jcol984 & jhar897 Design Report - Recommendations

For our assignment, we introduced a Music recommendation system that recommends up to 10 new songs when a user reviews a track highly.

RECOMMENDATIONS

Showing 10 new recommendations!

Track ID	Track Name	Artist	Album	Duration	Reviews
2077	Southshore	Thomas Dimuzio	Mono M::P Free	2:44	0.0/5 (0)
771	Brittle Woman	Glass Candy	Live at Death Disco 9/26/2001	3:02	0.0/5 (0)
1171	Black Market	Mors Ontologica	The Used Kids Session	2:46	0.0/5 (0)
674	Enterprizin	Fanatic	Change Thoughts (radio edit)	4:13	0.0/5 (0)
1048	Blue Coochology	Little Howlin' Wolf	Brave Nu World	3:00	0.0/5 (0)
1842	Mail Fraud	The Minor Thirds	Saskatchewan	2:52	0.0/5 (0)
3316	1	Big Digits	Live at WFMU on Marty McSorley's Show on 5/14/2008	1:53	0.0/5 (0)
1079	Claws That Bite	Lungs Of A Giant	Headfirst Handshakes	2:54	0.0/5 (0)
1158	Comeing Down	Mors Ontologica	The Used Kids Session	2:37	0.0/5 (0)
815	EinHauntedSommerplatz	Hans Gr??sel's Krankenkabinet	unreleased mp3s from Hans Gr??sel's	2:56	0.0/5 (0)

The main function (recommend_tracks) is called with a User object and accesses the user reviews stored in self.__reviews. The list of reviews is used to find all reviews rated 4 or higher to append to a list called self.__high_reviewed_tracks. Once all the user's reviews have been iterated through, the high_reviewed_tracks and a stored list of all tracks are looped through together in a nested for-loop.

Within the nested for-loop, sub-functions (self.recommend_from_artist, self.recommend_from_genre, self.recommend_from_album, self.recommend_from_duration) are run with their parameters being each track within high_reviewed_track and each track within the list of all tracks. Each function will return tracks matching the field in their name while also ensuring that the chosen track object is not already within the return list and not the initial track that has been highly reviewed.

Once the list is generated with all relative tracks, it is randomly shuffled and 10 songs are returned from the list as recommended tracks for the user. The 10 songs are picked randomly so that the user is not provided with the same recommendations every time, and also not provided recommendations from only one field (artist, album, genre, duration).

Initially, the program was created to recommend a track that fit within a set duration of the highly reviewed track, shared an artist or album, and shared at least one genre. Though we found this method was not reliable enough and often could not return enough recommendations; if any.

Our implementation follows the Dependency Inversion principle. Through the usage of a service layer (high-level module) method, browse_service.recommend_tracks(), we call the Abstract Repository interface (abstraction), which implements the recommend_tracks() method from the Memory Repository (low-level module). This ensures that high-level modules are not dependent on low-level modules and that the abstraction does not depend on details.

The view access to the recommendation web page is under the browse Blueprint, which houses all of the web pages involved with tracks. This ensures that changes to the domain model won't necessitate changes to the view layer. In addition to this, the service component that powers the recommendation method can be reused with other view layer methods (such as returning a JSON or an XML view instead of HTML.)

It's essential that, when the user attempts to access the recommended page, they're signed into a valid user account. Our code tests two cases when the user sends a get request for the recommended page:

- If the cookie has a username associated with it; and
- If the cookie's username is a username in our database of users

If they've passed these two tests, the user is allowed access to the recommended page. If they fail either of the tests, the app redirects the user to the login page. This is important as the recommend_tracks function requires a user object to be parsed. By implementing this security, it ensures that our website cannot be accidentally broken.

By breaking the recommend_tracks function into smaller functions, we follow the single responsibility principle that states, "Every module (or class) should have responsibility over a single part of the functionality provided by the software." This allows the software to be broken down into smaller, more understandable components and ensures that each function does one task and does it well.

```
def recommend_tracks(self, user_object: user):
    self.__high_reviewed_tracks.clear()
    self.__recommended_tracks.clear()
    for user_review in user_object.reviews:
        if user_review.rating > 3:
            self.__high_reviewed_tracks.append(user_review.track)
    for high_reviewed_track in self.__high_reviewed_tracks:
        for track_object in self.__tracks:
            self.recommend_from_artist(high_reviewed_track, track_object)
            self.recommend_from_genre(high_reviewed_track, track_object)
            self.recommend_from_album(high_reviewed_track, track_object)
            self.recommend_from_duration(high_reviewed_track, track_object)
    random.shuffle(self.__recommended_tracks)
    if len(self.__recommended_tracks) > 10:
        self.__recommended_tracks = self.__recommended_tracks[0:10]
    return self.__recommended_tracks

def recommend_from_artist(self, high_reviewed_track, track_object):
    if high_reviewed_track.artist == track_object.artist and high_reviewed_track != track_object and track_object not in self.__recommended_tracks:
        self.__recommended_tracks.append(track_object)

def recommend_from_genre(self, high_reviewed_track, track_object):
    for genre_type in high_reviewed_track.genres:
        if genre_type in track_object.genres and high_reviewed_track != track_object and track_object not in self.__recommended_tracks:
            self.__recommended_tracks.append(track_object)

def recommend_from_album(self, high_reviewed_track, track_object):
    if high_reviewed_track.album == track_object.album and high_reviewed_track != track_object and track_object not in self.__recommended_tracks:
        self.__recommended_tracks.append(track_object)

def recommend_from_duration(self, high_reviewed_track, track_object):
    if high_reviewed_track.track_duration-15 <= track_object.track_duration <= high_reviewed_track.track_duration+15 and high_reviewed_track != track_object and track_object not in self.__recommended_tracks
        self.__recommended_tracks.append(track_object)
```