Location Based Recommendations
Ergebnisse

Henrik Gerdes, Johannes B. Latzel, Leon Richardt

16. Oktober 2018

Universität Osnabrück

Vorüberlegungen

Was brauchen wir?

- **Datenbank:** Speichert die Events
- **Server:** Entscheidet, welche Events ein bestimmter Nutzer zu sehen bekommt
- **App:** GUI für den User
- **Client-Server-Interface:** Kommunikation zwischen App und Server

Datenbank

Datenbank – Implementation

Die Datenbank läuft auf `MariaDB`, einer Open-Source-Alternative zu `MySQL`. Die Kommunikation zwischen der Datenbank und Java geschieht mit `JDBC` (*Java Database Connectivity*).

Datenbank – Implementation

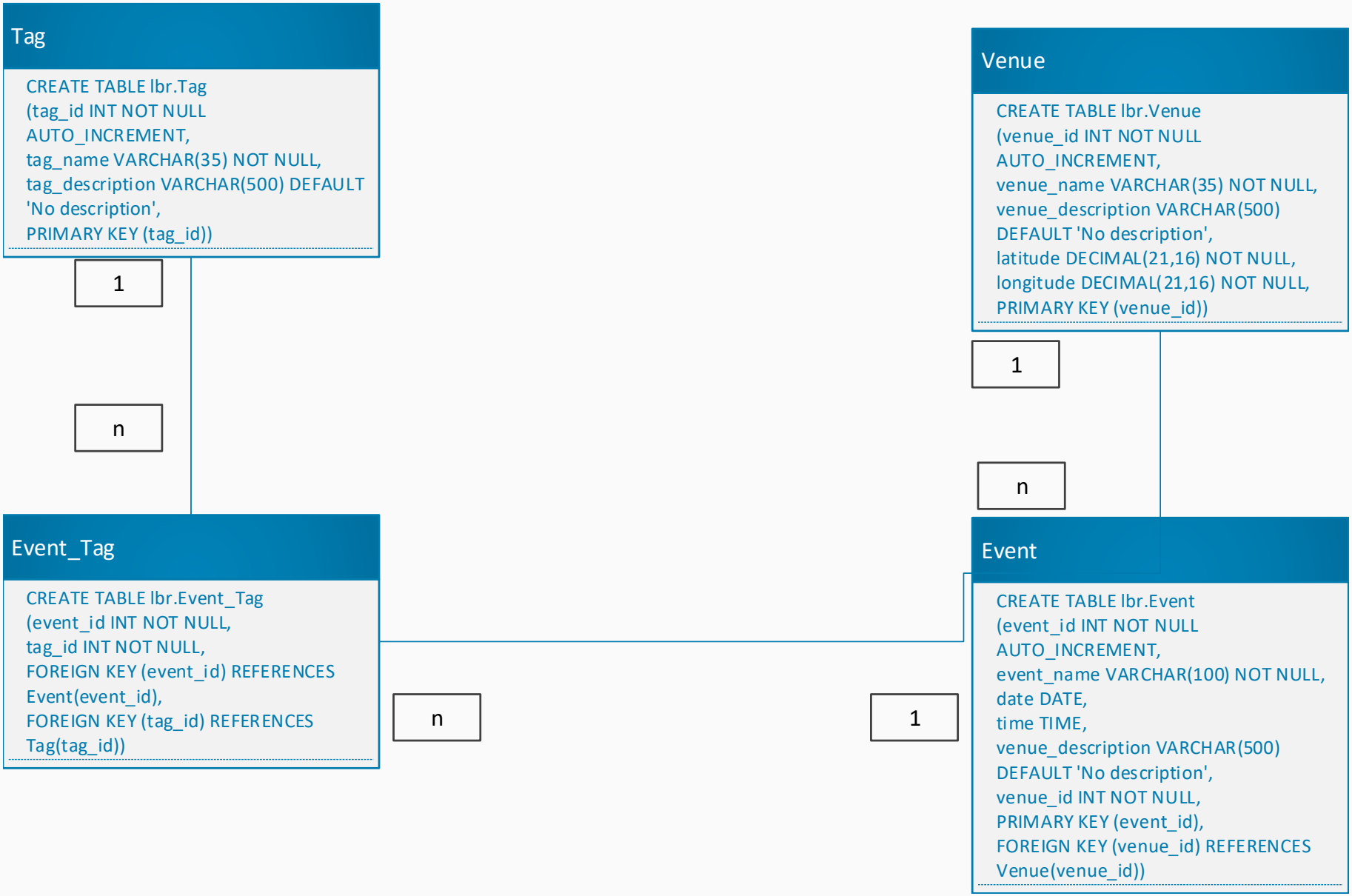
Die Datenbank läuft auf `MariaDB`, einer Open-Source-Alternative zu `MySQL`. Die Kommunikation zwischen der Datenbank und Java geschieht mit `JDBC` (*Java Database Connectivity*).

In der Datenbank gibt es je eine Table für:

- Events
- Venues
- Tags (*Kategorien, in die die Events eingeordnet werden*)

Außerdem gibt es eine weitere Table, die jedem Event seine Tags zuordnet.

Datenbank – Schematischer Aufbau



Server

Server – Hardware & Betriebssystem

Als Server wird ein `Raspberry Pi I B+` mit dem Betriebssystem `Raspbian` genutzt.

Der Server wartet auf Port `5445` auf neue Client-Verbindungen und verarbeitet diese. Da ein Thread-Pool mit vier Threads genutzt wird, können mehrere Verbindungen gleichzeitig akzeptiert werden.

Server – Hardware & Betriebssystem

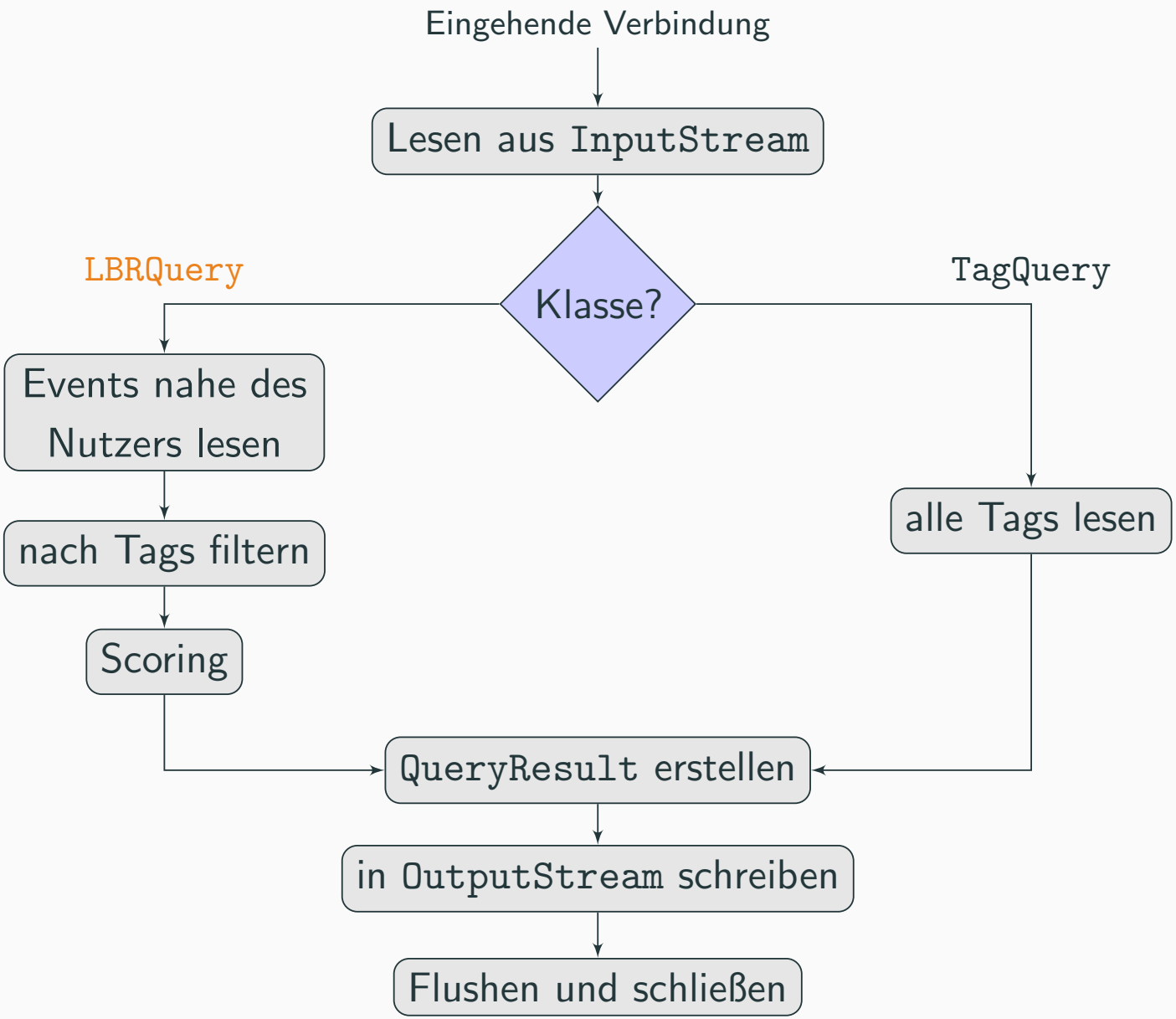
Als Server wird ein `Raspberry Pi I B+` mit dem Betriebssystem `Raspbian` genutzt.

Der Server wartet auf Port `5445` auf neue Client-Verbindungen und verarbeitet diese. Da ein Thread-Pool mit vier Threads genutzt wird, können mehrere Verbindungen gleichzeitig akzeptiert werden.

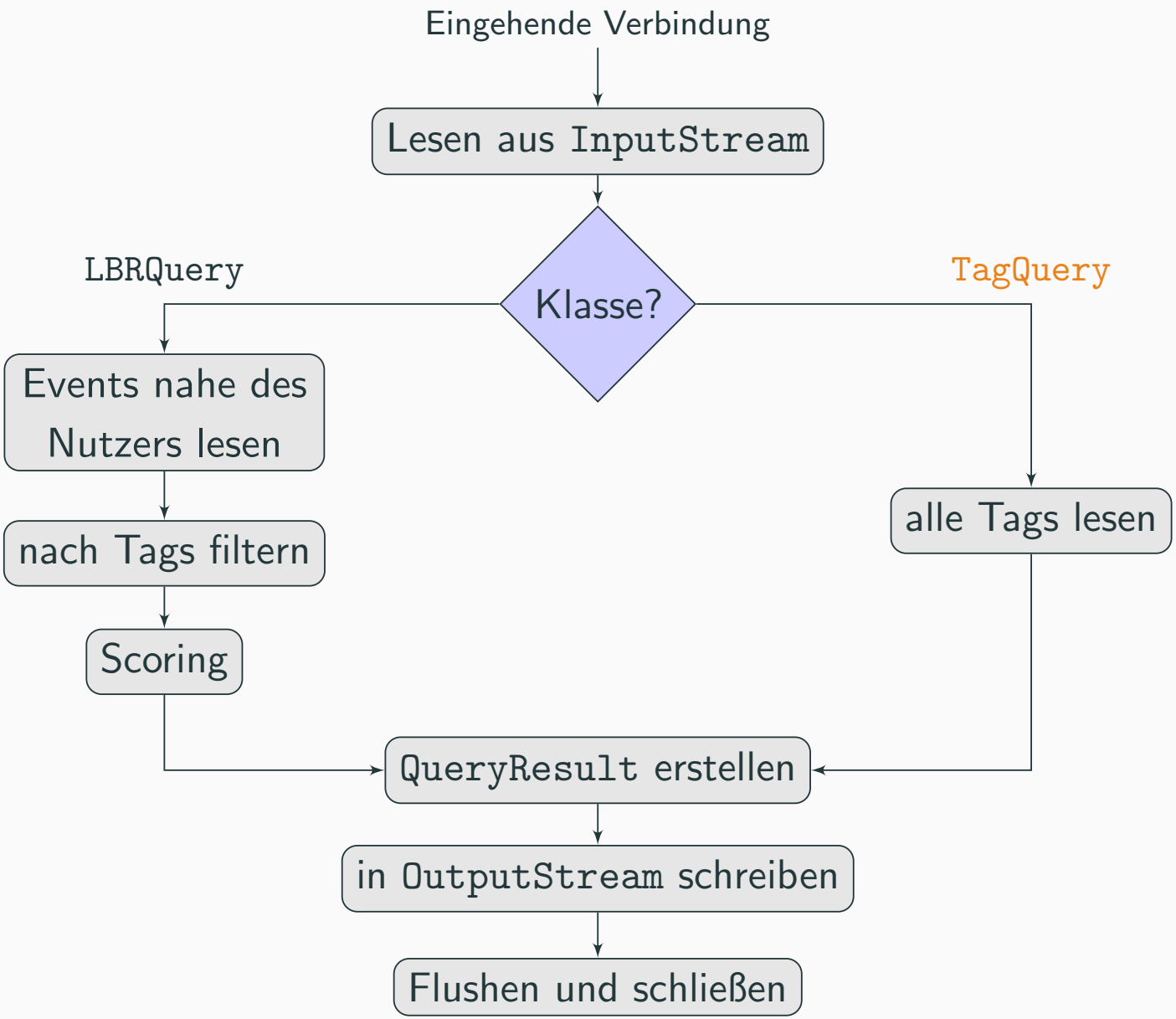
Kommt es während des Datenbank-Zugriffs zu einer `SQLException`, so wird automatisch ein Reconnect durchgeführt.

Das momentan verwendete Scoring ist zufallsbasiert, es ließe sich aber leicht ein anderes Scoring-System integrieren.

Server – Schematischer Ablauf



Server – Schematischer Ablauf



Client-Server-Interface

Client-Server-Interface – Klassen

Folgende Klassen sind besonders wichtig für die Kommunikation zwischen Server und App:

- `Venue` - Ein Ort, an dem Events stattfinden können
- `Event` - Ein Ereignis, wie z. B. ein *Konzert* oder ein *Festival*
- `Tag` - Eine Kategorie, wie z. B. *Tanzen* oder *Live-Musik*
- `Store` - Interface, um Tags und Events zwischenzuspeichern. Mit einem `StoreListener` kann auf Veränderungen komfortabel reagiert werden.

App

App – Funktionen I

- Der Nutzer landet beim Start auf einem Splash Screen und wird erst bei vorhandenen Standort-Berechtigungen weitergeleitet.
- Die App führt in regelmäßigen Abständen (15 Minuten) automatisch eine Aktualisierung der Events, die in der Nähe stattfinden, durch.

App – Funktionen II

- Die aktuelle Position des Users und nahe Events werden auf einer Karte angezeigt. Außerdem werden die Events in einer Liste aufgeführt.
- Für nahe Events wird ein Geofence registriert.
- Hält sich der User lange genug innerhalb eines Geofences auf, erhält er eine Benachrichtigung.

Herausforderungen

Herausforderungen bei der Implementation

- Einarbeitung in Android und in Kotlin
- Zuverlässige Netzwerk-Kommunikation zwischen App und Server (*Wie kriegen wir die Events vom Server zur App?*)
- Eigenheiten von Android, unter anderem:
  - Standortzugriff über den `FusedLocationProvider`
  - Regelmäßiges Fetchen von Tags und Events im Hintergrund über `JobService`
  - Unzuverlässigkeit der Geofencing-API
- Umfangreiche Frameworks, die viel Einarbeitung benötigen, wie zum Beispiel:
  - Android Job Scheduling
  - Geofencing

Herausforderungen bei der Implementation

- Einarbeitung in Android und in Kotlin
- Zuverlässige Netzwerk-Kommunikation zwischen App und Server (*Wie kriegen wir die Events vom Server zur App?*)
- Eigenheiten von Android, unter anderem:
  - Standortzugriff über den `FusedLocationProvider`
  - Regelmäßiges Fetchen von Tags und Events im Hintergrund über `JobService`
  - Unzuverlässigkeit der Geofencing-API
- Umfangreiche Frameworks, die viel Einarbeitung benötigen, wie zum Beispiel:
  - Android Job Scheduling
  - Geofencing

Herausforderungen bei der Implementation

- Einarbeitung in Android und in Kotlin
- Zuverlässige Netzwerk-Kommunikation zwischen App und Server (*Wie kriegen wir die Events vom Server zur App?*)
- Eigenheiten von Android, unter anderem:
  - Standortzugriff über den `FusedLocationProvider`
  - Regelmäßiges Fetchen von Tags und Events im Hintergrund über `JobService`
  - Unzuverlässigkeit der Geofencing-API
- Umfangreiche Frameworks, die viel Einarbeitung benötigen, wie zum Beispiel:
  - Android Job Scheduling
  - Geofencing

Herausforderungen bei der Implementation

- Einarbeitung in Android und in Kotlin
- Zuverlässige Netzwerk-Kommunikation zwischen App und Server (*Wie kriegen wir die Events vom Server zur App?*)
- Eigenheiten von Android, unter anderem:
  - Standortzugriff über den `FusedLocationProvider`
  - Regelmäßiges Fetchen von Tags und Events im Hintergrund über `JobService`
  - Unzuverlässigkeit der Geofencing-API
- Umfangreiche Frameworks, die viel Einarbeitung benötigen, wie zum Beispiel:
  - Android Job Scheduling
  - Geofencing

Screenshots

Screenshots I

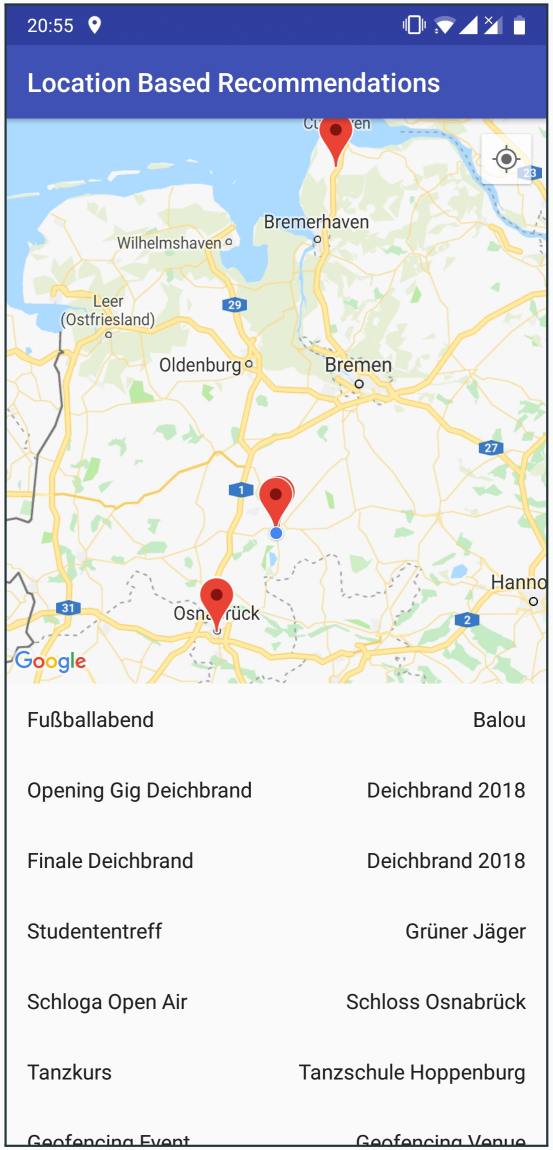


Abbildung 1: Karte & Liste

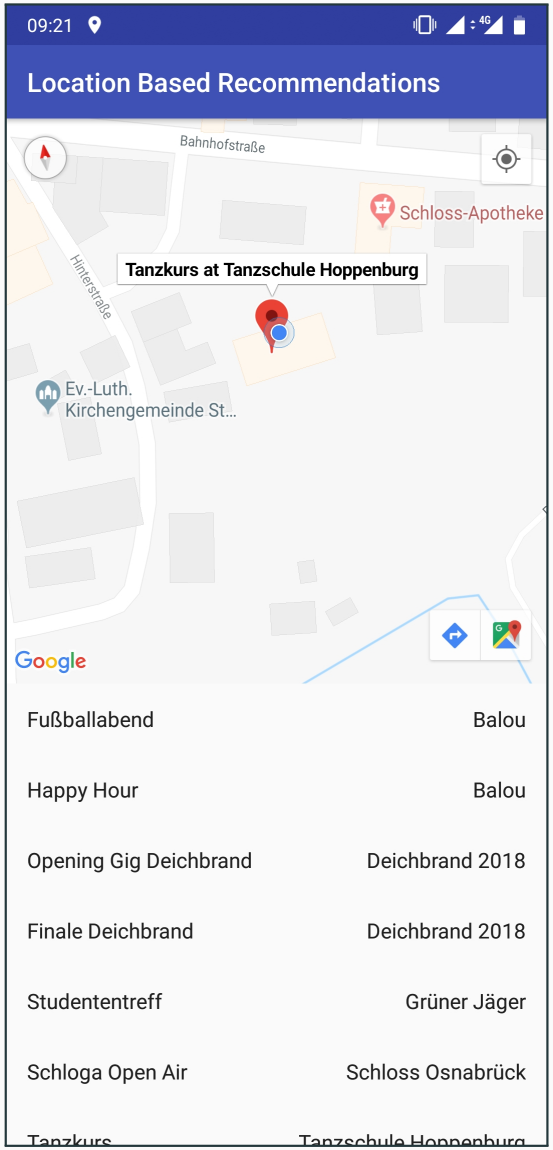


Abbildung 2: Marker auf Karte

Screenshots II

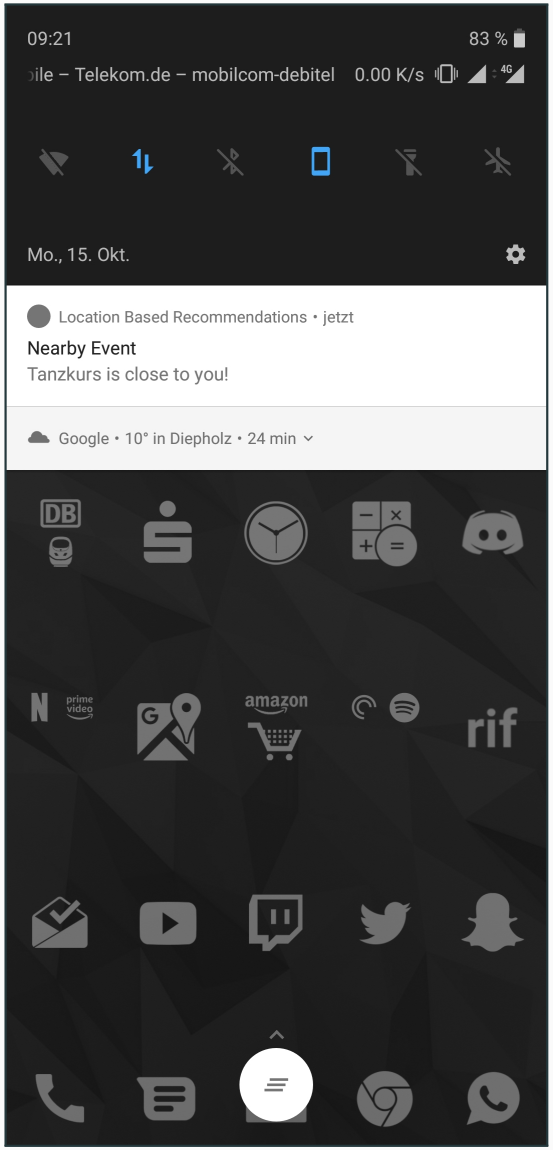


Abbildung 3: Geofence-Benachrichtigung

Abbildung 4: Event-Übersicht