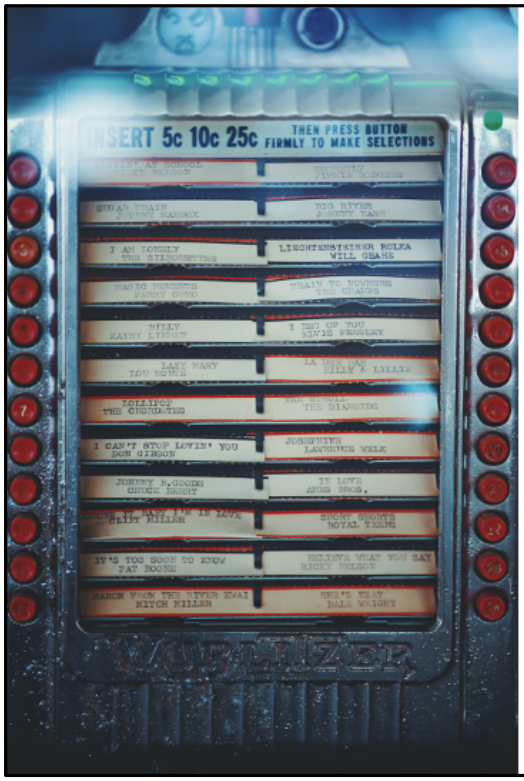
Audit 3

Ermöglichung einer demokratisch basierten Musikwahl bei Partys und Veranstaltungen

Bogdan Krenz, Carlos Bystron



Der Inhalt

- Realisierung PoCs
- Anwendungslogik
- User Flow Modelle
- Vertikaler Prototyp
- Projektplan

PoC - User Authentication

Beschreibung

Im bestmöglichen Fall soll eine Authentifizierung der NutzerInnen über eine Weiterleitung zur App des Anbieters auf dem Smartphone erfolgen. Dies soll durch die Nutzung von Deeplinks ermöglicht werden. Dort kann dann der Nutzung zugestimmt und die zu teilenden Daten gewählt werden. Dies ermöglicht ein Anmelden ohne die Eingabe eines Passwortes durch die NutzerInnen. Es ist davon auszugehen, dass der überwiegende Teil der NutzerInnen die App ihres jeweiligen Streaming Dienstes auf dem Smartphone installiert hat. Für die NutzerInnen, die die App nicht installiert haben soll eine Authentifizierung mit Email und Passwort ermöglicht werden.

Abgedeckte Projektrisiken

- Die Authentifizierung beim Streaming Anbieter ist nicht wie geplant mit einem redirect zur App des Anbieters möglich
- Login Prozess könnte zu lange dauern und NutzerInnen vergraulen
- NutzerInnen wissen Benutzername und Passwort ihres Streaming Anbieters nicht
- Die API des Streaming Anbieters ist instabil und zwischenzeitlich nicht erreichbar

Exit-Kriterien

- Eine erfolgreiche Authentifizierung hat stattgefunden
- Die Anmeldung durch redirect ist möglich

Fail-Kriterien

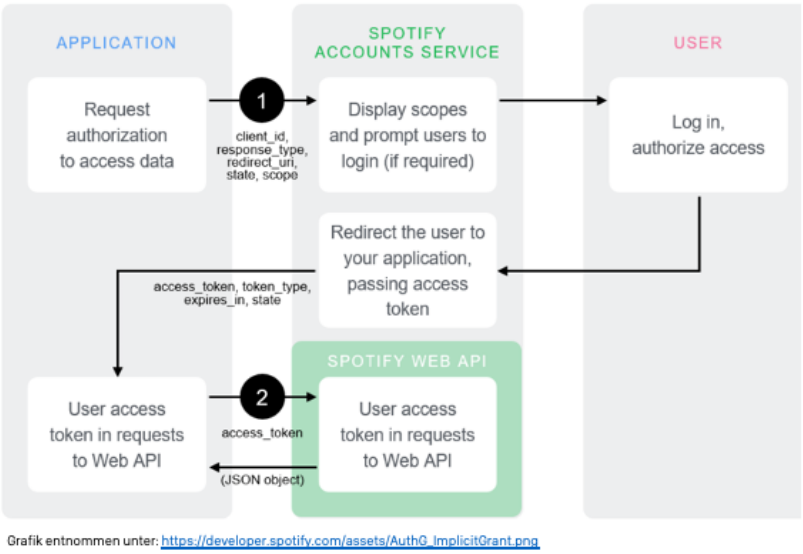
- Eine erfolgreiche Authentifizierung via redirect ist nicht möglich

Fallbacks

- Die Anmeldung ist lediglich über Email und Passwort möglich

Diese Folie dient lediglich um an die Konzeption des PoCs zu erinnern. Details zur Umsetzung folgen.

PoC - User Authentication



Grafik entnommen unter: https://developer.spotify.com/assets/AuthG_ImplicitGrant.png

Realisierung

- Authentisierung bei Spotify mit **Implicit Grant** (RFC 6749)
- Abwicklung über Client entlastet Server
- Bei vielen Geräten keine Eingabe von Email/PW nötig
- Token wird lediglich einmalig genutzt
  - Token muss nicht gespeichert werden
  - kein Bedarf für Token Refresh

Die Authentisierung unserer Nutzer bei Spotify wird mit dem in RFC 6749 (http://tools.ietf.org/html/rfc6749#section-4.2) spezifizierten *Implicit Grant* realisiert. Dieser bietet eine leichtgewichtige und gleichzeitig nutzerfreundliche Möglichkeit der Anmeldung für unseren Service. Der *Implicit Grant* wird vollständig über den Client ausgeführt, was zu einer Entlastung des Servers beiträgt.

Der Nutzer wird zu einer von Spotify zur Verfügung gestellten Seite weitergeleitet, dort kann er sich mit seiner Email und seinem Passwort anmelden. Wird ein Android Gerät oder ein Windows bzw. MacOS Gerät genutzt, bei welchem der Nutzer bereits in der Spotify App angemeldet ist, so ist ein erneutes eingeben der Email/PW Kombination nicht nötig. Nach Authentisierung Seitens Spotify findet ein redirect zurück zu unserer Anwendung statt. Dabei befindet sich das Token im Hash Fragment der genutzten URL. Das Token wird dann im Browser gelesen und Seitens des Servers einmalig für eine Abfrage bei der Spotify API genutzt.

Im Gegensatz zu Spotifys *Authorization Code Flow* (https://developer.spotify.com/documentation/general/guides/authorization-guide/#authorization-code-flow) bietet der Implizit Grant Flow keine Möglichkeit eines Token-Refresh. Dies wird für unsere Anwendung jedoch auch nicht benötigt, da lediglich eine einmalige Abfrage bei der API erfolgt.

PoC - DB testing

Beschreibung

Eine passendes Datenbanksystem wird gewählt ein Schema entwickelt und die DB mit ersten Testdaten gefüllt. Dabei soll insbesondere geprüft werden, ob gewünschte Zugriffsgeschwindigkeiten erreicht werden können, die Persistenz sowie Integrität gewährleistet ist und die Datenbank auch sehr große traffic, welche bei Großveranstaltungen auftreten könnte, verarbeiten kann.

Abgedeckte Projektrisiken

- Datenbankabfragen könnten bei Veranstaltungen mit vielen Gästen zu lange Dauern
- Datenunabhängige Struktur kann nicht gewährleistet werden
- Integrität der Datenbank kann nicht sichergestellt werden
- Persistenz -> Daten können bei Systemabsturz verloren gehen (besonders gespielte Lieder u. Präferenzen der Zuhörenden)

Exit-Kriterien

- Datenbank bildet gewünschtes Schema ab
- Datenbank kann gewünschte Zugriffsgeschwindigkeiten erreichen
- Datenbank hält hoher traffic stand

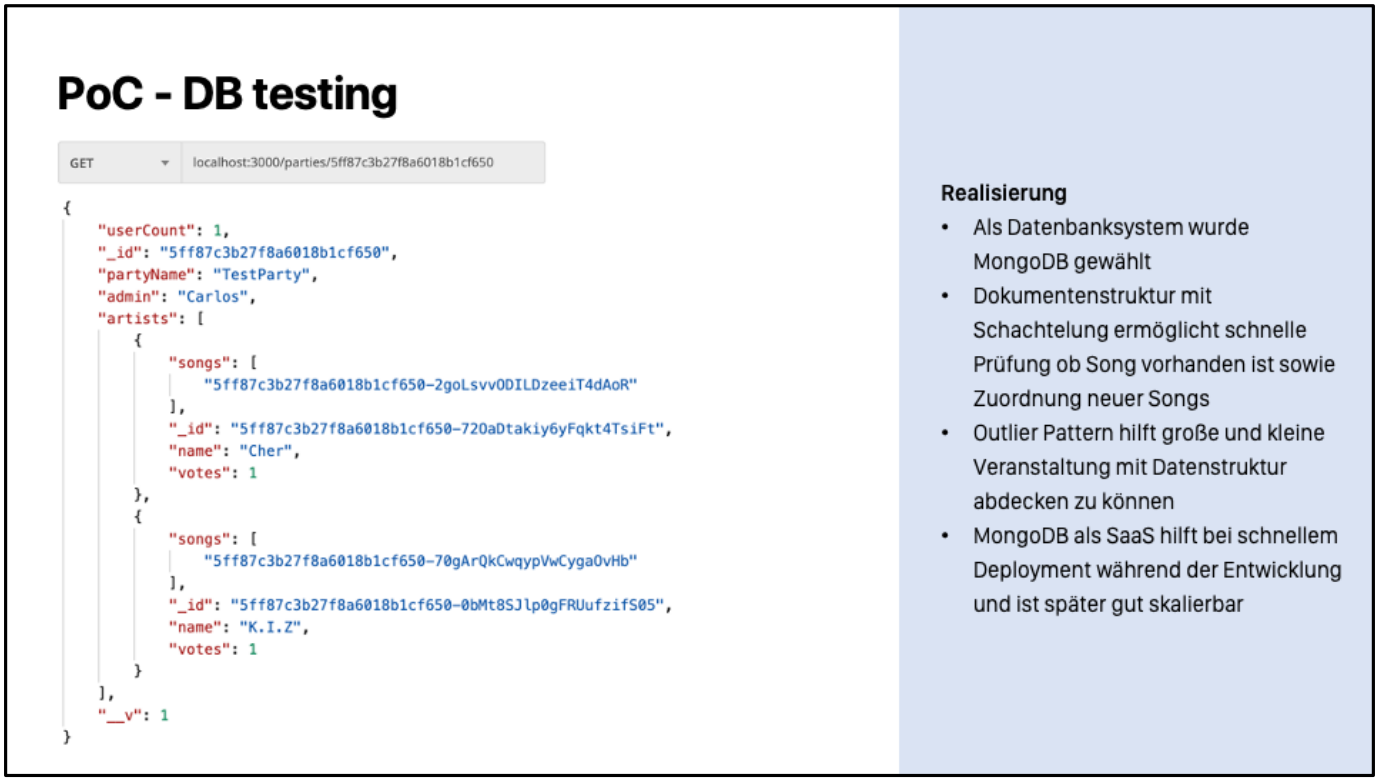
Fail-Kriterien

- Benötigte Daten können nicht rechtzeitig bzw. gar nicht abgerufen werden
- Datenbank bricht bei großer Traffic zusammen

Fallbacks

- Wechsel des Datenbanksystems
- Anpassen des Schemas

Diese Folie dient lediglich um an die Konzeption des PoCs zu erinnern. Details zur Umsetzung folgen.



Als Datenbanksystem wurde MongoDB gewählt. Durch die Dokumentenstruktur bietet Mongo diverse Vorteile, die uns bei diesem Projekt zugute kommen. Einerseits ermöglicht das verschachteln mehrerer Objekte, Lieder innerhalb ihrer Interpreten zu speichern, welche wiederum innerhalb der zugehörigen Party gespeichert werden. So ist eine Zuordnung neuer Lieder weitaus effektiver als es ohne diese Schachtelung wäre.

Es wird das *Outlier* Pattern genutzt (https://www.mongodb.com/blog/post/building-with-patterns-the-outlier-pattern). So konnte die Datenbankstruktur auf kleine Veranstaltungen wie beispielweise WG Partys ausgelegt werden, kann aber dennoch auch große Events abdecken.

Ein weiterer Vorteil bei MongoDB liegt in der Möglichkeit die Datenbank als *Software as a Service* zu nutzen. So können erste Prototypen schnell deployed werden und die Größe und Anzahl der Cluster später an die Nutzerzahl angepasst werden.

PoC - API und DB testing

Beschreibung

Bei diesem PoC soll sichergestellt werden, dass die benötigten Informationen von der API abgefragt und anonymisiert in unserer systemeigenen DB gespeichert werden können. Insbesondere soll bei diesem PoC die durchschnittlich übermittelte Anzahl von Liedern sowie die Zugriffsgeschwindigkeit der API ermittelt werden. Des Weiteren sollen die request Parameter für das künftige System ermittelt und getestet werden.

Abgedeckte Projektrisiken

- Die benötigten Ressourcen werden durch die API nicht zur Verfügung gestellt
- Persistenz -> Daten können bei Systemabsturz verloren gehen (besonders gespielte Lieder u. Präferenzen der Zuhörenden)
- Die API des Streaming Anbieters ist instabil und zwischenzeitlich nicht erreichbar

Exit-Kriterien

- Ressourcen können in angemessener Zeit von API abgerufen werden
- Die API liefert die benötigten Daten

Fail-Kriterien

- Benötigte Daten können nicht rechtzeitig bzw. gar nicht abgerufen werden

Fallbacks

- Wechsel der API

Diese Folie dient lediglich um an die Konzeption des PoCs zu erinnern. Details zur Umsetzung folgen.

PoC - API und DB testing

```
// long_term (calculated from several years of data and including all new data as it becomes available)
// medium_term (approximately last 6 months), short_term (approximately last 4 weeks). Default: medium_term
const timeRange = "short_term"

// The number of entities to return. Default: 20. Minimum: 1. Maximum: 50
const limit = "50"

const url = `https://api.spotify.com/v1/me/top/tracks?time_range=${timeRange}&limit=${limit}`

var header = {
    method: 'GET',
    headers: {
        'Authorization': `Bearer ${token}`,
    }
}

// Function die aufgerufen wird, un neuen Song in DB anzulegen auf den dann in Party referenziert wird
function addSong(song) {
    const songID = `${party._id}-${song.id}`
    const newSong = new Song({
        _id: songID,
        partyID: party._id,
        title: song.name,
        artist: song.artists[0].name,
        explicit: song.explicit,
        duration_s: song.duration_ms / 1000,
        images: song.albun.images,
        votes: 1,
    })
    newSong.save((err) => { if (err) { console.error(err)}  })
    userSongs.push(newSong)
    return newSong
}
```

Realisierung

- User-Top-Read-Scope gewährt Nutzern ein hohes Maß an Privatsphäre
- Normalisierung der Datenstruktur wird nicht angestrebt
  - Songs können mehrfach in DB abgespeichert werden
  - Ermöglicht effizientere Datenbankabfragen
  - Songs gehören immer zu einer Party und enthalten ihre Votes

Die Spotify API bietet unzählige Möglichkeiten auf die Daten der Nutzer zuzugreifen. Es wurde die Entscheidung getroffen den User-Top-Read-Scope zu nutzen, da dieser eine sehr eingeschränkte Zugriffsmöglichkeit bietet und den Nutzern somit ein hohes Maß an Privatsphäre bei ermöglicht. Diese Zugriffserlaubnis gestattet lediglich ein Abrufen der 50 meistgehörten Lieder und Künstler in je drei verschiedenen Zeiträumen.

Die Anzahl der tatsächlich abgerufenen Lieder und Künstler soll später je nach voraussichtlicher Anzahl der Gäste variiert werden, momentan werden immer alle 50 Datensets genutzt. Die Wahl des Zeitraums welcher die größte Zufriedenheit bei den Nutzern erzielt konnte noch nicht evaluiert werden.

Wird ein neuer Song zu einer Party hinzugefügt werden die wichtigsten Daten in unserer Datenbank übernommen. Dazu zählen neben Tittel und Interpret auch die Dauer, die Jugendfreundlichkeit sowie URLs zu dem Album Cover in verschiedenen Auflösungen.

Im Gegensatz zu klassischen, relationalen Datenbanken wird bei MongoDB keine Normalisierung der Datenstruktur angestrebt. Stattdessen ist das Ziel die Datenbank

so aufzubauen, dass Abfragen möglichst leichtgewichtig und effizient ausgeführt werden. Es wurde deshalb die Entscheidung getroffen nicht Partyübergreifend auf die selben Songs zu referieren.

Laufen somit gleichzeitig mehrere Veranstaltungen, bei denen Gäste *“Song 2“* von *Blur* in ihren Top Tracks haben, wird der Song auch mehrmals in der Datenbank angelegt. Dieses vorgehen bietet den Vorteil, dass jedes songObject seinen eigenen voteCount erhält und nicht das partyObject ein Mapping mit Votes zu Songs abdecken muss. Die intern genutzte songID setzt sich aus der partyID und der von Spotify genutzten songID zusammen. Somit genügt für eine nach Votes geordnete Rückgabe der Lieder einer Veranstaltung eine Abfrage der Song Collection mit der partyID. Da MongoDB automatisch einen Index für die gespeicherten IDs führt ist diese Abfrage sehr effizient.

PoC - API und DB testing

```
await json.items.forEach(async (song) => {

    // Prüfen ob der Künstler bereits in DB gespeichert ist
    const artistID = `${party._id}-${song.artists[0].id}`
    const existingArtist = party.artists.id(artistID)

    // Wenn Künstler nicht in DB gespeichert -> neuen Künstler anlegen u. Song speichern
    if(!existingArtist){
        const newSong = addSong(song)

        var artist = {
            _id: artistID,
            name: song.artists[0].name,
            votes: 1,
            songs: newSong
        }
        party.artists.push(artist)

    } else {
        // Wenn Künstler bereits in DB gespeichert -> Prüfen ob Song ebenfalls vorhanden ist
        const songID = `${party._id}-${song.id}`
        const existingSongID = existingArtist.songs.find(existingSongID => existingSongID == songID)

        // Wenn Künstler in DB gespeichert u. Song noch nicht -> Neuen Song hinzufügen und speichern
        if(!existingSongID){
            const newSong = addSong(song)
            existingArtist.songs.push(newSong)
        } else {
            // Wenn Künstler u. Song in DB gespeichert -> Song vote um eins erhöhen
            const existingSong = await Song.findById(existingSongID)
            existingSong.votes++
            existingSong.save((err) => {if (err) {console.error(err)}})

            userSongs.push(existingSong)
        }
    }
})
```

Realisierung

- User-Top-Read-Scope gewährt Nutzern ein hohes Maß an Privatsphäre
- Normalisierung der Datenstruktur wird nicht angestrebt
  - Songs können mehrfach in DB abgespeichert werden
  - Ermöglicht effizientere Datenbankabfragen
  - Songs gehören immer zu einer Party und enthalten ihre Votes

Dieser Codeblog zeigt den Vorteil, den die geschachtelte Datenstruktur ermöglicht. Für jeden neuen Song wird zuerst geprüft, ob der Interpret bereits in der Datenbank gespeichert ist. Ist dies nicht der Fall, so wird der Interpret neu angelegt und das Lied als neuer Song in der Datenbank gespeichert. Ist der Interpret bereits vorhanden wird geprüft ob auch der Song bereits vorhanden ist.

Beim ersten Nutzer Tests konnte festgestellt werden, dass die 50 Top Songs meist von weniger als 20 verschiedenen Interpreten stammen. So kann die Anzahl der zu durchlaufenden Daten deutlich gesenkt werden, wenn in erster Instanz nach dem Künstler gesucht wird.

Ist der Song noch nicht vorhanden so wird dieser neu in der Datenbank angelegt und in das Array der Lieder dieses Künstlers hinzugefügt. Ist der Song bereits vorhanden so wird der voteCount des Songs um eins erhöht.

PoC - Push Notifications

Beschreibung

Eine simple Anwendung des Notification Services zum Versenden von Push Benachrichtigungen soll die Nutzung veranschaulichen und uns helfen Möglichkeiten und Grenzen beim Versenden von Benachrichtigungen besser einschätzen zu können.

Abgedeckte Projektrisiken

- Der Push Notification Service funktioniert anders anders als erwartet, kann Notifications nicht zustellen bzw. den User nicht benachrichtigen
- Fehlende Erfahrung mit dem versenden von Push Notifications

Exit-Kriterien

- Benachrichtigungen können erfolgreich zugestellt werden

Fail-Kriterien

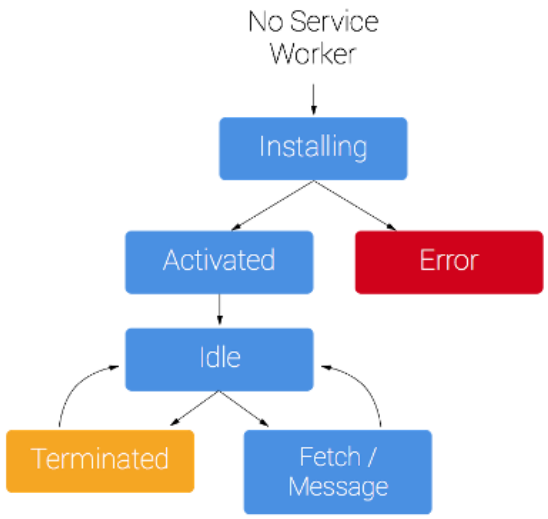
- Benachrichtigungen kommen nicht an
- Benachrichtigungen sind zeitlich stark verzögert
- Benachrichtigungen werden nicht sichtbar (aufwecken des Bildschirms, ggf. Vibration oder Ton) zugestellt

Fallbacks

- Wählen eines anderen PSN Services

Diese Folie dient lediglich um an die Konzeption des PoCs zu erinnern. Details zur Umsetzung folgen.

PoC - Push Notifications



Grafik entnommen unter: Quelle: https://developers.google.com/web/fundamentals/primers/service-workers

Realisierung

- Service Worker werden genutzt um Push Benachrichtigungen zuzustellen
- Implementierung funktioniert nicht in allen Browsern
- Service Worker laufen im Browser des Clients im Hintergrund, Hören auf Events und führen Code in Folge dieser aus

Dieser PoC beschäftigt sich damit, zu überprüfen ob man ein Service zum Erstellen und Versenden von Push Notifications benutzen kann und welcher Service in Frage kommt. Nach gründlicher Recherche sind wir zum Ergebnis gekommen, Service Worker zu verwenden. Service Worker sind kleine Teile von JavaScript Code, die im Browser des Clients im Hintergrund auf ein Event hören und ggf. Code ausführen, basierend auf dem Event.

Dies kann in unserem Fall eine Benachrichtigung sein. In unserem Beispiel-Code haben wir einen Service Worker installiert und den Client bei einem Soft-Reload der Seite eine Benachrichtigung geschickt. Dies soll keinesfalls die Logik in unserem System simulieren. Dieses kleine Beispiel dient nur zur Demonstration, wie Service Worker auf dem System des Clients installiert werden, welche Prozesse im Hintergrund passieren und welche Möglichkeiten diese Technologie bietet.

Neben dem Versenden von Push Benachrichtigungen kann man die Applikation auch als Native App auf dem jeweiligen Gerät installieren. Dies funktioniert, in dem man die Files der Applikation im Cache speichert, um Ladezeiten bei erneutem Aufruf der Seite zu minimieren.
Es können auch Daten, beim ersten Aufruf der Seite gefetched und gecached werden,

falls es zu Problem mit der Internetverbindung kommt und man trotzdem mit der App interagieren möchte. Dies Beschreibt im Grunde das Konzept der Progressive Web Applications (PWA), bei denen Service Worker eine wesentliche Rolle spielen.

Allerdings gibt es Nachteile, wenn man Teile dieser Technologien verwenden möchte. Einer dieser Nachteile ist zum Beispiel, dass man Standartmäßig keinen Support für alle Browser bekommt. Zu den Browsern, die die Service Worker unterstützen zählen Firefox, Google Chrome und Opera. Zwar hat Safari den Support für Service Worker in der Zukunft bestätigt, ist es im Moment schwierig diese sowohl für die oben genannten Browser, als auch Safari, zu implementieren. Ein weiterer Nachteil von Service Worker ist, wenn die Applikation auf ständige Kommunikation mit dem Server oder anderen Services angewiesen ist.

Zusammengefasst erlauben Service Worker dem Browser einen Schritt näher den Nativen Applikationen zu kommen. Reichhaltiges offline Benutzererlebnis, ständige Aktualisierungen im Hintergrund und Push Benachrichtigungen. All dies sind Merkmale einer Native Application, die vermehrt im Web eingesetzt werden.

Anwendungslogik

```
partySongs.forEach(song => {
    const guestPreference = (song.votes / party.guestCount) * (song.artist.votes / party.guestCount )* 100

    if (guestPreference > averageGuestPreference) {
        const partyFit = audioCompare(party.audioProfile, song.audioProfile)
        song.score = guestPreference + partyFit / 2
    }
});

function compare (partyProfile, songProfile) {

    if(songProfile.explicit && !partyProfile.explicit) {
        return 0
    }

    if(songProfile.danceability < 0.7 * partyProfile.danceability || songProfile.speechiness > 1.5 * partyProfile.speechiness) {
        return 0
    }

    const partyFit = Math.abs(songProfile.danceability - partyProfile.danceability) +
                     Math.abs(songProfile.speechiness - partyProfile.speechiness) * 0.9 +
                     Math.abs(songProfile.tempo - partyProfile.tempo) * 0.9 +
                     moreProfiles

    return (1 - partyFit) * 100
}
```

Anmerkung: Bei den hier dargestellten Codeblöcken handelt es sich um Pseudocode. Die Anwendungslogik wurde noch nicht realisiert und wird im Zuge der Implementierung weiter ergänzt.

Das klar definierte Ziel der Anwendungslogik unseres Systems ist das Finden der bestgeeignetsten Lieder für die anwesenden Gäste. Dieses Ziel wird in zwei Schritten erreicht:

1. Das Werten der Top 50 Songs aller anwesenden Gäste hinsichtlich der Eignung für die jeweilige Veranstaltung
2. Die ergänzende Auswahl weiterer Songs passend zu den Musikgeschmäckern der Gäste sowie den Vorstellungen des Party Hosts

Zuerst wird dazu ein gewünschtes Audioprofil der Party ermittelt. Sollen die Gäste tanzen? Soll die Musik er ruhig und akustisch oder laut und elektronisch sein? Solche Einschätzungen kann der Host vor Beginn der Party vornehmen und während der laufenden Party anpassen. Angedacht ist später auch die Gäste an dieser Umfrage teil haben zu lassen, im Rahmen dieser Veranstaltung wird diese Implementierung wohl jedoch nicht möglich sein.

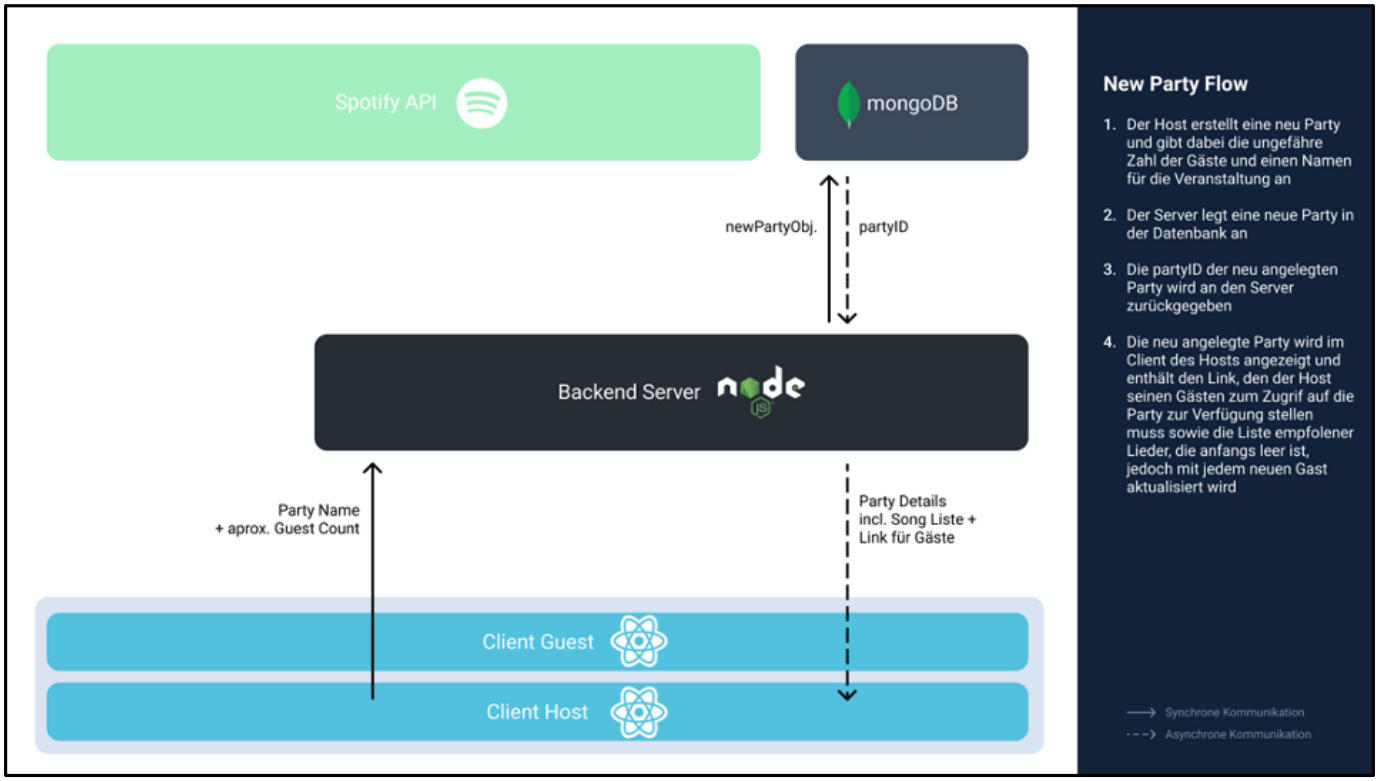
Anhand der Song Votes, sowie der Votes für die Künstler wird in Relationen zur Gesamtzahl der anwesenden Veranstaltungsgäste ein guestPreference Score ermittelt. Ist der Score größer als der durchschnittliche guestPreference Score so wird das Audio Profil des entsprechnden Liedes mit dem der Party verglichen. Es ergibt sich der partyFit, ein Sore welcher die Passgenauigkeit des Liedes zu den Wünschen des Veranstalters und der Gäste für diese Party darstellt.

Geprüft wird dabei zuerst ob der Song explizite Inhalte enthält, obwohl der Veranstalter dies nicht wünscht. Ist dem nicht der Fall, wird geprüft ob die Mindestübereinstimmung in punkto Tanzbarkeit sowie Anteil an Liedern mit bzw. ohne Lyrics gegeben ist. Wünscht der Host lediglich Instrumentalmusik, werden an dieser Stelle bereits jegliche Tracks mit Gesang aussortiert. Im Anschluss wird für die übrig gebliebenen Songs der partyFit berechnet. Dazu werden jeweils die Beträge der songProfile Werte minus derer des partyProfiles errechnet und addiert. Je größer die Übereinstimmung desto kleiner bleibt der Wert des partyFit Scores. Die hier dargestellten Audio Profile bilden nur einen Teil der tatsächlichen Rechnung ab.

Um den Rückgabewert zu erhalten, wird der berechnete partyFit von 1 abgezogen und mit 100 multipliziert. So kann eine Prozentzahl zurückgegeben werden. Je höher diese ist, desto eher stimmt das Audio Profil des Songs mit dem der Party über ein.

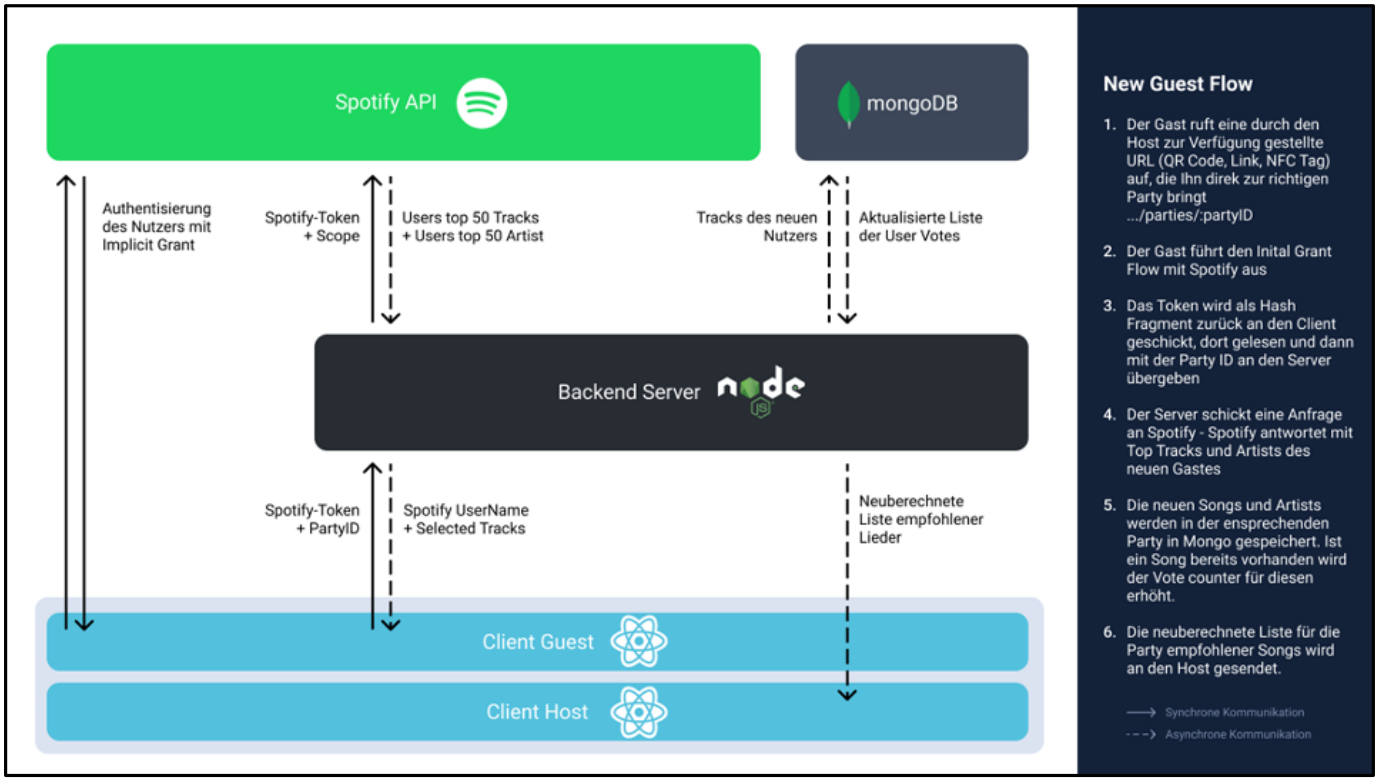
Die Gesamttauglichkeit des Liedes berechnet sich dann aus dem Durchschnitt des partyFit und der guestPreference.

Angedacht ist außerdem ein eigenständiges finden neuer Lieder durch die Anwendung. Dies soll mithilfe des festgelegten Audioprofils der Party sowie den Künstlerpräferenzen der Gäste geschehen. Diese Funktion soll verhindern, dass nicht ausreichend passende Lieder vorhanden sind um die Veranstaltung füllen zu können.



Der *New Party Flow* bildet das Erstellen einer neuen Veranstaltung durch den Host ab. In der aktuellen Implementierung ist keine Anmeldung des Hosts erforderlich. Dies ist der Fokussierung auf andere Aspekte der Entwicklung geschuldet, später aber angedacht. Da dem Host jedoch keine sensiblen Daten zur Verfügung gestellt werden (sondern lediglich eine Liste empfohlener Songs) stellt dies kein Datenschutzproblem dar. Eine Anmeldung würde lediglich den Wechsel zwischen unterschiedlichen Geräten beim Host erleichtern.

Zentrale Idee ist, dass der Host nach dem erstellen einer Veranstaltung einen Link (.../parties/:partyID) zu dieser erhält, den er seinen Gästen zur Verfügung stellen kann. Dies könnte Beispielsweise geschehen, indem er den Link über Social Media Kanäle der Veranstaltung publiziert, im Eingangsbereich der Veranstaltung QR Codes platziert oder NFC Chips in den Eintrittskarten verarbeitet. Das Interface des Hosts wird dann mit jeder Anmeldung aktualisiert. Ein GuestCount zeigt die Anzahl bereits angemeldeter Partygäste, die Liste von Liedern die zu spielende Musik.

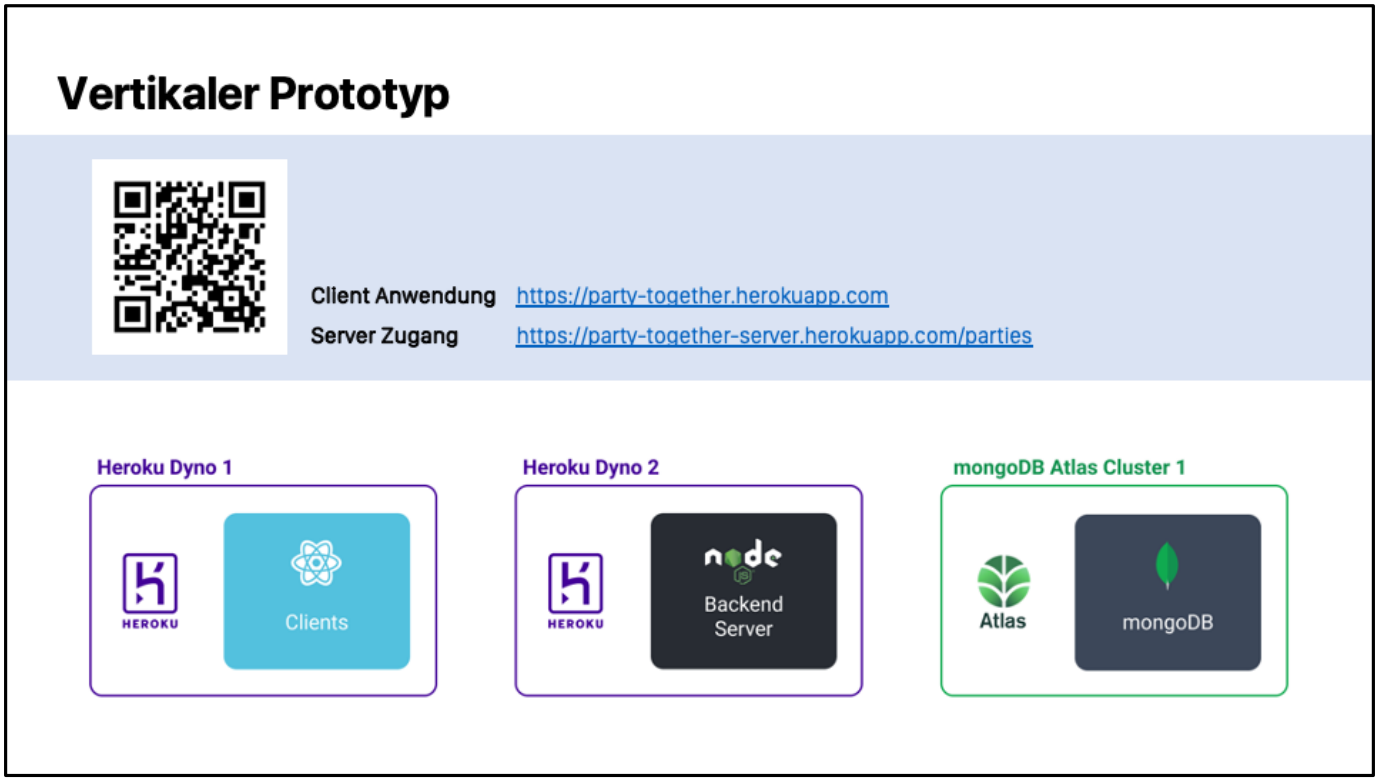


Der *New Guest Flow* ist der zentrale Aspekt unseres Systems. Es handelt sich bei dem hier aufgezeigten Interaktionsmodell um den Systemablauf, der bei der Anmeldung jedes neuen Gastes ausgeführt wird. Zentrales Designziel war es hierbei die benötigten Interaktionsschritte des Gastes so gering wie möglich zu halten. Eine Wahl der zu registrierenden Party wird dem Gast durch den direkten Link des Hosts erspart. Er kann sofort mit der Registrierung über Spotify beginnen. Diese wurde über einen Inital Grant realisiert und ermöglicht, bei den meisten Geräteklassen, dass die Anmeldedaten übernommen werden, sollte auf einem Gerät bereits ein Nutzer bei Spotify angemeldet sein.

Nachdem das Token an den Client zurückgegeben wurde führt dieser automatisch eine Weiterleitung an den Server aus. Ein direktes Senden des Tokens an den Server ist nicht möglich, da das Token von Spotify im Hash Fragment der URL gesendet wird. Dieses lässt sich lediglich im Browser des Clients auslesen. Auf dem Server werden von der Spotify API die 50 Top Tracks sowie die 50 Top Artists des jeweiligen Users in den letzten drei Monaten erfragt und im JSON Format zurückgegeben. Die weiteren Abläufe werden asynchron ausgeführt, da sie in Abhängigkeit der von Spotify zurückgelieferten Daten stehen.

Nach Erhalt der Daten von Spotify werden diese der Veranstaltung zugeordnet. Es wird ein neues Ranking der bestgeeignetsten Lieder erstellt und in mongoDB gespeichert. Parallel wird dem Gast eine Liste seiner aktuellen Top 10 Songs angezeigt und ihm wird für die Anmeldung gedankt. Die Interaktion des Hosts, die zur Anmeldung nötig ist, ist an dieser Stelle (nach einem Tap!) beendet. Die Entscheidung den User kurz warten zu lassen und diesem nicht bereits nach erfolgreicher Authentisierung eine Success Message anzuzeigen wurde zugunsten einer geringeren *Black Box Wirkung* getroffen. Durch das anzeigen der Top 10 Songs des Nutzers wird das Vertrauen in die Anwendung erheblich gestärkt und der Nutzer weiß, dass diese wirklich seine Musikvorlieben erkannt hat.

Nach dem aktualisieren der Datenbank wird das Interface des Hosts ebenfalls aktualisiert und zeigt nun die neuberechnete Liederliste an. Dem Host Client wird dazu eine neue JSON Datei zur Verfügung gestellt.



Ein erster vertikaler Prototyp wurde entwickelt und zu Testzwecken ebenfalls bereits deployed. Der Prototyp deckt den New Guest Flow (vorherige Folie) ab, welcher als zentraler Aspekt der Anwendung ausgemacht wurde.

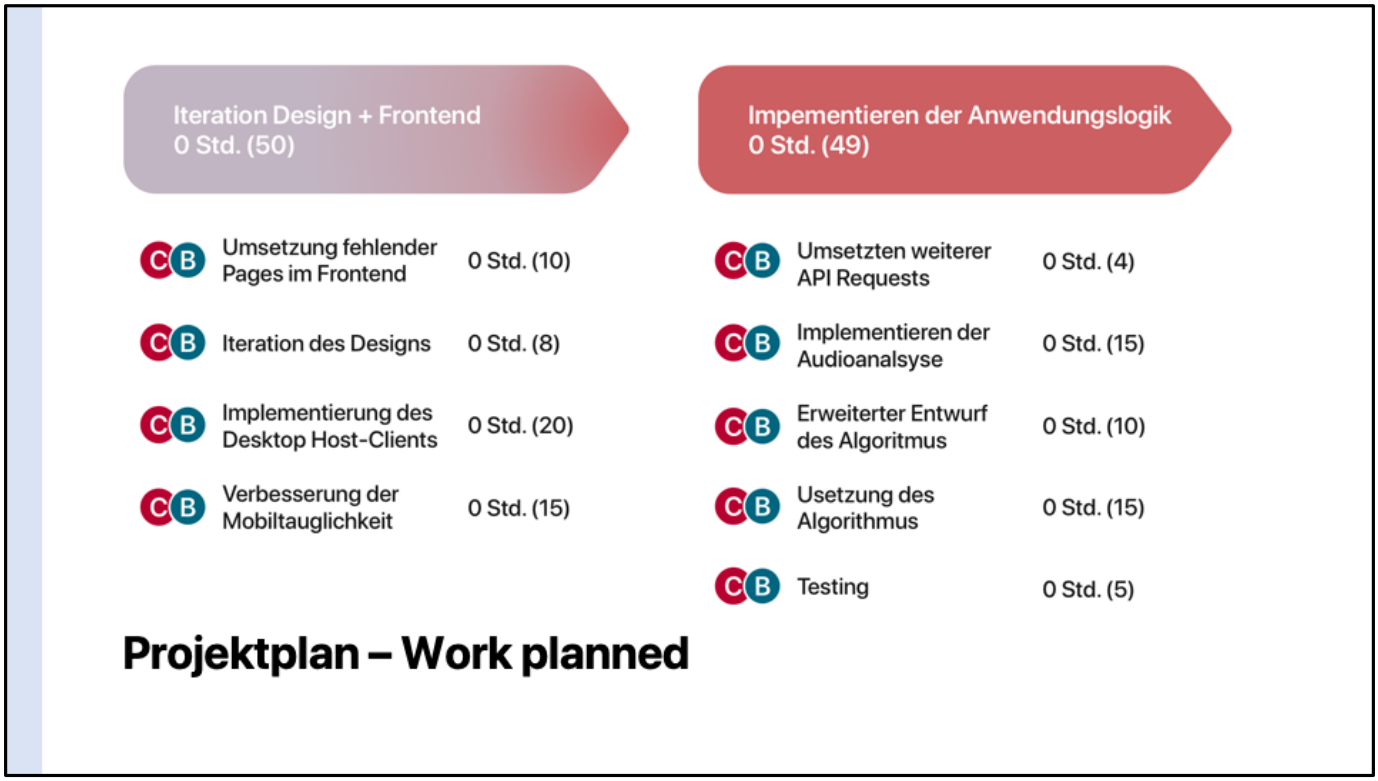
Das Deployment läuft über Heroku wobei der Client in einem vom Server unabhängigen Dyno läuft. Die Datenbank wird über MongoDB Atlas bereitgestellt. Für ein späteres Production-Deployment wäre ein Umstieg auf Docker Container und Kubernetes denkbar, wobei auch die aktuelle Infrastruktur problemlos skalierbar wäre.

Bei der Entwicklung des Prototypens wurde insbesondere Wert auf die Vollständigkeit der Architektur gelegt. Alle Routes der Anwendung werden bereits abgedeckt. Noch nicht vollständig umgesetzt wurde jedoch das Frontend beider Clients sowie der Auswahlalgorithmus zur Bestimmung der richtigen Songs.

<u>Erläuterungen zum aktuellen Stand beim Frontend:</u>
In der App wird zwischen Guest und Host unterschieden. Im Frontend wird dies mit durch zwei Buttons gekennzeichnet. Mit dem grünen Button verbindet man sich mit seinem Spotify Account. Es wird ein Redirect zu Spotify gemacht und der User wird

authentisiert. Nach erfolgreichen Authentisierung wird der User weitergeleitet und ihm werden Lieder angezeigt, die der Liste des Hosts hinzugefügt wurden. Dabei wird ein Request zum Server gemacht und ein neuer User der Liste hinzugefügt. Im Backend wird nur der User Count erhöht. Mit dem Token von der Authentisierung, schickt das Backend eine Anfrage an Spotify mit den Lieblingsliedern des Users.

Im nächsten Schritt wird das Backend um die Funktionen erweitert um die Liste für den Host zu manipulieren. Außerdem soll die Anfrage per Push Notifications vom Host and die Guests verteilt werden, ob den Guests die gespielte Musik gefällt.



Für das nächste Audit planen wir unser Frontend zu überarbeiten sowie die Anwendungslogik vollständig zu implementieren.

Bei der Überarbeitung des Frontends ist das Hauptziel dem mobile-first Approach des Gast Interfaces noch mehr Geltung zu tragen und diesen noch konsequenter umzusetzen. Beim Interface des Hosts hingegen ist von einer Nutzung auf größeren Geräten auszugehen. Hier muss das Frontend noch vollständig entworfen werden.

Für die Umsetzung der Anwendungslogik müssen zuerst weitere API Abfragen getätigt werden um die Songdaten in der DB um solche bezüglich der Audioinformationen anzureichern. Diese werden dann kombiniert mit den Votes der User genutzt um über passende Musik für die Veranstaltung zu entscheiden.

In diesem Sinne werden wir weiter mit Hochdruck an der Realisierung unseres Projektes arbeiten um zukünftig nur noch gute Musik auf Partys hören zu müssen 😉