Location Based Recommendations
Ergebnisse

Henrik Gerdes, Johannes B. Latzel, Leon Richardt

16. Oktober 2018

Universität Osnabrück

Rückblick

Projektziel

Must-Haves:

- Geordnete Liste passender Veranstaltungen
- Hervorhebung empfohlener Events auf der Karte

Nice-to-Have:

- Push-Nachrichten, wenn geeignete Veranstaltungen in der Nähe stattfinden

Vorüberlegungen

Was brauchen wir?

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

Datenbank

Datenbank – Implementation

Für die Datenbank wird MariaDB benutzt, ein Fork von MySQL. Die Kommunikation zwischen Datenbank und LBR-Server geschieht mit JDBC (*Java Database Connectivity*).

Datenbank – Implementation

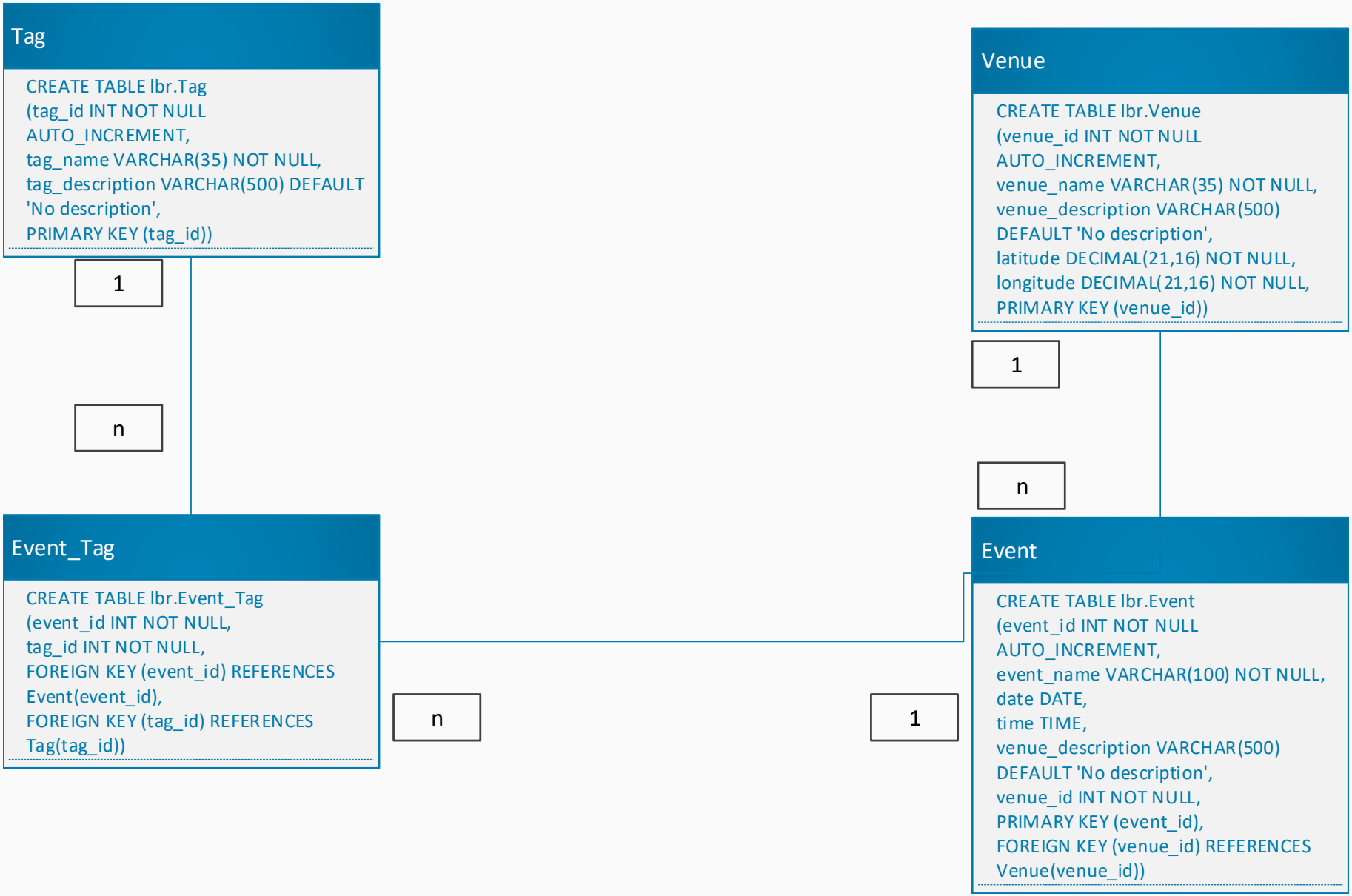
Für die Datenbank wird `MariaDB` benutzt, ein Fork von `MySQL`. Die Kommunikation zwischen Datenbank und LBR-Server geschieht mit `JDBC` (*Java Database Connectivity*).

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



LBR-Server

LBR-Server – Hardware & Betriebssystem

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

Der LBR-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.

LBR-Server – Hardware & Betriebssystem

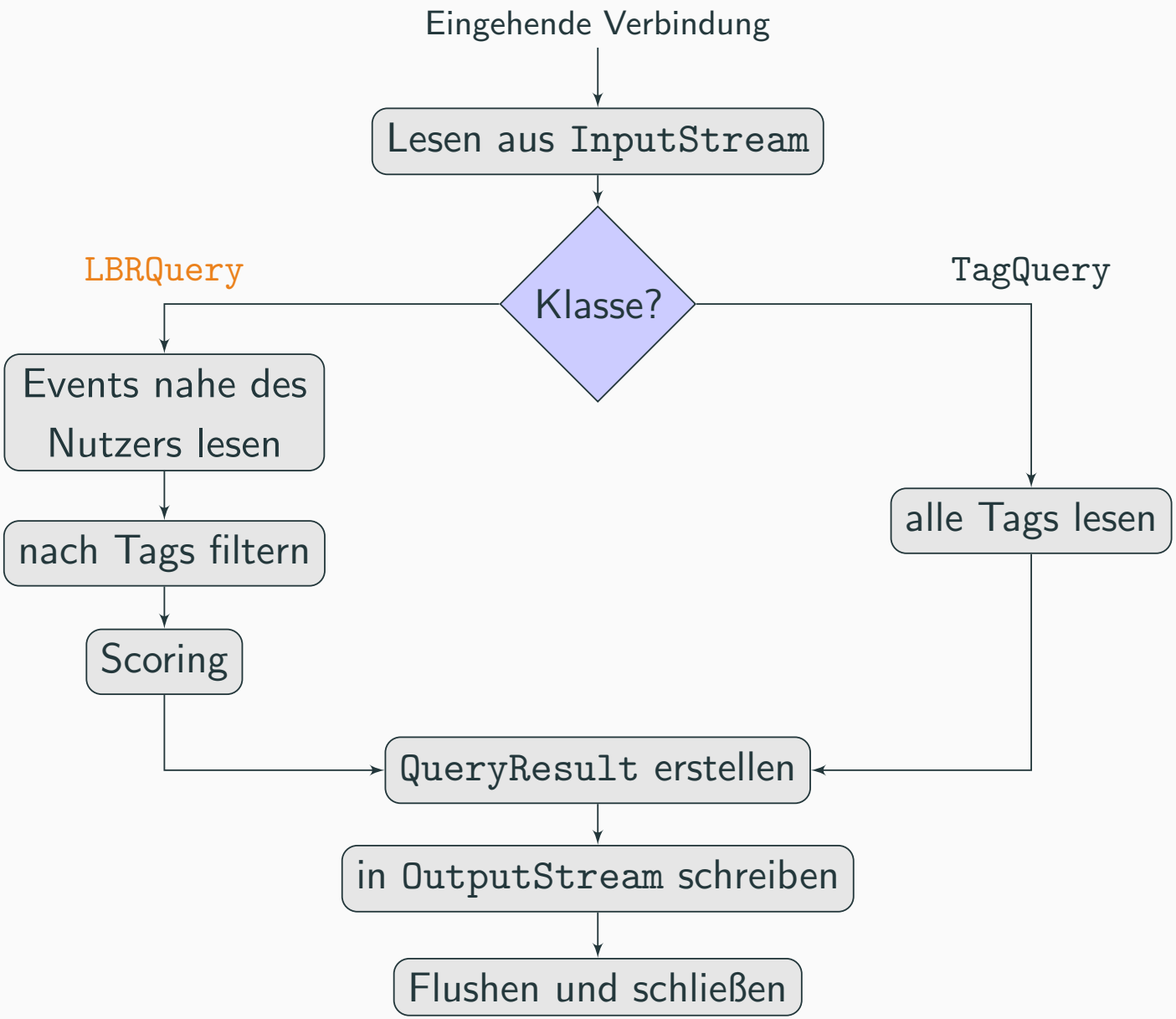
Als physikalischer Server wird ein `Raspberry Pi I B+` mit dem Betriebssystem `Raspbian` genutzt.

Der LBR-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.

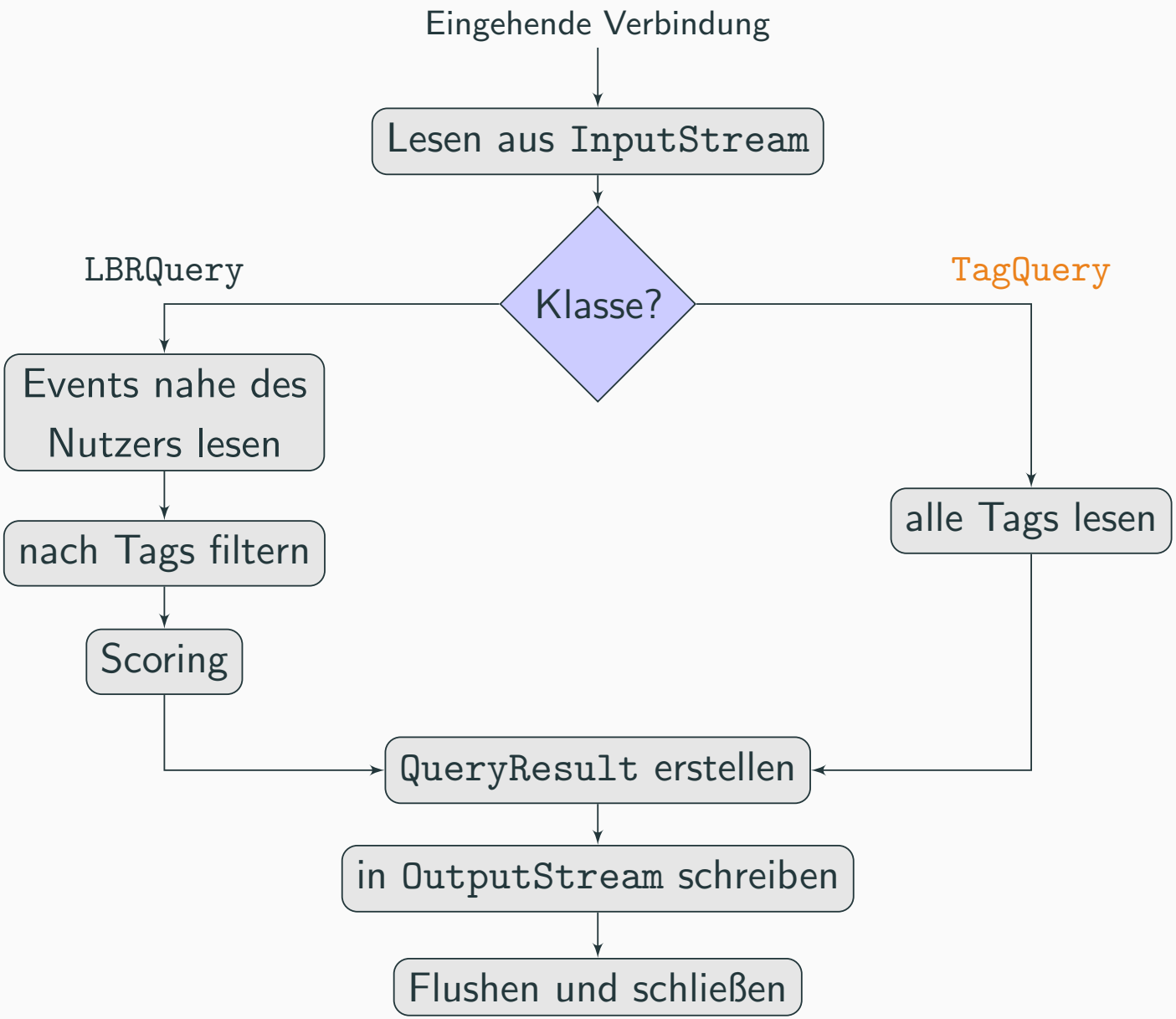
Kommt es während des Datenbank-Zugriffs zu einer `SQLException`, so wird automatisch ein Reconnect durchgeführt.

Das verwendete Scoring lässt sich dynamisch mithilfe des `EventScoreCalculator`-Interfaces festlegen.

LBR-Server – Schematischer Ablauf



LBR-Server – Schematischer Ablauf



Client-Server-Interface

Client-Server-Interface – Klassen

Folgende Klassen sind besonders wichtig für die Kommunikation zwischen LBR-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` - Zwischenspeicher für Objekte aus der Datenbank. 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 LBR-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 LBR-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 LBR-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 LBR-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

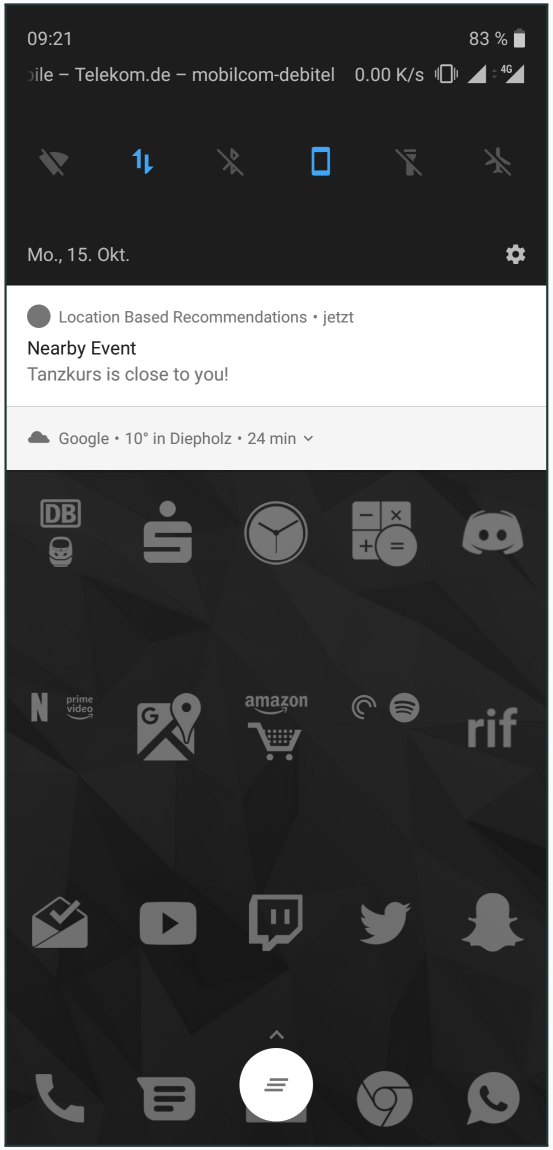


Abbildung 1: Geofence-Benachrichtigung

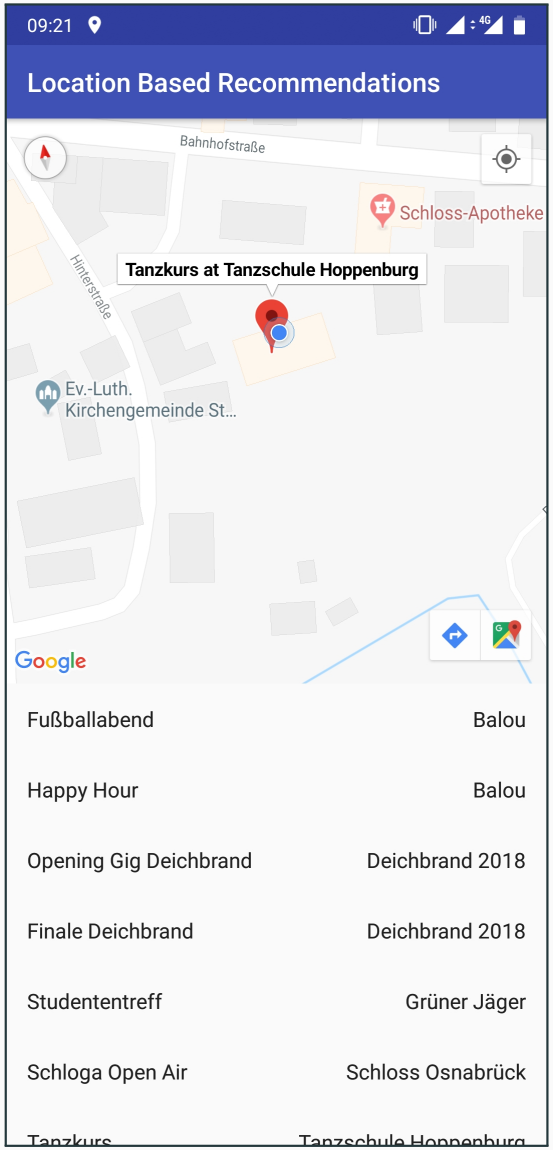


Abbildung 2: Marker auf Map

Screenshots II

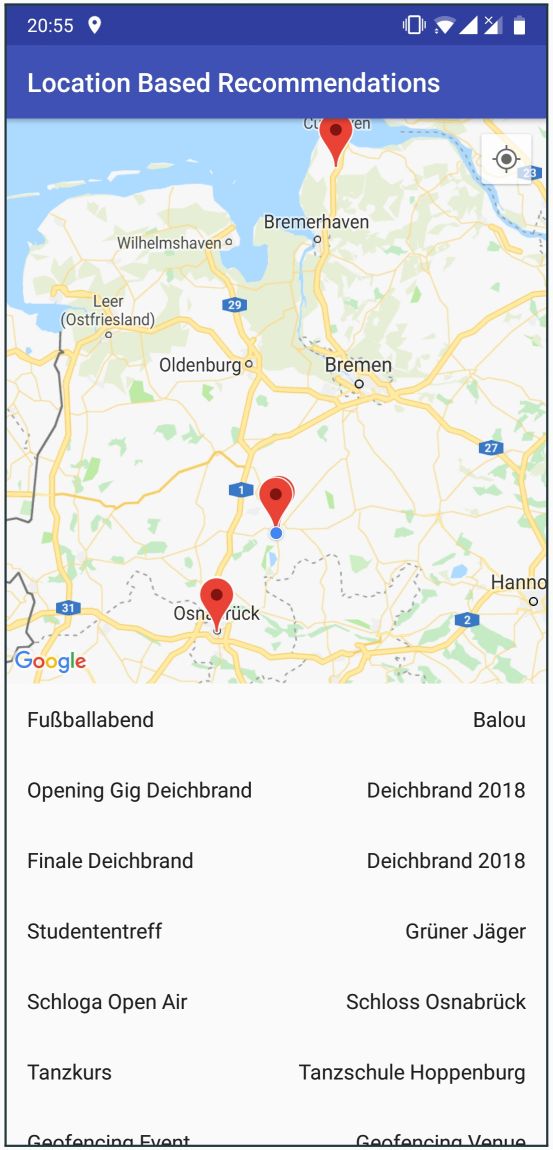


Abbildung 3: Karte & Liste

Abbildung 4: Event-Übersicht