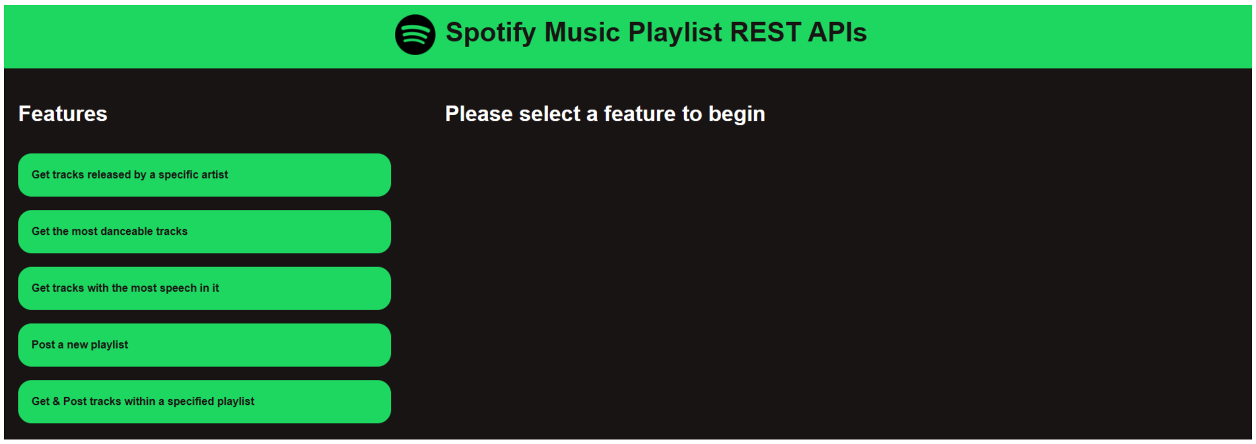
CM3035 Advanced Web Development MidTerm Report


Spotify Music Playlist REST APIs
Features
Please select a feature to begin
Get tracks released by a specific artist
Get the most danceable tracks
Get tracks with the most speech in it
Post a new playlist
Get & Post tracks within a specified playlist

Submission Date: 7 January 2024

Word Count: 2045

Table of Contents

1 Application Access

Running the deployed application

The application has been deployed on an AWS EC2 cloud instance with an NGINX web server. You may access the application through this link:
http://ec2-18-209-6-8.compute-1.amazonaws.com:8000/

Running the application locally

To run the application, unzip the submitted file (spotify.zip) and open the extracted folder in a code editor (e.g. Visual Studio Code). The project was developed with the Windows 11 operating system, using python version 3.11.3. Once you have activated your own virtual environment, ensure you are in the project's root directory (spotify folder) and install all packages from the requirements.txt file. You can do this by running the following command in the terminal: ***pip install -r requirements.txt***.

If you need to access the django admin site, please use the following superuser login details:

- Username: admin
- Password: 37wtkxz128

2 Data

About the Dataset

The data for this project was taken from a Kaggle dataset posted by Joakin Arvidsson of almost 30,000 tracks from the Spotify API. The dataset contains track and playlist details as well as computed audio features of each track. The computed audio features in this table makes the data interesting as it allows us to make queries based on quantified features that are not usually available when analysing music.

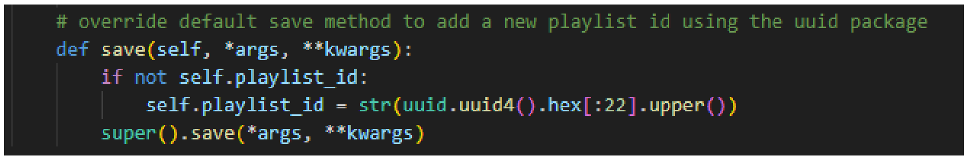
With nearly 30,000 tracks, the original dataset contained a total of over 700,000 entries. Therefore, for the purpose of this project, I only included tracks and playlists associated with the 10 artists that had more than 60 tracks in the dataset. To ensure efficiency of database queries, the dataset was also normalised from 1 single table into 3 unique tables. This helps to reduce redundancy and ensures data integrity. Therefore, the data loaded to the application comes from 3 separate csv files. In total, since only columns related to the API endpoints created in this project were kept, there were a total of 8,852 entries across the tables.

Data Model & Migrations (R2 / R1A)

The following section details the models created for each data table. The classes Tracks, Playlists, and PlaylistTracks in the **models.py file** handles the creation of these data models.

The table "**Tracks**" details the key features of each track, including 3 quantitative features that can be used to describe the song - danceability, wordiness, and duration in milliseconds.

Spotify allows users to create public playlists containing tracks across various artists and albums. The table "**Playlists**" contains generic details of these playlists. Its genre can have one of the following values: pop, rap, rock, latin, r&b, and edm. To ensure a unique ID is generated when new records are added by users, I overwrote the default save method, replacing the playlist_id with a UUID string with 22 alpha-numeric characters.



```
    # override default save method to add a new playlist id using the uuid package
    def save(self, *args, **kwargs):
        if not self.playlist_id:
            self.playlist_id = str(uuid.uuid4().hex[:22].upper())
        super().save(*args, **kwargs)
```

The "**PlaylistTracks**" table contains the mapping of tracks to playlists. Since there can be multiple playlists with the same track, and multiple tracks within each playlist, this table serves as a juncture table, with a many-to-many relationship between the 2 foreign key variables of the playlists and tracks tables. Since there should be only 1 record for each track_id and playlist_id combination, a unique constraint was added to the data model.

After these models were created, a migration file was generated using the ***python manage.py makemigrations*** command. To execute the creation of tables into our SQLite database, the

python manage.py migrate command was run. This was performed after any modifications to the data model was made.

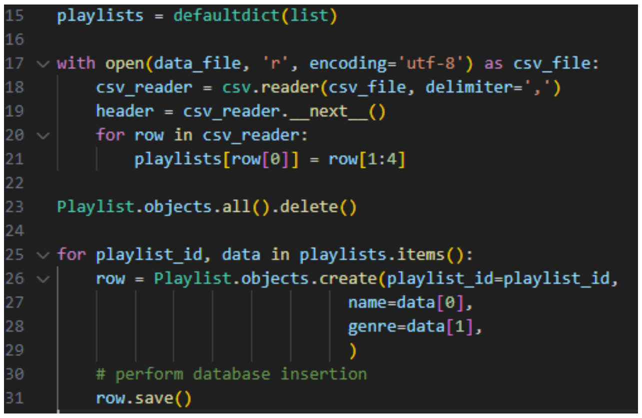
Bulk Loading Data (R4)

There are 720 rows and 8 columns (6,240 entries) seeded into the database for the **Tracks** table. Since the data is loaded from a CSV file, all entries with commas in it had their commas removed. The track Just For One Day (Heroes) (Extended Version) by David Guetta did not have a release date. This was updated based on a search on Google of its release date.

There are 168 playlists and 3 columns (579 entries) seeded into the database for the **Playlist** table, but new playlists can be added through a HTTP POST method. To prevent complexities from character encoding in the SQLite database, I also removed all emoji characters.

The **PlaylistTracks** table was seeded with 1054 rows of track-playlist combinations within its 2 foreign key columns (2108 entries), but new tracks can be added to specific playlists through a HTTP POST method.

Each of the 3 csv files were parsed and its data stored to its respective database tables through 3 separate scripts. All csv and python files for data loading can be found in "**spotify\scripts**". Since PlaylistTracks is a junction table, populate_playlisttrack.py should only be run after populate_track.py and populate_playlist.py has been run. For each of the scripts, the csv reader delimits the data by commas, skips the heading row, then appends each column of the row into a temporary dictionary with each ID recorded as the dictionary's key (*see lines 15-21 below*). Previous entries were removed from the database (see line 23), and for each key in the temporary dictionary, a new record is created and inserted into the database (see lines 25-31).



```
15  playlists = defaultdict(list)
16
17  with open(data_file, 'r', encoding='utf-8') as csv_file:
18      csv_reader = csv.reader(csv_file, delimiter=',')
19      header = csv_reader.__next__()
20      for row in csv_reader:
21          playlists[row[0]] = row[1:4]
22
23  Playlist.objects.all().delete()
24
25  for playlist_id, data in playlists.items():
26      row = Playlist.objects.create(playlist_id=playlist_id,
27                                    name=data[0],
28                                    genre=data[1],
29                                    )
30      # perform database insertion
31      row.save()
```

To reload all data to the application from the CSV files, please run the following commands:

1. python .\scripts\reset.py	2. python .\scripts\populate_track.py
3. python .\scripts\populate_playlist.py	4. python .\scripts\populate_playlisttrack.py

3 URLs & Endpoints (R1B / R1C / R1D / R3)

Each table above had a corresponding serializer which adopts the model's metadata. The TrackSerializer returns a subset of columns that users will understand. The ID values were added so that users can note down the track_id of tracks from the GET endpoints, which they can use to add new tracks to their playlists. The root page of the project's music application showcases the features implemented in this project, as showcased in this section.

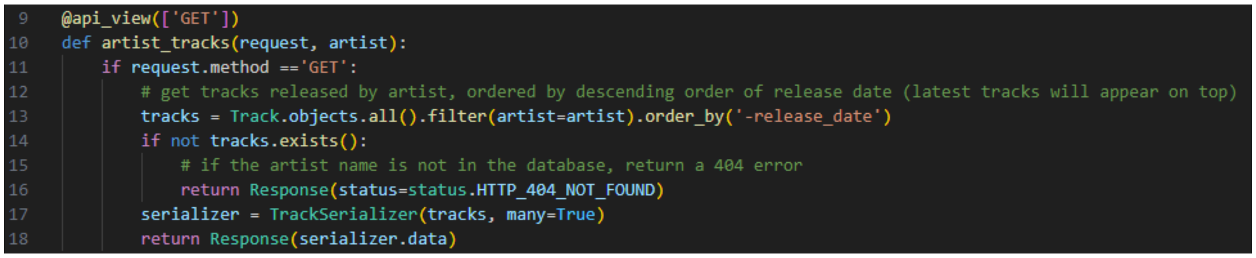
Get tracks released by a specific artist

Web URL Routing: /artists

API URL Routing: /api/tracks/artist/<str:artist>

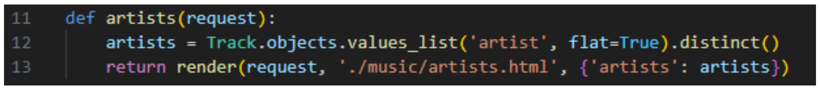
Serializer: TrackSerializer

This endpoint allows users to find a list of tracks released by a specific artist. This was achieved by implementing a Django REST API using the djangorestframework python package. The endpoint only accepts the GET method. 2 columns of the Track table were used - the artist field was used as a filter, and the release date was used to sort the data, with the most recent releases appearing at the top of the results.



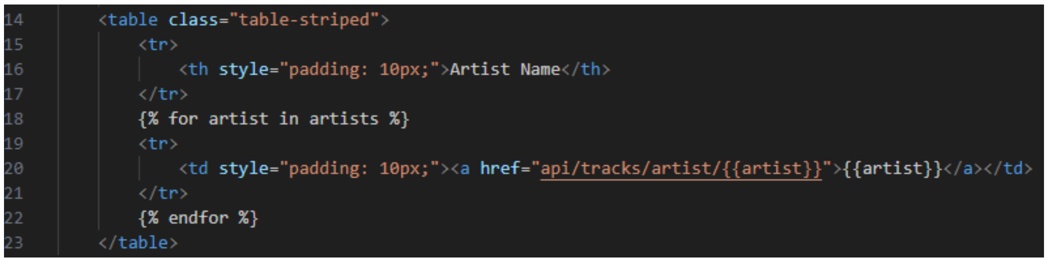
```python
@api_view(['GET'])
def artist_tracks(request, artist):
    if request.method =='GET':
        # get tracks released by artist, ordered by descending order of release date (latest tracks will appear on top)
        tracks = Track.objects.all().filter(artist=artist).order_by('-release_date')
        if not tracks.exists():
            # if the artist name is not in the database, return a 404 error
            return Response(status=status.HTTP_404_NOT_FOUND)
        serializer = TrackSerializer(tracks, many=True)
        return Response(serializer.data)
```

Since the artists' names are case and space sensitive, I pulled a list of valid artist name values using the values_list() and distinct() methods on the Track table. This prevents users from misspelling artist names in their requests.



```python
def artists(request):
    artists = Track.objects.values_list('artist', flat=True).distinct()
    return render(request, './music/artists.html', {'artists': artists})
```

The artists' names were rendered with a template that creates a table entry for each artist, along with a link that directs users to its respective API endpoints.

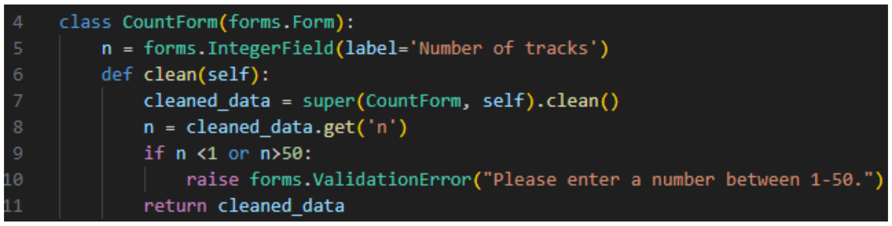


```html
    <table class="table-striped">
        <tr>
            <th style="padding: 10px;">Artist Name</th>
        </tr>
        {% for artist in artists %}
        <tr>
            <td style="padding: 10px;"><a href="api/tracks/artist/{{artist}}">{{artist}}</a></td>
        </tr>
        {% endfor %}
    </table>
```

Get the most danceable tracks / tracks with the most speech in it

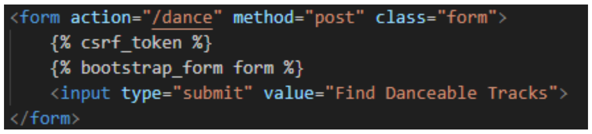
Web URL Routing: /dance **API URL Routing:** api/tracks/dance/<int:n> **Serializer**: TrackSerializer	**Web URL Routing:** /wordy **API URL Routing:** /api/tracks/wordy/<int:n> **Serializer**: TrackSerializer

The dance endpoint allows users to identify the most danceable tracks in the database, using the danceability index of the dataset. The wordy endpoint allows users to identify the tracks with the most speech in it. This would often return rap songs as it typically has a higher speech to music ratio. Both the endpoints only accept the GET methods. The integer n refers to the number of tracks to be returned to the user. To help users use this endpoint, I created a simple form, CountForm. This form will allow users to enter the number of tracks they would like to filter to through the web interface. To prevent too many records to be returned, I added a validation to limit the number of records to be between 1 and 50.

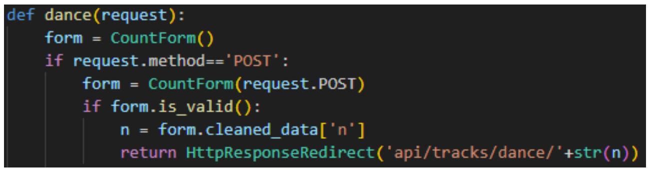


```python
class CountForm(forms.Form):
    n = forms.IntegerField(label='Number of tracks')
    def clean(self):
        cleaned_data = super(CountForm, self).clean()
        n = cleaned_data.get('n')
        if n <1 or n>50:
            raise forms.ValidationError("Please enter a number between 1-50.")
        return cleaned_data
```

Submitting the form with trigger a POST request, redirecting the user to the API endpoint with the number of tracks submitted through the form.



```html
<form action="/dance" method="post" class="form">
    {% csrf_token %}
    {% bootstrap_form form %}
    <input type="submit" value="Find Danceable Tracks">
</form>
```



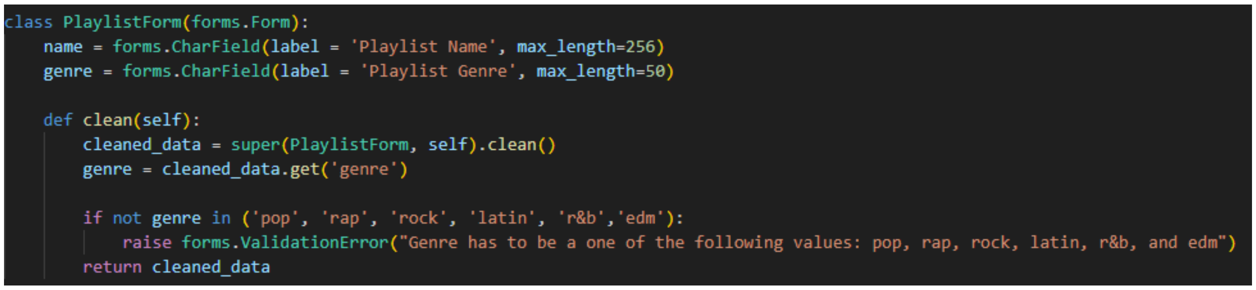
```python
def dance(request):
    form = CountForm()
    if request.method=='POST':
        form = CountForm(request.POST)
        if form.is_valid():
            n = form.cleaned_data['n']
            return HttpResponseRedirect('api/tracks/dance/'+str(n))
```

Post a new playlist

Web URL Routing: /create_playlist/

Serializer: PlaylistSerializer

This feature enables users to add a new playlist to the database. This creates an empty playlist with no tracks within it yet. Each playlist has a genre assigned to it, which can only be one of 6 acceptable values. Therefore, I created a form with validation to restrict the values submitted to the database.

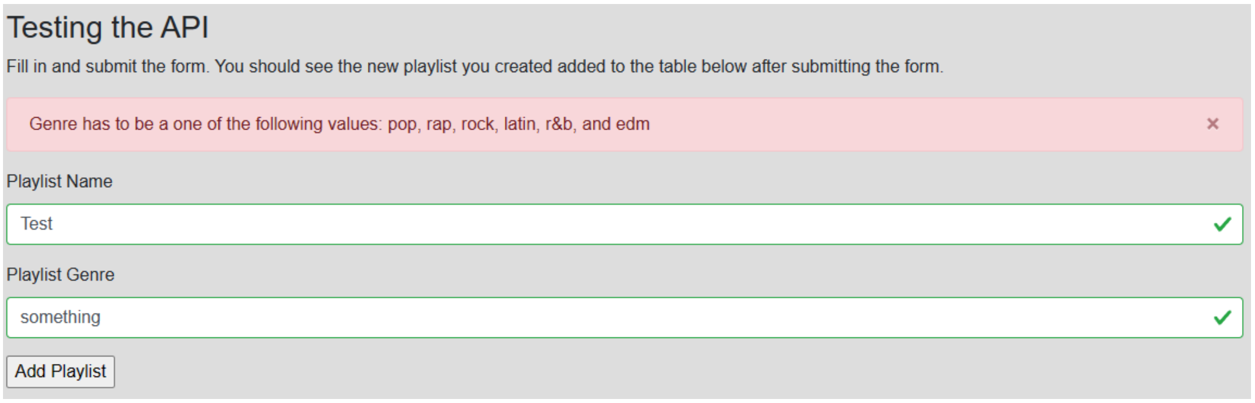


```python
class PlaylistForm(forms.Form):
    name = forms.CharField(label = 'Playlist Name', max_length=256)
    genre = forms.CharField(label = 'Playlist Genre', max_length=50)

    def clean(self):
        cleaned_data = super(PlaylistForm, self).clean()
        genre = cleaned_data.get('genre')

        if not genre in ('pop', 'rap', 'rock', 'latin', 'r&b','edm'):
            raise forms.ValidationError("Genre has to be a one of the following values: pop, rap, rock, latin, r&b, and edm")
        return cleaned_data
```

With the validation implemented, genre entries that do not match the list of acceptable values will not be added to the database. Instead, a warning will be shown to the user along with the list of values that can be accepted.



Testing the API

Fill in and submit the form. You should see the new playlist you created added to the table below after submitting the form.

Genre has to be a one of the following values: pop, rap, rock, latin, r&b, and edm ×

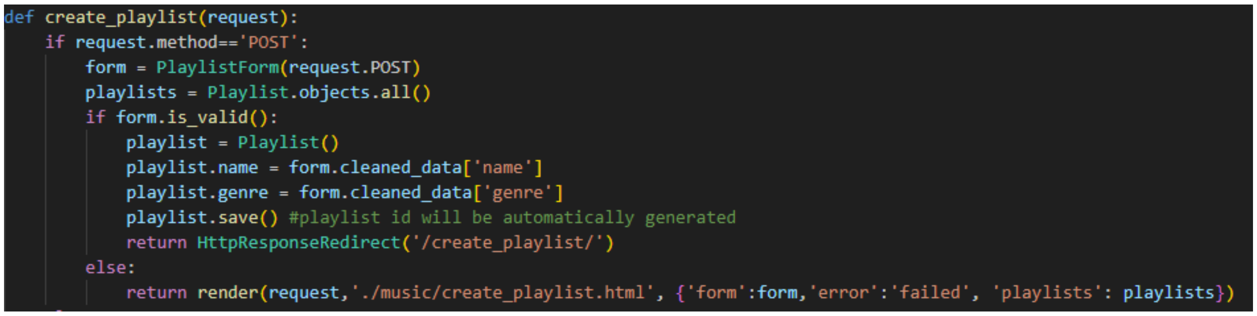
Playlist Name

Test ✓

Playlist Genre

something ✓

If the values entered are compliant, a new record will be added to the database with a unique HEX value (see Data Model & Migrations section for more details). A successful POST method would redirect users back to the same page.



```python
def create_playlist(request):
    if request.method=='POST':
        form = PlaylistForm(request.POST)
        playlists = Playlist.objects.all()
        if form.is_valid():
            playlist = Playlist()
            playlist.name = form.cleaned_data['name']
            playlist.genre = form.cleaned_data['genre']
            playlist.save() #playlist id will be automatically generated
            return HttpResponseRedirect('/create_playlist/')
        else:
            return render(request,'./music/create_playlist.html', {'form':form,'error':'failed', 'playlists': playlists})
```

As I have added a table listing all playlists in the create_playlist.html template, users will be able to see the playlist they had just added at the bottom of the page.

Get & Post tracks within a specified playlist

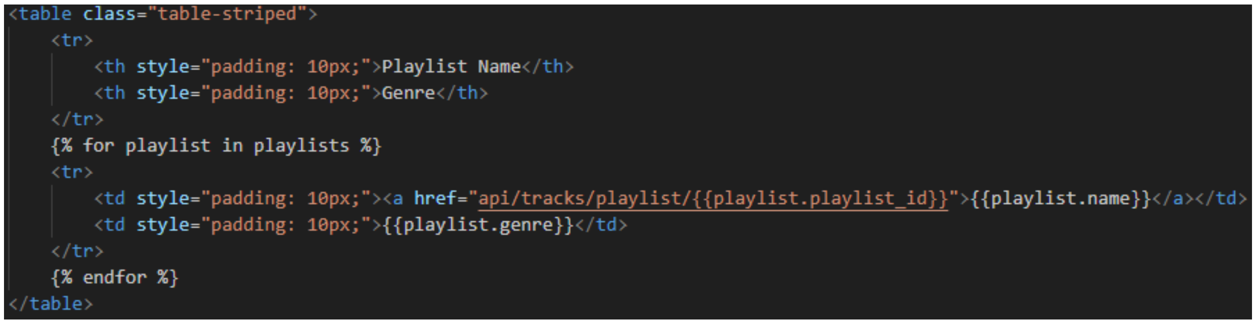
Web URL Routing: /playlists

API URL Routing: /api/tracks/playlist/<str:playlist_id>

Serializer: PlaylistTrackSerializer

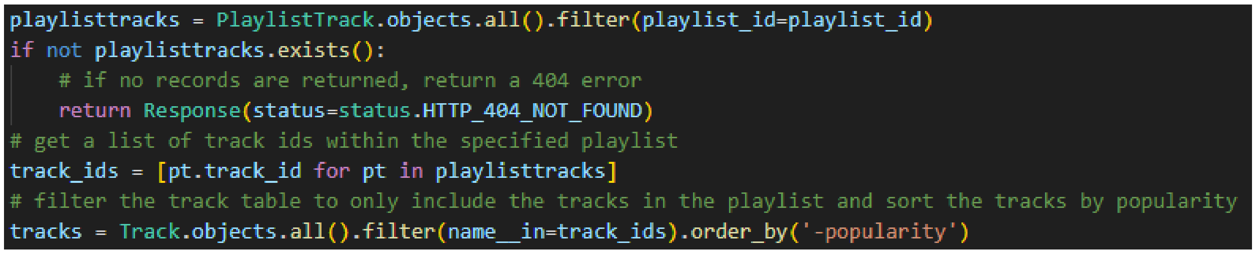
This endpoint had the highest degree of complexity in its implementation. It allows users to get a list of all tracks within a specific playlist, and post a new track into the playlist. As the PlaylistTrack table only includes foreign keys, this endpoint utilises all 3 tables to provide users with insightful and understandable data.

The web interface displays a table of all playlists available, along with its genre. This requires us to utilise the Playlist table. If the user had added a new playlist previously, it will be displayed at the bottom of the table. Each playlist has a unique link that directs them to the API endpoint of the playlist selected. Notice that while the playlist name is used for display in the table, the link actually contains its associated playlist_id.



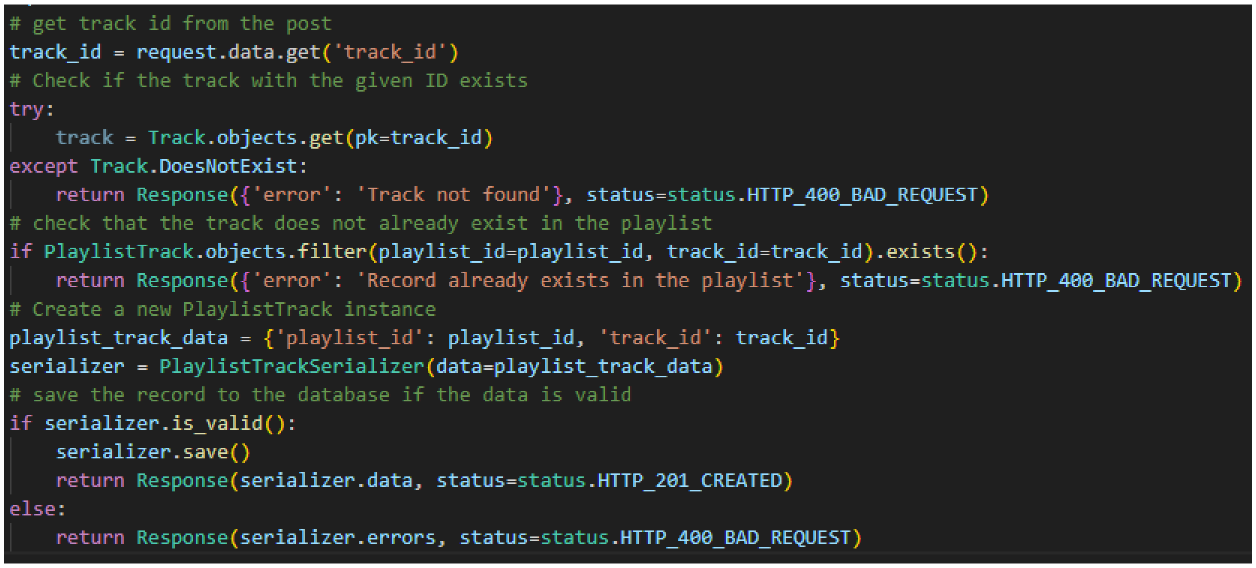
```
<table class="table-striped">
    <tr>
        <th style="padding: 10px;">Playlist Name</th>
        <th style="padding: 10px;">Genre</th>
    </tr>
    {% for playlist in playlists %}
    <tr>
        <td style="padding: 10px;"><a href="api/tracks/playlist/{{playlist.playlist_id}}">{{playlist.name}}</a></td>
        <td style="padding: 10px;">{{playlist.genre}}</td>
    </tr>
    {% endfor %}
</table>
```

There are 2 methods accepted to this endpoint - the GET and POST methods. The GET method first filters all records of the PlaylistTrack table associated with the specified playlist. With a python list comprehension, we get the list of all track ids in the playlist from the PlaylistTrack table. We can then filter the Track table to match the list of tracks in the python list. The order_by method helps us order the results by descending order of popularity, so that the most popular songs appear at the top of the returned results.



```
playlisttracks = PlaylistTrack.objects.all().filter(playlist_id=playlist_id)
if not playlisttracks.exists():
    # if no records are returned, return a 404 error
    return Response(status=status.HTTP_404_NOT_FOUND)
# get a list of track ids within the specified playlist
track_ids = [pt.track_id for pt in playlisttracks]
# filter the track table to only include the tracks in the playlist and sort the tracks by popularity
tracks = Track.objects.all().filter(name__in=track_ids).order_by('-popularity')
```

The POST method allows users to add new records to an existing playlist. It takes the track_id value from the request and checks that it is an actual record in the Track table. Since a track should not be added to a playlist twice, we check to ensure the track id is not already included in this playlist. If either of these conditions fail, a 400 bad request status is returned. Otherwise, the data is parsed through the PlaylistTrack serializer, and saved to the database.



```python
# get track id from the post
track_id = request.data.get('track_id')
# Check if the track with the given ID exists
try:
    track = Track.objects.get(pk=track_id)
except Track.DoesNotExist:
    return Response({'error': 'Track not found'}, status=status.HTTP_400_BAD_REQUEST)
# check that the track does not already exist in the playlist
if PlaylistTrack.objects.filter(playlist_id=playlist_id, track_id=track_id).exists():
    return Response({'error': 'Record already exists in the playlist'}, status=status.HTTP_400_BAD_REQUEST)
# Create a new PlaylistTrack instance
playlist_track_data = {'playlist_id': playlist_id, 'track_id': track_id}
serializer = PlaylistTrackSerializer(data=playlist_track_data)
# save the record to the database if the data is valid
if serializer.is_valid():
    serializer.save()
    return Response(serializer.data, status=status.HTTP_201_CREATED)
else:
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

4 Unit Testing (R1E)

Test fixtures to populate the database with arbitrary data were created using the factory_boy python package to achieve repeatability of this task during tests. They are added via the model_factories.py file. Since the Faker package was installed along with factory_boy, I used it to create fake ids, names, and dates. The random python package was used to create random integer and 2 decimal float values, as well as choose randomly between the acceptable genre values for the playlist. This helps us create a more dynamic test.

Test cases were written in the test.py file. The setUp function creates the tables and stores good and bad url values that will be used across tests. As we have 2 test classes (one for the APIs and the other for the serializers), the tearDown function was important for ensuring the database is cleared before the next test is run. I tested the GET & POST APIs for status code, response structure, and response value accuracy. I also tested the serializers for its structure and data value accuracy.

To run the tests, simply run the following command in the spotify root folder:

```
python manage.py test
```