PirateBayTours

Projekt im Modul Verteilte

Informationsysteme WS2016/17

Jesko Appelfeller

Robin Naundorf

Frederik Broer

Jonas Droste

betreut durch

Prof. Dr.-Ing. Thomas Christian Weik

11. April 2017

Zusammenfassung

Abstrakt schreiben

Abstract

Abstrakt übersetzen

Inhaltsverzeichnis

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

Aus welchen Komponenten besteht die Software?
Was sind die Schnittstellen?

2.2 Backend

2.2.1 Django und Admin Interface

Was bieted Django alles von Hausaus?

2.2.2 Database

Wie ist die Datenbank aufgebaut?
Welche Replikation setzen wir ein?
Wie ist diese Implementiert?

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

Welche Konzepte nutzt der Client? MVC
Wie ist die Struktur im Client?
Welche Klassen wurden implementiert?
Wie erfolgt die Datenhalt ung?

2.3.2 Local Cache DB

Name 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.

Verweise auf API Kapitel

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.

Verweis auf DB-Schema einfügen

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 2.1 graphisch dargestellt.

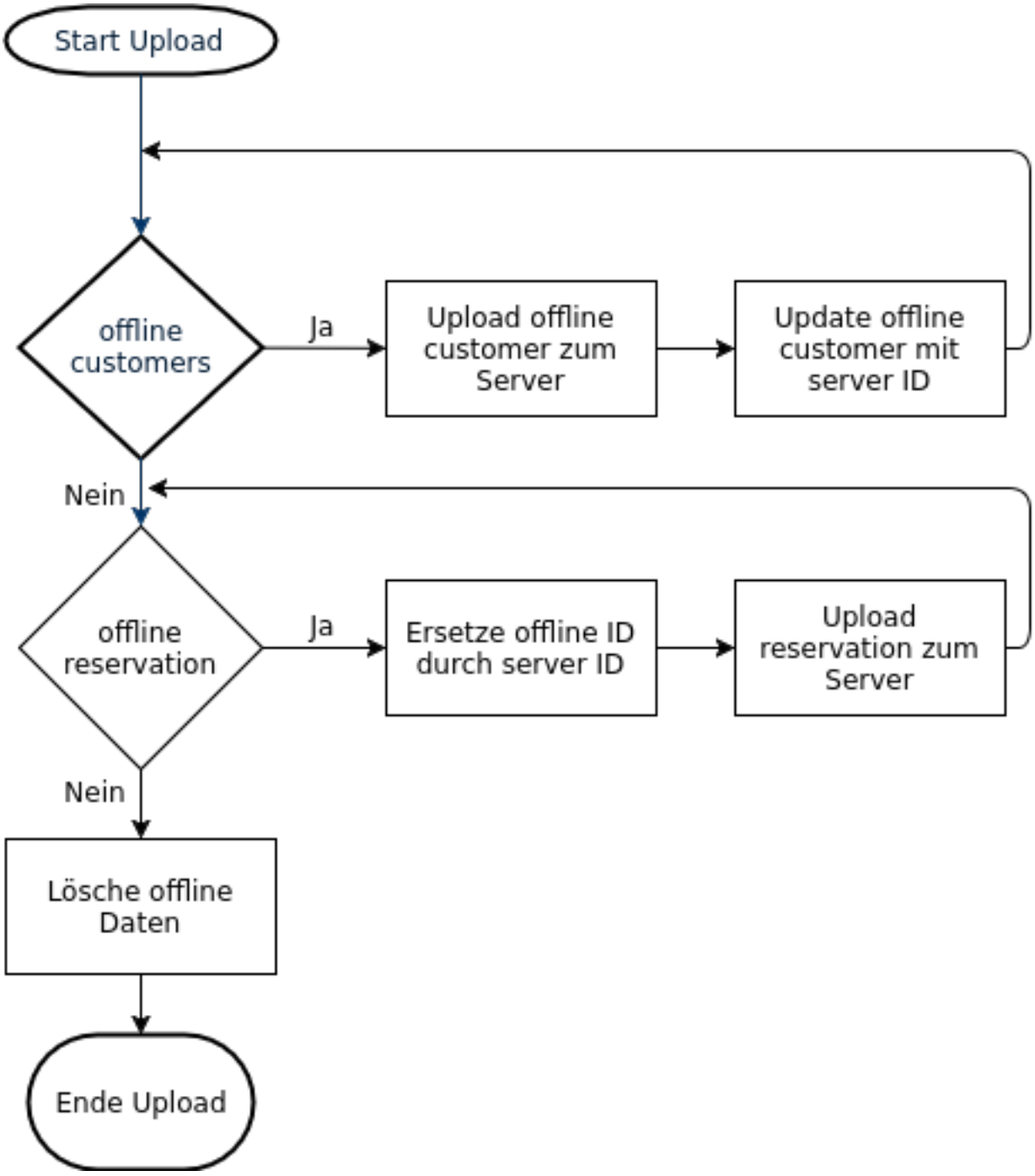

Abbildung 2.1: Ablaufdiagramm schreibende Replikation

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.