

Die `offline_bookings` Tabelle enthält alle Buchungen, die der Vertriebsmitarbeiter im Offline-Fall tätigt. Werden im Offline-Modus neue Kundenstammdaten angelegt werden diese in der Tabelle `offline_customers` gespeichert. Dies ist notwendig, um ID Konflikte bei der Replizierung zu vermeiden. Weiterhin lässt sich so mit wenig Aufwand nachvollziehen welche Buchungen noch nicht repliziert wurden. Da das Serverabbild bei einer Offline-Buchung nicht verändert wird, besteht kein Risiko, das es zu inkonsistenten Ausgangsdaten kommt und ein Offline-Arbeiten nicht mehr möglich ist.

2.4 Replizierung

Wie in Abschnitt 3.2 beschrieben, gibt es zwei Replikationsprozesse. Zum einen lesend vom Server zur lokalen DB, zum anderen schreibend, von der lokalen DB zum Server.

2.4.1 Lesende Replizierung

Der lesende Replikationsvorgang ist trivial. Die Inhalte der Tabellen `agents, customers, quotas, ships` und `tours` werden per HTTP bei der Django-API angefragt und von dieser als JSON zurückgegeben. Clientseitig werden diese JSON-Strings geparst und in die lokale DB eingefügt. Vergleiche hierzu Kapitel 2.2.1

2.4.2 Schreibende Replizierung

Bei der schreibenden Replizierung sollen die Eintrage der Tabellen `offline_customers` und `offline_bookings` aus der lokalen Datenbank an die Tabellen `customers` und `bookings` der zentralen Datenbank übertragen werden. Dies geschiecht, analog zum lesenden Zugriff, durch Umformung der Einträge in JSON-Objekte welche dann mittels HTTP-Requests an eine Django-API geschickt werden.

Hierbei ergibt sich die Schwierigkeit, dass die Einträge in der lokalen DB einen automatisch eine ID bekommen haben, die jedoch nur lokal gültig. Bei der Übertragung werden neue, dort gültige IDs für jeden Eintrag vergeben. Dies bedeutet jedoch, dass die Verweise aus der `offline_bookings` Tabelle auf IDs in der `offline_customers` Tabelle serverseitig nicht mehr gültig sind. Vergleiche hierzu Kapitel 2.2.1 Abbildung 2.2
Es müssen also vor dem Upload der Einträge aus der Tabelle `offline_bookings` die Verweise auf die lokalen IDs in der Tabelle `offline_customers` ersetzt werden durch

Verweise auf die zentralen IDs in der Tabelle `customers`. Diese werden aber vom Server erst vergeben, wenn die Einträge aus der `offline_customers` zum Server gesendet werden. Die bereits erwähnte Django-API beantwortet jeden erfolgreichen Schreibzugriff mit dem Versand des erzeugten Eintrags als JSON-Objekt. Diese Objekte enthalten die vom Server vergebene ID die somit direkt in den entsprechenden Datensatz der lokalen `offline_customers` eingetragen werden kann.

Es werden jedoch nicht die lokalen IDs überschrieben, sondern die zentralen IDs hinzugefügt. Dies ermöglicht es in einem zweiten Schritt für jeden Eintrag der Tabelle `offline_bookings` den Verweis von der lokalen auf die zentrale ID anzupassen und den Eintrag dann zum Server zu senden.

Nach Abschluss der Replikation werden die Einträge der Tabellen `offline_bookings` und `offline_customers` nicht mehr benötigt. Beide Tabellen werden gelöscht und neu angelegt, um Duplikate sicher auszuschließen. Der gesamte Vorgang wird in Abbildung 2.5 graphisch dargestellt.

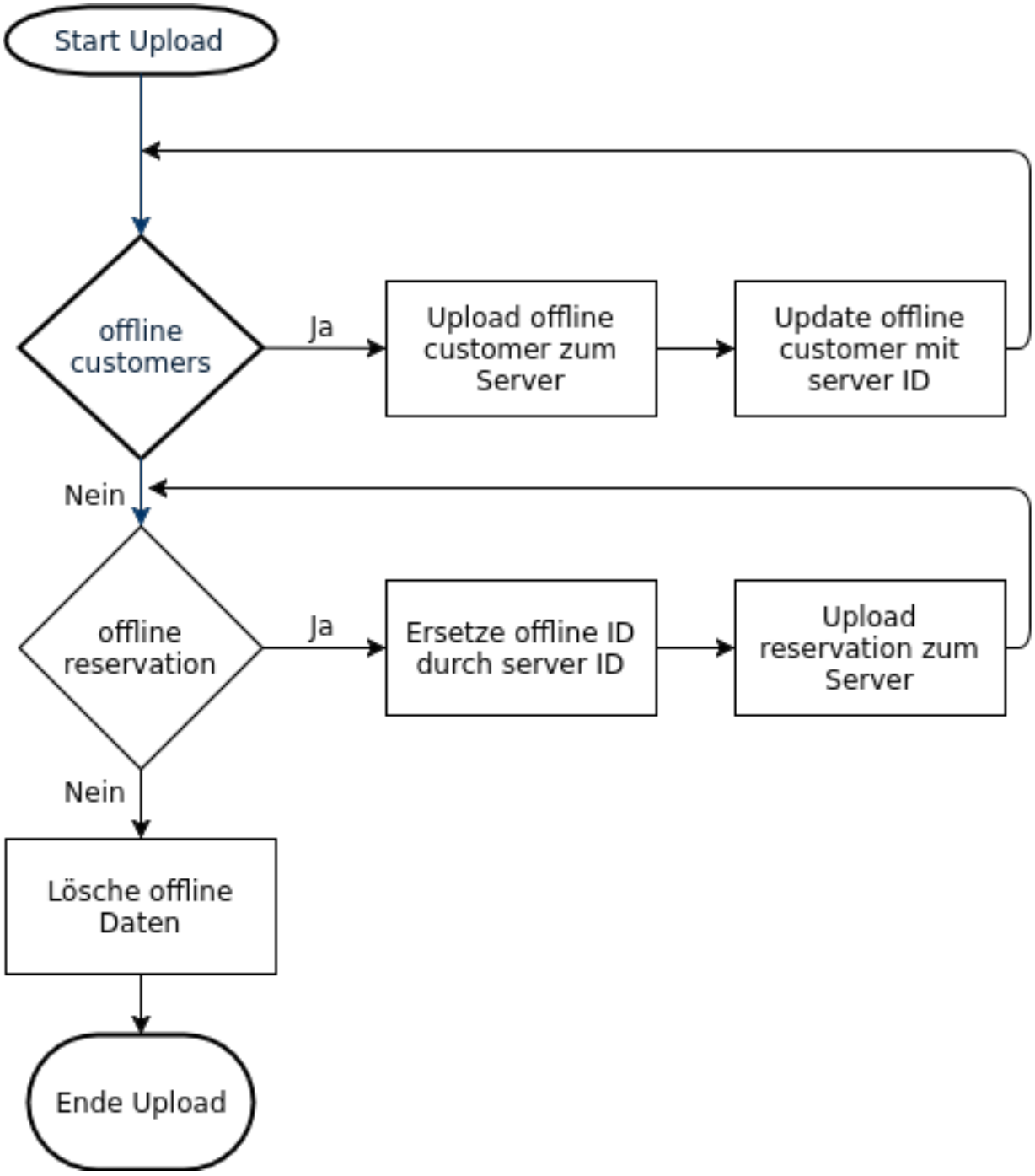

Abbildung 2.5: Ablaufdiagramm schreibende Replikation

Kapitel 3

Geschäftslogik

Wie werden Fälle wie Überbuchen gehandelt?

3.1 Buchungslogik

Ein entscheidender Punkt in der Geschäftslodig einer Anwendung wie der piratebaytours ist der Bereich der Buchungslogik. Da die Angenten die meiste Zeit offline arbeiten, muss ein Weg gefunden werden einerseits den Angenten den Spielraum zu geben um auch größeren Gruppen Angebote machen zu können. Andererseits sollten Überbuchungen vermieden werden, da diese zu Verstimmungen bei den Kunden sowie Rück- oder gar Entschädigungszahlungen für das Unternehmen führen.

Es müssen also die vorhandenen Plätze auf jedem Schiff und jeder Route auf die Agenten verteilt werden, in solch einer Weise dass das Kontingent dem einzelnen Agenten möglichst reicht. Unser Lösungsansatz besteht darin, Überbuchungen generell nicht zuzulassen. Stattdessen berechnen wir jede Nacht die Kontingente pro Tour und Agent für die kommenden Tage neu, um somit erfolgreichen Agenten größere Möglichkeiten zu bieten. Dies hat außerdem den Effekt, dass erfolgreiche Agenten für ihre Arbeit belohnt werden, während der negative Einfluss von weniger erfolgreichen Agenten auf das Gesamtergebnis des Unternehmens minimiert wird.

3.2 Replikationszeitpunkt

Auf Grund der begrenzten Verfügbarkeit von mobilen Internet für unsere mobile Anwendung, wird im Normalfalle nur zu bestimmten Zeitpunkten eine Replizierung von der zentralen Datenbank zur lokalen Datenbank und zurück durchgeführt. Bei-

der Vorgänge werden manuell vom Benutzer ausgelöst. Bei erhöhter Verfügbarkeit des Internets steht es diesem frei häufiger zu replizieren. Minimal wird jedoch davon ausgegangen, dass die Agenten zu Beginn und Ende ihres Arbeitstages Internetzugang haben und folgende Replizierungen durchführen können:

Muss eine Reihenfolge eingehalten werden?

Lesend/Morgens Zu Beginn des Arbeitstages muss jeder Agent einen Lesevorgang auslösen. Dies geschieht über das GUI mittels der Schaltfläche *Download*. Hierbei werden die Tabellen `agents`, `customers`, `quotas`, `ships` und `tours` vom Server heruntergeladen und in die lokale DB eingelesen. Somit verfügt der Agent nach Abschluss des Vorgangs über alle Touren und Schiffe und kann mittels der bereits vorhandenen Buchungen ausrechnen welche Pältze noch frei sind. Er kennt außerdem seine persönlichen Quoten und weiß daher wie viele Pätze er maximal verkaufen kann.

Schreibend/Abends Am Ende des Arbeitstages muss jeder Agent einen Schreibvorgang auslösen. Dies geschieht über das GUI mittels der Schaltfläche *Upload*. Hierbei werden die neu angelegten Kunden und Buchungen aus den Tabellen `offline_bookings` und `offline_customers` der lokalen DB in die Tabellen `bookings` und `customers` der zentralen DB hochgeladen. Der Uploadmechanismus ist in Abschnitt 2.4 beschrieben. Nach Abschluss dieses Vorgangs durch alle Agenten sind dem Server alle neuen Kunden und Buchungen bekannt.

Kapitel 4

Fazit

Was sind die Lessons Learned
Was könnte man das nächste mal besser machen?
Waren unsere ausgewählten Komponenten für das Problem geeignet?

4.1 Replikation

Unsere Lösung setzt clientseitig mit SQLite eine andere Datenbank ein als serverseitig mit PostgreSQL. Dies vereinfacht sowohl die Entwicklung als auch das Deployment des Client, da keine separate DB aufgesetzt und angesprochen werden muss sondern alles in einer Datei und mit einem Treiber funktioniert. Dies ist insbesonder von Vorteil wenn viele verschiedene Endgeräte eingesetzt werden, da die von unserem Client eingesetzte Kombination von SQLite und Java auf sehr vielen Plattformen verfügbar ist.

Bei der Replikation entstehen durch diesen Ansatz jedoch Probleme, da es für diese Kombination aus Datenbanken keine vorgefertigte Replikationslösung gibt. Auf dem Weg vom Server zum Client verursacht dies keinen großen Mehraufwand zumindest in der Entwicklung. Die Umwandlung zwischen JSON und der jeweiligen Datenbank dürfte jedoch Performanznachteile mit sich bringen.
Lesend werden die verschiedenen Datenbanken jedoch zu einem größeren Problem. Wie in Unterabschnitt 2.4.2 beschrieben, ist ein in der Entwicklung aufwendiger Mechanismus zur Synchronisierung der IDs nötig. Weiterhin wird jeder Datensatz einzeln übertragen, was Performanznachteile mit sich bringt.

Ob diese Nachteile durch den Vorteil des einfacheren Setups aufgewogen werden, hängt davon ab wie oft in der Praxis ein Client neu eingerichtet wird. Bei den im

Szenario vorkommenden “fliegenden” Händlern, kann sowohl mit einer recht hohen Fluktuation als auch mit einer heterogenen Gerätelandschaft gerechnet werden, so dass sich der Einsatz von SQLite rechnen sollte. Zusamenfassend ist zu sagen, dass sich die gewählten Komponenten für den Einsatzzweck eignen.