

Dokumentation zur Studienarbeit

SpotifyParty

A Motivation und Anforderungen

- I. Projektidee und Recherche
- II. Anforderungen

B Planung und Entwurf

- I. Framework und Bibliotheken
- II. Organisation der Zusammenarbeit
- III. Entwurf der Anwendung

C Entwicklung

- I. Index-Seite
 - 1. Betreten einer Session
 - 2. Erstellen einer Session
 - 3. Log-in über Spotify
 - 4. Abfrage der Playlists und Geräte
- II. Settings-Seite
 - 1. Auswahl Wiedergabeliste und -gerät
 - 2. Erstellung der Sitzung
- III. Session-Seite
 - 1. Verbindung mit Websocket
 - 2. Starten der Sitzung
 - 3. Abstimmung und Auszählung
 - 5. Verlassen des Websockets

D Inbetriebnahme

- I. Voraussetzungen
- II. Lokale Inbetriebnahme

E Fazit

- I. Umsetzung der Anforderungen
- II. Erweiterungsmöglichkeiten
- III. Persönlicher Eindruck

F Quellen

A Motivation und Anforderungen

I. Projektidee und Recherche

Die grundlegende Projektidee war es, eine Anwendung zu entwickeln, die es ihren Nutzern erlaubt, beispielsweise auf Partys eine eigene Spotify-Playlist mit anderen Gästen zu teilen und gemeinsam über das als nächstes abzuspielende Lied abzustimmen. Diese Idee ist keineswegs völlig neuartig, da dieses Konzept einer "Social Jukebox" bereits von zahlreichen Programmierern wie z. B. bei „festify.rocks“ oder „jukestar.mobi“ umgesetzt wurde.

Die oben genannten Anwendungen bieten zudem einen äußerst großen Funktionsumfang, der weit über die bloße Abstimmung hinausgeht und beispielsweise das Vorschlagen anderer Songs außerhalb der eigenen Playlist oder zusätzliches Streaming auf Wiedergabegeräten von Sitzungsteilnehmern erlaubt.

II. Anforderungen

Nach der anfänglichen Recherche ergaben sich für die Anwendung die folgenden grundlegenden Muss-Kriterien:

Die Applikation soll es ermöglichen, eine bestehende Spotify-Playlist abzuspielen und aus einer festgelegten Anzahl an Tracks den jeweils nachfolgenden Song durch eine Abstimmung festzulegen. Hierzu soll eine gemeinsame Sitzung über einen Link mit anderen Nutzern geteilt werden. Alle Beteiligten sollen zudem Live-Feedback über den Stand der Abstimmung sowie den Wiedergabefortschritt des laufenden Songs erhalten.

Die Wunsch-Kriterien betreffen in diesem Fall vor allem die Bedienung der Anwendung.

Im Vordergrund aller Designentscheidungen soll eine möglichst geringe Einstiegshürde für einen spontanen Einsatz auf Partys stehen.

Nachdem die Anwendung aller Wahrscheinlichkeit nach hauptsächlich auf Mobilgeräten genutzt werden wird, ist eine dafür optimierte Benutzeroberfläche wünschenswert.

Zusätzlich sollen die Nutzer mit möglichst wenigen Klicks zur eigentlichen Abstimmung gelangen und nicht durch die Notwendigkeit einer Registrierung von der Benutzung der Website abgeschreckt werden.

Da die Einladung weiterer Teilnehmer zur eigenen Sitzung ein wesentlicher Bestandteil der Anwendung ist, sollte der Link zur jeweiligen Session möglichst kurz und lesbar gehalten werden.

B Planung und Entwurf

I. Framework und Bibliotheken

Als Framework der Anwendung kommt "Django" zum Einsatz, als Entwicklungsumgebung wird JetBrains' PyCharm verwendet.

Nachdem die Applikation Live-Feedback an alle Nutzer voraussetzt, wird das Framework zur Implementierung des dazu benötigten Websockets um die Bibliothek "DjangoChannels" erweitert.

Der Zugriff auf die Inhalte der Spotify-Playlist des Users sowie das Play-back auf dem ausgewählten Wiedergabegerät muss durch die Spotify-API erfolgen.

Um die Nutzung der davon bereitgestellten Funktionalitäten zu erleichtern, wurde die "Spotipy"-Bibliothek ausgewählt, auch wenn ein Großteil ihrer OAuth-Funktionalität durch die Verwendung von Django für dieses Projekt überflüssig ist.

Eine genaue Auflistung aller verwendeten Pakete mit ihren jeweiligen Versionen findet sich in der Datei "requirements.txt":

```
aioredis=1.3.1
asgiref=3.3.1
async-timeout=3.0.1
attrs=20.3.0
autobahn=20.12.2
Automat=20.2.0
certifi=2020.12.5
cffi=1.14.4
channels=3.0.2
channels-redis=3.2.0
chardet=4.0.0
constantly=15.1.0
cryptography=3.3.1
daphne=3.0.1
Django=3.1.4
hiredis=1.1.0
hyperlink=20.0.1
idna=2.10
incremental=17.5.0
msgpack=1.0.2
pyasn1=0.4.8
pyasn1-modules=0.2.8
pycparser=2.20
PyHamcrest=2.0.2
pyOpenSSL=20.0.1
pytz=2020.4
requests=2.25.1
routing=0.2.0
service-identity=18.1.0
six=1.15.0
spotipy=2.16.1
sqlparse=0.4.1
txaio=20.4.1
urllib3=1.26.2
zope.interface=5.2.0
```

II. Organisation der Zusammenarbeit

Die Zusammenarbeit im Zweierteam wurde größtenteils über das Codeverwaltungsprogramm „Git-Lab“ organisiert. Das dort verfügbare Issues-Board diente hierbei über den gesamten Entwicklungsprozess als To-do-Liste.

Des Weiteren fanden regelmäßige Zoom-Meetings zur weiteren Planung, Besprechung des aktuellen Projektstandes oder zur gemeinsamen Lösung etwaiger Probleme statt.

Zudem wurde die Webanwendung Figma zur kollaborativen Erstellung eines rudimentären Styleguides verwendet.

III. Entwurf der Anwendung

Um die Anforderung der Einladung weiterer Nutzer zur eigenen Sitzung zu erfüllen, wird eine Art "Einladungscode-System", wie es z.B. bei der Quizanwendung „Kahoot“ oder Partyspielen wie „skribbl.io“ zum Einsatz kommt, verwendet. Hierbei wird beim Erstellen einer Session ein kurzer Zufallscode generiert, welcher die Sitzung während ihrer Laufzeit eindeutig identifiziert und Nutzern den Beitritt zu dieser erlaubt.

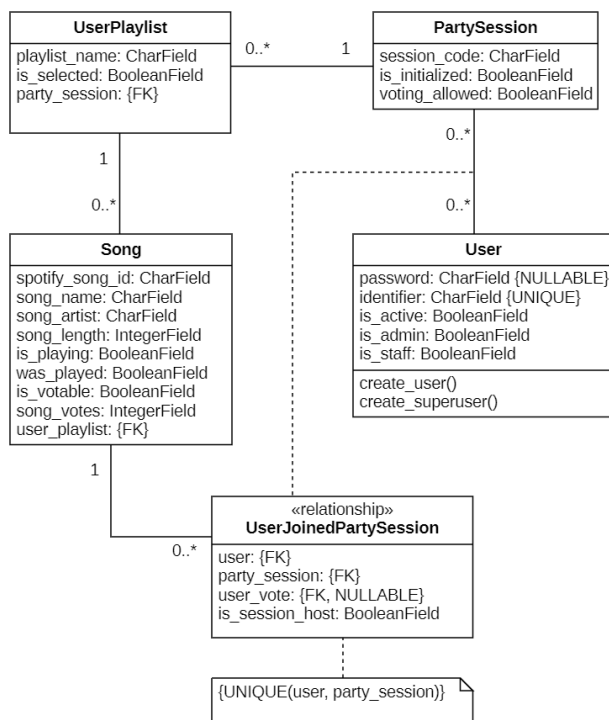
Dies ermöglicht zudem die Erfüllung der Wunsch-Kriterien einer niedrigen Einstiegshürde, einfacher Einladung von Gästen und möglichst geringer Anzahl an Log-ins/ Registrierungen.

Einige der verwendeten Technologien wie beispielsweise die Spotify-API oder "DjangoChannels" waren für alle Teammitglieder neuartig, konnten jedoch erst nach fortgeschrittener Entwicklung des Projekts implementiert werden.

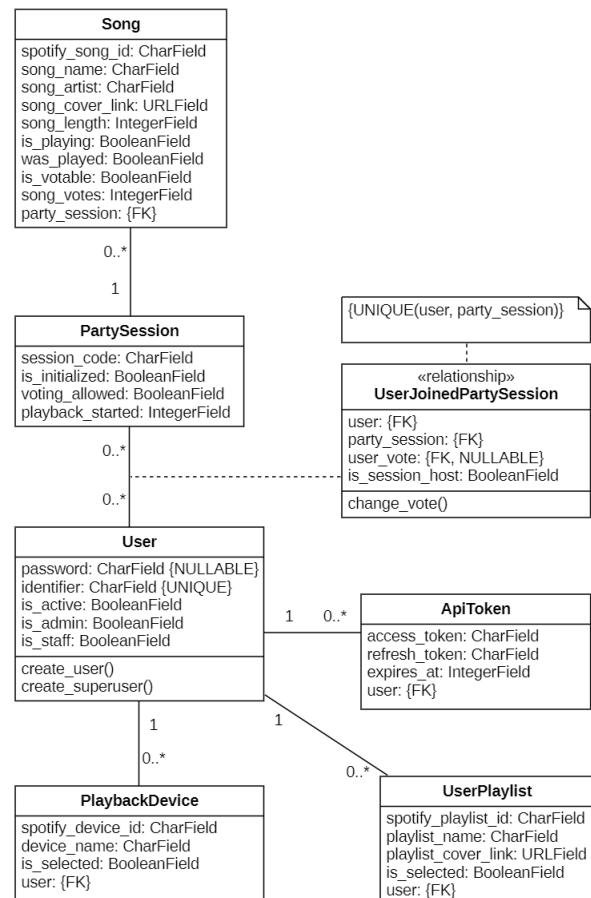
Um also losgelöst vom Haupt-Repository einige Grundlagen-Prototypen zu entwickeln und zu testen, wurden diese in lokalen Einzelprojekten umgesetzt. Die daraus abgeleiteten finalen Lösungen konnten anschließend direkt in die Hauptanwendung übernommen werden.

Dieser Entwicklungsprozess erlaubte ein sehr freies Erkunden neuer Technologien ohne die Stabilität des Haupt-Repositorys zu gefährden.

Die im ersten Programmentwurf festgelegte grobe Struktur der Datenbank lässt sich im folgenden UML-Diagramm erkennen:



Diese wurde im Verlauf der Entwicklung stetig an die neuen Anforderungen der Anwendung, wie zum Beispiel das Speichern der Ergebnisse der API-Anfragen, angepasst. Daraus ergibt sich das aktuelle Datenbanksystem:



Der erste Entwurf der Anwendung teilte diese in drei Hauptbestandteile auf:

Die Index-Seite bietet dem Nutzer zunächst die Möglichkeit einer bereits existierenden Session mittels des Zufallscodes beizutreten oder selbst eine Party zu erstellen.

Wird eine eigene Sitzung eröffnet, kann der User auf einer Settings-Seite die abzuspielende Playlist und das Wiedergabegerät auswählen.

Schlussendlich finden sich sowohl Gäste als auch Hosts auf der Hauptseite der Anwendung wieder, auf der über die verschiedenen Lieder abgestimmt werden kann.

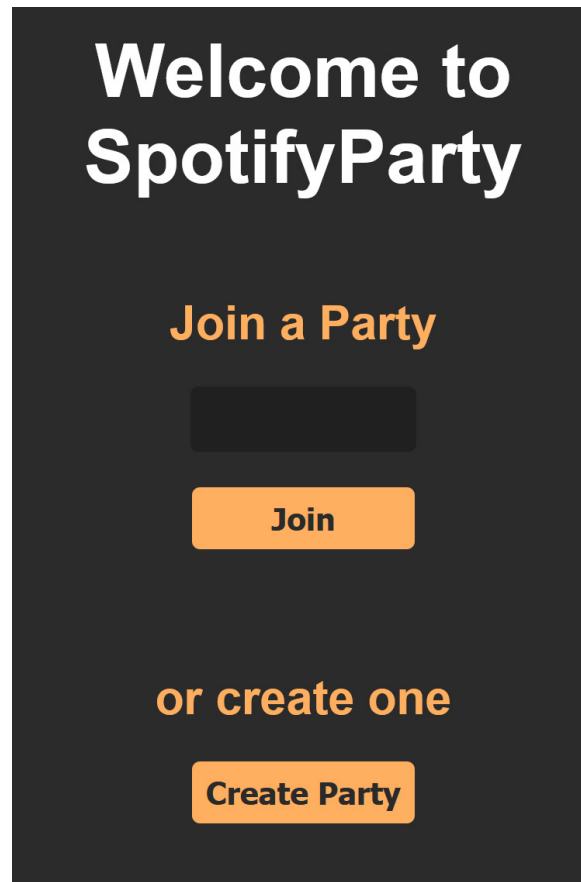
Diese Struktur blieb über den gesamten Entwicklungsprozess weitgehend erhalten, wenngleich sie zwischenzeitlich etwa bei der anfänglichen Implementierung der Spotify-API angepasst werden musste.

Final wurde die Anwendung jedoch wieder in die ursprüngliche Aufteilung zusammengeführt, sodass der aktuelle Entwurf lediglich die Umleitung zum Spotify-Login bei erstmaliger Anmeldung als zusätzliches Element beinhaltet.

C Entwicklung

I. Index-Seite

Startet der User die Anwendung, so findet er sich zunächst auf der Index-Seite wieder. Diese stellt wie bereits beschrieben zwei Grundfunktionen zur Verfügung.



Welcome to
SpotifyParty

Join a Party

Join

or create one

Create Party

1. Betreten einer Session

Mit dem Formular "Join a party" kann der Nutzer einer laufenden Sitzung mithilfe des 6-stelligen Zufallscodes beitreten. Hierbei wird zunächst direkt auf die "party_session"-View weitergeleitet, welche in der Datenbank nach einer Session mit dem entsprechenden Code sucht und je nach deren Existenz entweder wie unter C III. beschrieben weiter verfährt oder den User zurück zur Index-Seite leitet.

Selbiger Ablauf kann auch durch das Anhängen von z. B. "/abcdef" an die Base-URL ausgelöst werden.

2. Erstellen einer Session

Auf der Index-Seite findet der User ebenfalls die Möglichkeit, über den "create party"-Button eine eigene Session zu erstellen.

Hierzu leitet der Link auf die "login_spotify"-View weiter. In den ersten Entwicklungsschritten wurde anstatt dieses Schrittes direkt zur Settings-Seite umgeleitet, durch die Verwendung der Spotify-API wird allerdings die Anmeldung des Users bei Spotify nötig.

3. Log-in über Spotify

Für die Anmeldung wird zunächst ein “SpotifyOAuth” mit den mit folgenden Parametern generiert:

```
def create_spotify_oauth():
    return SpotifyOAuth(
        client_id='badxxxxxxxxxxxxxxxx',
        client_secret='c75xxxxxxxxxxxxxxxx',
        redirect_uri='http://127.0.0.1:8000/redirect/',
        scope='user-library-read' +
            'user-modify-playback-state' +
            'user-read-playback-state'
    )
```

“client_id” und “client_secret” sind für die Authentifizierung der zuvor unter “Spotify for Developers” registrierten App notwendig. Die “redirect_uri” dient der Umleitung nach einem erfolgreichen Spotify-Log-in, während das “scope” den Umfang des Zugriffs der Anwendung auf das Spotify-Konto des Users festlegt.

Dieser beläuft sich hierbei auf die Abfrage der User-Libraries, sowie das Lesen und Modifizieren des Play-back-Status’.

Zur Umsetzung der Anmeldung stellt die “Spotipy”-Library die Funktion “get_authorize_url()” zur Verfügung, welche den User mit dem Spotify-Login verbindet.

Nach erfolgreicher Anmeldung wird zur „redirect_page“-View umgeleitet.

Zusätzlich zur Anmeldung bei Spotify muss der User auch in der Anwendung registriert und angemeldet werden. Selbst wenn nach den Anforderungen kein expliziter Log-in der Nutzer erfolgen soll, ist es zur späteren Implementierung des Abstimmungssystems nötig, einzelne User eindeutig zu identifizieren. Hierzu kommt ein Custom-Usermodell zum Einsatz, bei dessen Nicht-Superusern das Passwort bei der Erstellung standardmäßig None gesetzt und der Benutzername zu Debugging-Zwecken durch eine UUID ersetzt wird.

```
class User(AbstractBaseUser):
    password = models.CharField(max_length=128, null=True)
    identifier = models.CharField(max_length=10, unique=True)
    USERNAME_FIELD = 'identifier'
    is_active = models.BooleanField(default=True)
    is_admin = models.BooleanField(default=False)
    is_staff = models.BooleanField(default=False)
    REQUIRED_FIELDS = []

    objects = UserManager()
```

Wichtig hierbei war es, die Funktionalität eines normalen Users mit Benutzernamen und Passwort für eventuelle Superuser zu erhalten. Für noch nicht angemeldete User kann also im Hintergrund ein neues Benutzerobjekt erstellt und angemeldet werden.

Für alle nachfolgenden Zugriffe auf das Spotify-Profil des Users wird ein sogenanntes “access_token” benötigt. Nachdem Django eine sehr breite Datenbank-Funktionalität bereitstellt, wird an dieser Stelle die “Spotipy”-eigene Speicherung des Tokens über Cookies umgangen und stattdessen ein entsprechendes Datenbank-Objekt erstellt.

Somit wird in der “redirect_page”-View die Datenbank zunächst nach einem bestehenden Token durchsucht. Ist dieses vorhanden, muss es jedoch je nach Ablaufzeit erneut mithilfe des „refresh_tokens“ verlängert werden.

Zudem muss die Exception eines ungültigen Tokens abgefangen werden, welches beispielsweise nach dem Zurücksetzen des eigenen Spotifyprofils in der Datenbank zurückbleiben kann.

In diesem Fall oder bei erstmaliger Anmeldung wird über die “Spotipy OAuth”-Funktion “get_access_token()” ein neues Token generiert und in der Datenbank gespeichert.

```
def redirect_page(request):
    no_token_saved = False
    ...
    try:
        if not get_user_token(request.user):
            no_token_saved = True
        # catch invalid refresh token exception and delete old tokens
    except SpotifyException:
        user_tokens = ApiToken.objects.filter(user=request.user)
        if user_tokens.exists():
            user_token = user_tokens[0]
            user_token.delete()
            no_token_saved = True
        # get new token from spotify-api
    if no_token_saved:
        sp_oauth = create_spotify_oauth()
        code = request.GET.get('code')
        token_info = sp_oauth.get_access_token(code=code, check_cache=False)
        # save new token to database
        api_token = ApiToken(access_token=token_info['access_token'],
                             refresh_token=token_info['refresh_token'],
                             expires_at=int(token_info['expires_at']),
                             user=request.user)
        api_token.save()
    return redirect(settings)
```

4. Abfrage der Playlists und Geräte

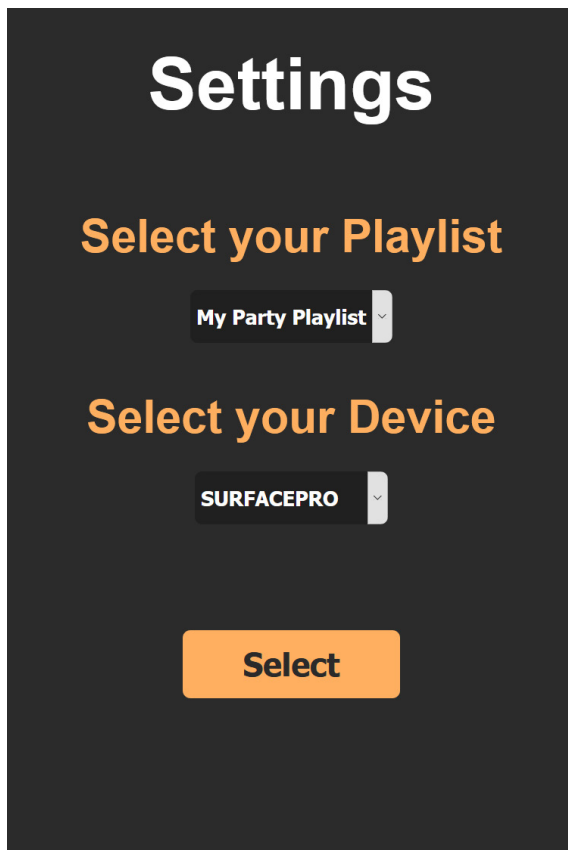
Bevor der Nutzer zur Settings-Seite umgeleitet werden kann, müssen die ihm zur Verfügung stehenden Wiedergabelisten und -geräte von der Spotify-API erfasst werden.

Die “Spotipy”-Funktionen “current_user_playlists()” und “devices()” erlauben die Abfrage dieser Daten, wobei die Selektion durch optionale Parameter spezifiziert werden kann.

Die jeweiligen Attribute zunächst als String empfangen und in ein JSON-Format umgewandelt, woraus die verschiedenen Datenbankobjekte abgeleitet und schlussendlich gespeichert werden. Nachdem die Anwendung in ihrer aktuellen Konfiguration auf vier wählbare und einen wiedergegebenen Song ausgelegt ist, werden dabei Playlisten mit weniger als fünf Liedern ignoriert.

Selbiges gilt für Wiedergabegeräte, die nicht aktiv oder eingeschränkt, also nicht für die Anwendung zugänglich sind.

II. Settings Seite



1. Auswahl Wiedergabeliste und -gerät

Nach dem erfolgreichen Log-in und Abfrage der verfügbaren Playlists und Wiedergabegeräte findet der Nutzer sich nun auf der Settings-Seite.

Hier können im Drop-down-Menü die gewünschte Wiedergabeliste und das Play-back-Device ausgewählt und anschließend die eigentliche Sitzung gestartet werden. Sollten keine geeigneten Geräte und/oder Playlists verfügbar sein, wird der User darauf durch entsprechende Fehlermeldungen hingewiesen.

2. Erstellung der Sitzung

Die gewählten Einstellungen werden beim Absenden des Formulars in den entsprechenden Datenbank-Objekten übernommen und ein "PartySession"-Objekt mit einem Zufallscode aus lowercase ASCII-Zeichen erstellt.

```
def create_session_code():
    characters = string.ascii_lowercase
    random_session_code = ''.join(random.choice(characters) for i in range(6))
    # checks if string is already in use
    if PartySession.objects.filter(session_code=random_session_code).exists():
        # create new string
        create_session_code()
    else:
        return random_session_code
```

Im Anschluss werden die zur ausgewählten Playlist gehörigen Songs durch die Spotify-API abgefragt und mit Relation zum zugehörigen PartySession-Objekt gespeichert.

Vor der Implementierung der API wurden an dieser Stelle stattdessen Mock-Datensätze erzeugt und in der Datenbank abgelegt.

Vor der letztendlichen Weiterleitung zur "party_session"-View wird für den User ein Relationship-Objekt erstellt, das neben der Zugehörigkeit des Users zu einer Session weitere Attribute wie beispielsweise die Host-Rolle oder die Stimme für einen bestimmten Song speichert.

III. Session Seite

Obleich eine eigene Sitzung erstellt oder einer bestehenden Session beigetreten wurde, gelangen alle Nutzer final zur "party_session"-View. Hier erfolgt sowohl der Log-in noch nicht angemeldeter Gast-Nutzer als auch das Anlegen eines entsprechenden Relationship-Objekts.

Handelt es sich beim Nutzer um den Host der Session, so wird das HTML-Dokument mit einem Formular zum Starten der Sitzung ausgeliefert.

1. Verbindung mit Websocket

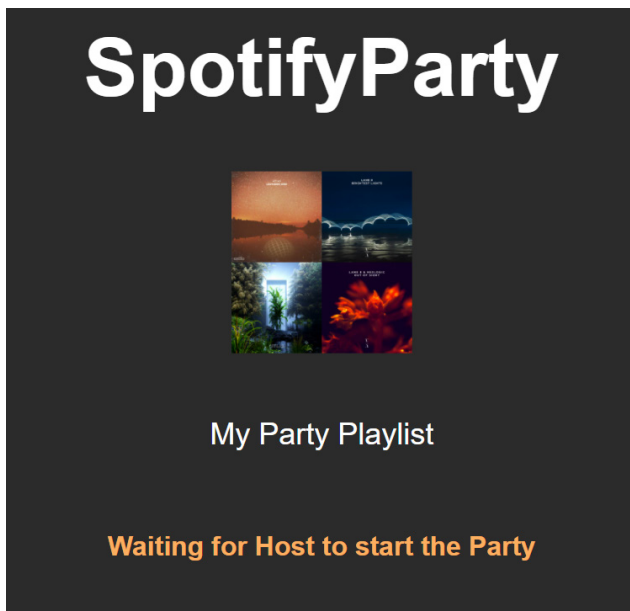
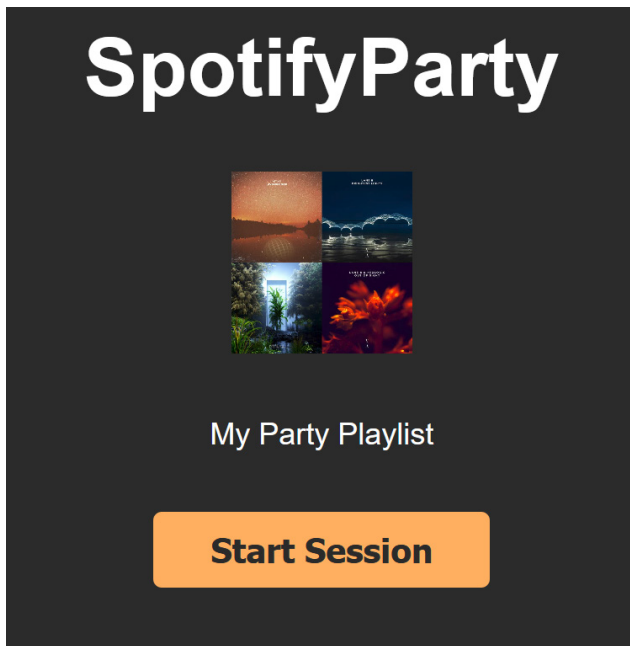
Vom Frontend aus muss nun eine Verbindung zum Websocket aufgebaut werden. Die Strings aus den Variablen "wsStart", "loc.host" und "loc.pathname" werden zu einem Pfad konkateniert, welcher den Client zu der spezifischen Websocket URL (r'(?P<room_name\w+)/\$' routet.

Dort ist der "SessionConsumer" für das Handling der Websocket-Messages zuständig.

```
let wsStart = 'ws://';
if (loc.protocol === 'https:') {
    wsStart = 'wss://';
}

let endpoint = wsStart + loc.host + loc.pathname; //
let socket = new WebSocket(endpoint);
```


2. Starten der Sitzung



```
async def collect_session_data(self, message_type):
    # get songs selected for playing and voting from database as dictionaries
    playing_song = await self.get_playing_song_dict(...)
    votable_songs = await self.get_votable_songs_dict(...)

    # create dictionary with data from above
    collected_data = {
        "type": message_type,
        "playing_song": playing_song,
        "votable_songs": votable_songs
    }

    if message_type == 'session_init':
        current_session = await self.get_current_party_session(self.room_name)
        current_session.is_initialized = True
        await database_sync_to_async(current_session.save)()
        # send initial data to all users in session
        playback_started = await self.record_playback_start(self.room_name)
        collected_data["playback_started"] = playback_started
        asyncio.create_task(self.send_to_session_task(collected_data, message_type))
        # start playback
        await self.play_song()

    ...
```

Im Frontend wird das JSON-Dump aus der "session_init"-Message nun in die entsprechenden HTML-Elemente übersetzt.

```
socket.onmessage = function (e) {
    let message = JSON.parse(e.data);
    let message_text = message.text;

    console.log(message)

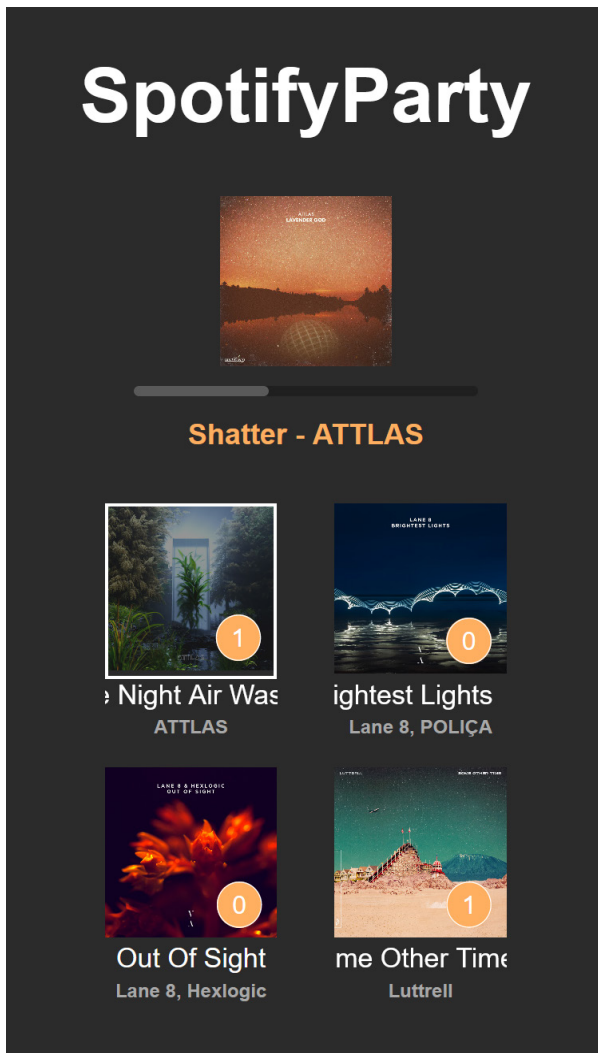
    // session is initialized
    if (message_text["type"] == "session_init" ||
        message_text["type"] == 'user_session_init') {
        fillCards(message_text);
    }
}
```

Nachdem Nutzer auch erst nach der Initialisierung der Session beitreten können ist es notwendig für diese User ebenfalls die Daten der Sitzung an das Frontend zu übergeben. Hierzu dient die Websocket-Message "user_session_init", welche ähnlich zu "session_init" die erforderlichen Informationen zusammengestellt, jedoch nur an einen einzelnen User gesendet wird.

Zu Beginn einer Session muss diese zunächst durch den Host gestartet werden. Dies ermöglicht es bereits kurz vor einer Feier eine Sitzung zu erstellen, sie mit Gästen zu teilen. Somit kann diesen die Möglichkeit gegeben werden, beizutreten, bevor die Abstimmung begonnen wird.

Das Starten der Session sendet die Nachricht "start_party_session" an den Consumer welcher daraufhin den ersten Song der Playlist zum Abspielen sowie vier weitere Lieder zu Abstimmung auswählt. Die Rollen der Songs werden entsprechend in den Datenbankobjekten gespeichert und die für das Frontend erheblichen Informationen in einem Dictionary gesammelt und als JSON-Dump an alle in der Sitzung befindlichen User gesendet.

3. Abstimmung und Auszählung



Auf der Hauptseite kann nun durch Klick auf eine der Kacheln eine Stimme für den jeweiligen Song abgegeben werden.

Hierbei sendet ein Eventlistener die "id" der Kachel, welche mit einer "spotify_song_id" übereinstimmt, als einfachen String an den Consumer.

Dort werden, soweit das Voting aktiviert ist, alle Strings, die nicht "start_party_session" entsprechen, an die "change_vote"-Methode des jeweiligen Relationship-Objekts des Users übergeben. Hier wird überprüft, ob es sich bei dem String um eine valide "spotify_song_id" handelt.

Ist dies der Fall, wird die neue Stimme im "UserJoinedSession"-Objekt gespeichert und automatisch das "song_votes"-Attribut des gewählten "Song"-Objekts erhöht.

Hat der User bereits zuvor einen Vote abgegeben, muss der Stimmenzähler dieses Songs um eins erniedrigt werden. Handelt es sich beim vorherigen und neuen Lied um das gleiche Objekt, wird lediglich der Vote-Count erniedrigt und anschließend das "user_vote"-Attribut des Nutzers „None“ gesetzt, um das Zurücknehmen einer Stimme zu ermöglichen.

```
def change_vote(self, spotify_song_id):
    song = Song.objects.filter(spotify_song_id=spotify_song_id,
                              party_session=self.party_session,
                              is_votable=True)

    if song.exists():
        voted_song = song[0]
        # remove one vote from old song if exists
        if self.user_vote is not None:
            old_song = self.user_vote
            old_song.song_votes = old_song.song_votes - 1
            old_song.save()

        # add one vote to new song
        # and save as voted song if exists
        if not self.user_vote == voted_song:
            voted_song.song_votes = voted_song.song_votes + 1
            voted_song.save()
            self.user_vote = voted_song
            self.save()

        # remove user vote if already voted-for song
        # has been clicked again
        else:
            self.user_vote = None
            self.save()

    return True
return False
```

Nach der Auszählung der Stimmen und Wahl der nächsten Wiedergabe müssen vier neue Lieder zur Abstimmung ausgewählt werden. Das Attribut "was_played" sorgt hierbei dafür, dass bereits abgespielte Songs erst nachdem weniger als vier neue Tracks zur Auswahl stehen erneut wählbar werden.

5. Verlassen des Websockets

Beim Abbruch der Verbindung zu Websocket muss festgestellt werden, ob es sich beim verlassenden User um den Host der Sitzung handelt.

Ist dies nicht der Fall, kann schlicht das entsprechende Relationship-Objekt gelöscht werden, wodurch ein "Django-Signal" des "Song"-Objekts die Stimmenzahl des Liedes, für das der Nutzer gestimmt hatte um eins erniedrigt, sofern eine Stimme vorhanden ist.

```
@receiver(pre_delete, sender=UserJoinedPartySession)
def remove_vote_on_user_leave_party_session(instance, **kwargs):
    if instance.user_vote:
        song = Song.objects.filter(
            spotify_song_id=instance.user_vote.spotify_song_id,
            party_session=instance.party_session)[0]
        song.song_votes = song.song_votes - 1
        song.save()
```

Verlässt jedoch der Host die Session, wird die gesamte Channel-Gruppe gelöscht, die Websocket-Verbindungen aller Gäste beendet und die „Party-Session“ Instanz kaskadierend gelöscht, wodurch auch alle Relationship- und "Song"-Objekte aus der Datenbank entfernt werden.

Wird die Verbindung zum Websocket beendet, so wird der Nutzer durch das Frontend wieder zur Index-Seite der Anwendung zurückgeleitet.

D Inbetriebnahme

I. Voraussetzungen

Python installiert
Docker installiert
Spotify Premium Account vorhanden

II. Lokale Inbetriebnahme

1. Klonen des Repositorys aus GitLab, oder:
ZIP-Ordner

2. Installieren der Packages mit:
"pip install -r requirements.txt"

bei fehlgeschlagener Installation:

Manuelle Installation der Datei "Twisted-20.3.0-cp39-cp39-win_amd64.whl"

Installation aktueller Version der Buildtools für Visual Studio 2019

Erneutes Installieren der Packages mit:

"pip install -r requirements.txt"

3. Initialisierung der Datenbank mit:
"...\\projectRoot\\website>python manage.py makemigrations"
"...\\projectRoot\\website>python manage.py migrate"

4. Starten von Docker

5. Starten des Redis Servers mit:
"...\\projectRoot\\website>docker run -p 6379:6379 -d redis:5"

6. Starten des lokalen Servers mit:
"...\\projectRoot\\website>python manage.py runserver"

Vermeiden von "Datenbankmüll":

Verlassen aktiver Sitzungen vor dem Beenden des lokalen Servers

E Fazit

I. Umsetzung der Anforderungen

Nachdem während der gesamten Entwicklung auf die Bezugnahme zu den zuvor definierten Anforderungen geachtet wurde, konnten diese gezieht umgesetzt werden.

Die grundlegenden Muss-Kriterien, wie in 1.2. beschrieben, werden vom aktuellen Stand der Applikation vollständig erfüllt.

Auch die Wunsch-Kriterien sind grundsätzlich bereits gegeben, wenngleich die Entwicklung der Anwendung keinesfalls abgeschlossen ist.

II. Erweiterungsmöglichkeiten

Die Anwendung bietet somit eine gute Grundlage hinsichtlich zukünftiger Erweiterungsmöglichkeiten. So können den Nutzern einige der momentan hardgecodeten Voreinstellungen wie z. B. die Anzahl der zur Abstimmung stehenden Tracks oder die Laufzeit des Votings als konfigurierbare Optionen zur Verfügung gestellt werden.

Einer der nächsten Entwicklungsschritte wäre es, die Implementierung des zurzeit eher prototypischen Designs mithilfe eines Frameworks wie z. B. "Bootstrap" oder "Tailwind CSS" umzusetzen.

Außerdem wäre die Implementierung einiger Features der „Konkurrenz“-Anwendungen, wie beispielsweise das Streaming der Wiedergabe zu Gast-Nutzern für sozial distanzierte Feiern oder etwa die Möglichkeit, weitere Songs außerhalb der Playlist zur Abstimmung vorzuschlagen, denkbar.

Alle möglichen Erweiterungen sollten allerdings kritisch hinsichtlich den obersten Anforderungen einfacher Bedienung und geringer Einstiegshürde betrachtet werden.

III. Persönlicher Eindruck

Martin:

Das Spannende am Kurs „Agile Webanwendungen mit Python“ war das themenübergreifende Arbeiten. Von HTML/CSS über Python/Django bis hin zum Datenbankmanagement war ein großes Spektrum an Know-how erforderlich. Des Weiteren musste zum ersten Mal langfristig eine Projektplanung im Team aufgesetzt werden. Durch vorangegangene Team-Projekte und bekannte Fähigkeiten konnten wir die Aufgabenteilung unkompliziert durchführen.

Die ersten Wochen wurden individuell genutzt, um Python und Django kennenzulernen.

Wie beschrieben wurden danach phasenweise Komponenten entwickelt und in das Projekt eingefügt. Sehr gut ist es uns gelungen, diese Features verständlich für das andere Teammitglied zu programmieren. Im laufenden Prozess verschmolzen die erst eigens entwickelten Aufgabenfelder mehr und mehr in ein Teamgefüge. Die vorgenommenen Problemstellungen konnten gelöst und das Projekt kompromisslos fertiggestellt werden.

Das Projekt bietet eine stabile und gut durchdachte Basis. Für die Zukunft wäre es so denkbar, die SpotifyParty mit neuen Features zu erweitern.

Moritz:

Für mich war dies das erste Projekt dieser Größenordnung, was vor allem zu Beginn eine große Herausforderung darstellte.

Die sehr freie Wahl des Projektthemas erlaubte die in den bisherigen Semestern erlernten Programmier- und Softwareentwicklungsfähigkeiten anzuwenden und auszubauen sowie mit etlichen neuen Technologien, allen voran „Django“ und „DjangoChannels“, vertraut zu werden.

Auch die Zusammenarbeit im Team funktionierte stets gut organisiert und zuverlässig.

Insgesamt sind wir besonders in Anbetracht des Zeitrahmens sehr zufrieden mit dem Stand unserer Anwendung und durchaus daran interessiert, diese auch weiterhin in derselben Teamkonstellation bis hin zu einer möglichen tatsächlichen Inbetriebnahme weiterzuentwickeln.

F Quellen

Django:

<https://www.djangoproject.com/>

Django Channels:

<https://channels.readthedocs.io/en/stable/>

Spotipy:

<https://spotipy.readthedocs.io/en/2.16.1/>

Docker:

<https://www.docker.com/>

Random Session Code (Grundlage):

<https://pynative.com/python-generate-random-string/>

Marquee Effekt (Grundlage):

<https://stackoverflow.com/questions/21233033/how-can-i-create-a-marquee-effect/56593910#56593910>

<https://stackoverflow.com/questions/9333379/check-if-an-elements-content-is-overflowing/9541579#9541579>

<https://stackoverflow.com/questions/44482706/how-to-track-down-elements-causing-horizontal-scroll-on-a-web-Seite>

Notification Badges (Grundlage):

https://www.w3schools.com/howto/howto_css_notification_button.asp

Carbon (Styling Codeschnipsel):

<https://carbon.now.sh/>

Recherche/ Inspiration:

<https://festify.rocks/>

<https://jukestar.mobi/>

<https://kahoot.it/>

<https://skribbl.io/>

Codeverwaltung:

<https://r-n-d.informatik.hs-augsburg.de:8080/mo-kro/django-spotifyparty-ws2020>

Styleguide:

<https://www.figma.com/file/amac8khjTATR3T-xKTaFJIU/SpotifyParty-StyleGuide?node-id=0%3A1>