Systemarchitektur v2

Aufgrund von neuen Erkenntnissen und Änderungen, in sowohl Anwendungslogik als auch benötigten Systemkomponenten, muss die Systemarchitektur aus MS1 teilweise angepasst und neu formuliert werden.

{{Diagramm einfügen}}

Server, Kommunikation, Protokoll

Das System benötigt eine Möglichkeit Benutzer auf abgeschlossene Algorithmen (Abrechnung) oder bestimmten, sie betreffenden, Änderungen in der Datenbank aufmerksam zu machen. Allerdings lässt sich dies nicht mit einer REST-Architektur über HTTP/1.1 so einfach implementieren, wie bisher angenommen (siehe Anhang: Konzept). HTTP/1.1 ist halb-duplex, d.h. der Server antwortet nur auf Anfragen des Clients und kann nicht eigenständig Nachrichten senden.
Um Clients kontaktieren zu können müssen deren Daten (Geräte-Id, Token, etc) in Sessions verwaltet werden. Dies verstößt gegen das Prinzip der Statuslosigkeit. Man müsste sich also Gedanken machen, ob man sich von der REST-Architektur abwendet und eine Websocket-Architektur (vollduplex) verwendet. Oder sogar eine Mischung aus beidem (siehe Zukunftsaussicht). Dafür muss man die Vor- und Nachteile der Architekturen abwägen (Quelle: https://blogs.windows.com/buildingapps/2016/03/14/when-to-use-a-http-call-instead-of-a-websocket-or-http-2-0/).

REST:
pro
- Standards für sichere und idempotente Methoden
- HTTP Fehlercodes
- synchronisation (Sicherheit, dass Nachricht angenommen wird oder nicht)
- kein großer Aufwand pro Verbindung Client-Server
- performanter für wenige, große Daten in großem Zeitraum
- performanter für Daten die sich nicht mehr oder relativ selten ändern

contra
- großer Header, lohnt sich nicht für sehr kleine Daten
- keine vollduplexe, bidirektionale Kommunikation
    - keine Server-Client Kommunikation

Websockets:
pro
- performanter für viele, kleine Daten in kurzem Zeitraum
- performanter für sich ständig ändernde Daten
- native Push-Features, vollduplexe und bidorektionale Kommunikation

- kleine Header

contra

- größerer Aufwand pro Verbindung Client-Server
- keine Standards für sichere und idempotente Methoden
- keine nativen Fehlercodes

Um sich nun für die passende Architektur entscheiden zu können, muss man definieren wie oft und in welcher Form Daten versendet werden. In dieser Anwendung sind sowohl Daten vorhanden, die sich regelmäßig ändern (Kassenzettel, Abrechnung, Statistik), als auch Daten die sich gar nicht bzw selten ändern (Benutzer, Gruppe, Kategorien, einzelne Ratings). Jedoch hat die Analyse der Benutzergruppe festgestellt, dass sich die häufiger aktualisierenden Daten nur voraussichtlich mehrmals am Tag verändern. Am Beispiel des Kassenzettels wäre dies der initiale Upload und daraufhin einige Veränderungsvorschläge der Gruppenmitglieder. Bzw. bei der Statistik immer nur dann, wenn ein neuer Kassenzettel hochgeladen/geändert wurde (höchstens 1-2x am Tag pro Benutzer). Die Daten, welche zum Server gesendet oder von diesem empfangen werden, sind nicht besonders groß (vsl. mehrere Kilobyte für die größten Datensätze). Eine Ausnahme wäre das Kassenzettelbild, dies wird jedoch später behandelt. Zudem kann man davon ausgehen, dass die Gruppenmitglieder sich auf ihre Reise konzentrieren wollen und persönlich miteinander kommunizieren. Da keine Chat-Funktion implementiert wird, muss kein Fokus auf Echtzeitkommunikation gelegt werden. Wenn man nun daraufhin REST und Websockets miteinander abwägt, ist REST, für das System,die optimalere Architektur.
Um den Benutzer dennoch, in bestimmten Bereichen der Anwendung, über den Server kontaktieren zu können, kann der Push-Service FCM (Firebase Cloud Messaging) von Google einbezogen werden. Dieser bietet Schnittstellen für die von uns verwendeten Plattformen Node.js und Android. Er ist einfach zu implementieren, effizient und kostenlos. Um diesen Dienst jedoch verwenden zu können, muss der Datenschutz berücksichtigt werden.
In den Push-Nachrichten, die von unserem Server über FCM an die betreffenden Clients gesendet werden, sind nur pseudonymisierte Daten enthalten. So stehen nur die hexadezimalen Ids der referenzierten Objekte in der Nachricht. Man benötigt direkten Zugang zu der Datenbank um diese zu entschlüsseln. So können die Daten von Google und von Sniffing geschützt werden. Notfalls kann die Nachricht zusätzlich noch mit Kryptographie-Algorithmen verschlüsselt werden. Der legitime Benutzer kann beim Empfang der Push-Nachricht die relevanten Daten, anhand der Datenbank-Id, mit einer HTTP GET Methode querien und einsehen.
Obwohl die direkten Daten nicht von Google einsehbar sind, so haben sie dennoch Zugriff auf die Geräte-Id des Benutzers, welche sie eindeutig zuordnen können, falls dieser bereits andere Google-Dienste verwendet. Daher bieten wir die Möglichkeit den externen Push-Service abzuschalten. So müssen die Benutzer jedoch regelmäßig den Server selbst kontaktieren, um Änderungen an Daten einsehen zu können.

Abschließend kann man also sagen, dass der Großteil der Kommunikation zwischen Client und Server, synchron ablaufen wird. Begründet wird dies durch die relativ seltene Verbindung und die geringe Größe der versendeten Daten. Die benötigte Zeit und die Ressourcen für diese Transaktion ist so gering, dass sich vollduplexe und bidirektionale

Kommunikation nicht lohnt. Die Kommunikation über FCM ist in sich selbst zwar auch synchron (HTTP). Im Blick auf das ganze System ist es jedoch zeitlich entkoppelt, also asynchron, da der Benutzer bei seiner Ursprünglichen Nachricht an den Server nicht warten muss, sonder eine Benachrichtigung bekommt.