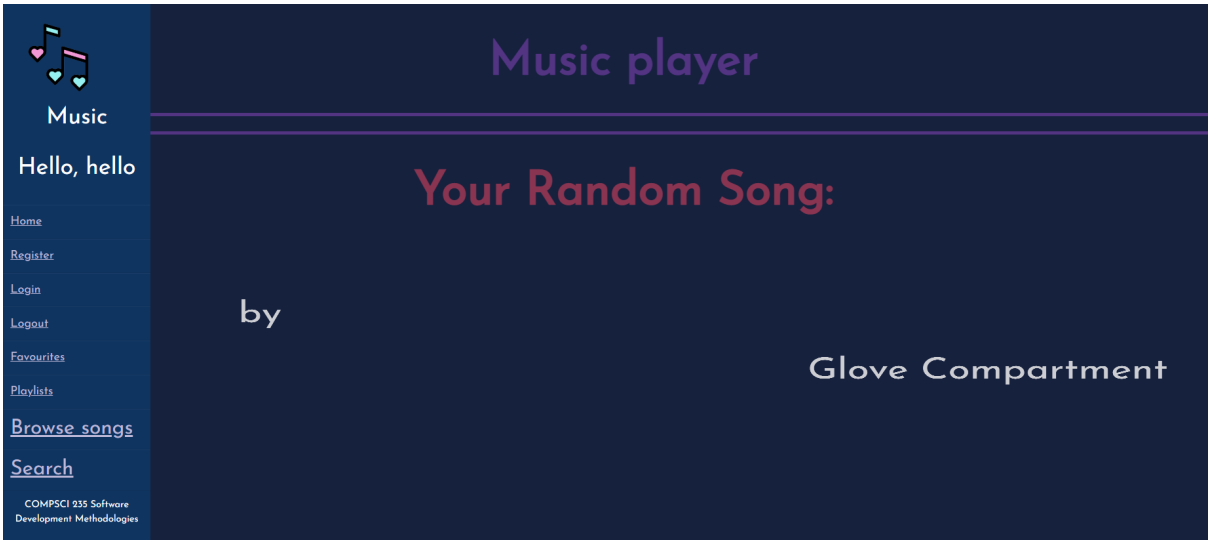
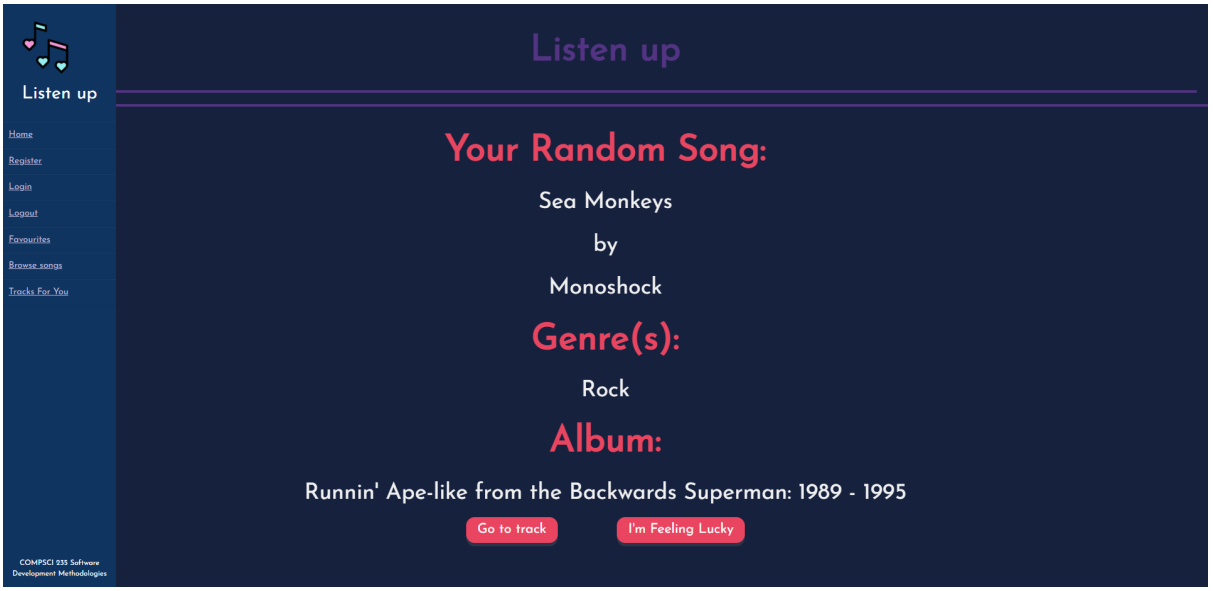
Cool Features Design report

Random suggested song:

Random suggested song when you're on the home page. This feature displays fun animations and you can click on "I'm feeling lucky" to get a new suggestion or click to go to the track to get more details. We decided to implement this to add something fun. If the user wants to discover a new song that could be vastly different from their usual, this is a way for them to try something new.





We created a blueprint for getting random tracks for the home page to comply with the single responsibility principle.

```
@home_blueprint.route('/', methods=['GET'])
def home():
    return render_template('home/home.html',track = utilities.get_random_track())
```

The single responsibility principle makes the software easier to implement and prevents unexpected side-effects of future changes. To follow this principle, the home blueprint doesn't have more than one responsibility, i.e., to render the home HTML template. This avoids any unnecessary, technical coupling between responsibilities and reduces the

probability that I need to change the function. It also lowers the complexity of each change because it reduces the number of dependent classes that are affected by it. There is no need to have multiple classes that all hold just one function. The blueprint calls for the repository pattern from the service layer.

```
def get_random_track():
    random_track = services.get_random_track(repo.repo_instance)
    return random_track
```

```
def get_random_track(repo: AbstractRepository):
    return repo.get_random_track()
```

```
def get_random_track(self):
    return random.choice(self.__tracks)
```

The repository pattern encapsulates the Abstract repository interface and Memory repository. Here, the service layer depends on the Abstract repository interface and the Database repository also depends on the Abstract repository interface. This demonstrates a high-level module and low-level module depending on abstractions, which means the Dependency Inversion Principle is applied.

<u>Genre-based image:</u>

When browsing through tracks, when you click on a track, the displayed image changes depending on the genre. The image is of a gradient of colours that embodies the genre and mood. For example, Rock displays black and red, and Pop displays yellow and pink.

Tracks can be classified using multiple genres, e.g., Pop, Rock, and Folk.

Clicking on a track on the browse page returns a page containing a single track that is associated with genre parameters. To get the genre(s), the track_page blueprint calls track.genres which gets the genres associated with a track from the track class in the domain model.

```
if len(track.genres) >0:
    genre = track.genres[0]
else:
    genre = "No Genre"
```

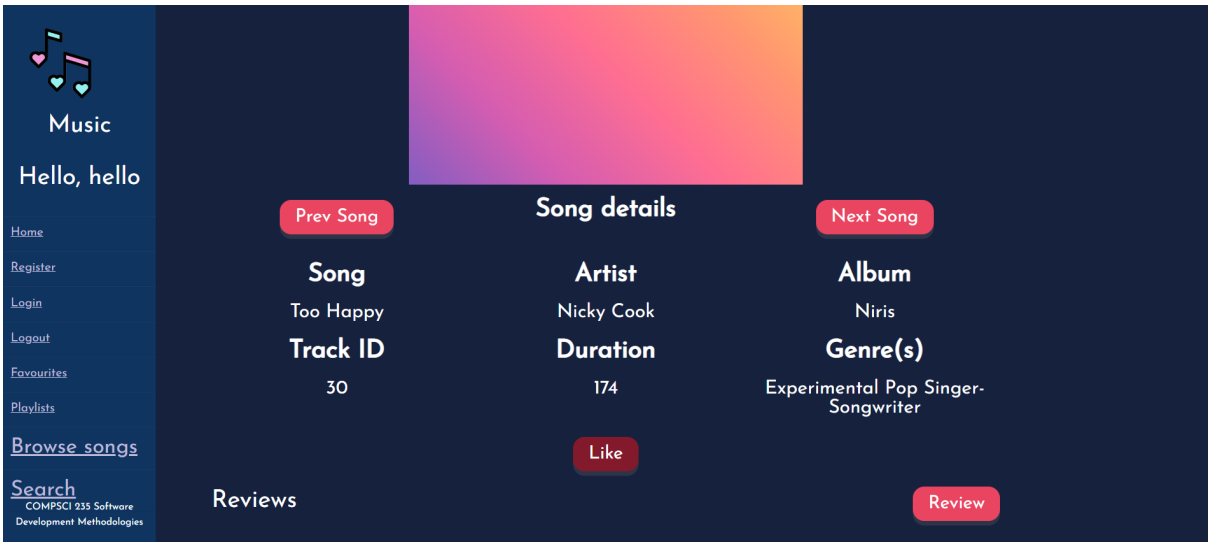
```
self.__genres: list = []
```

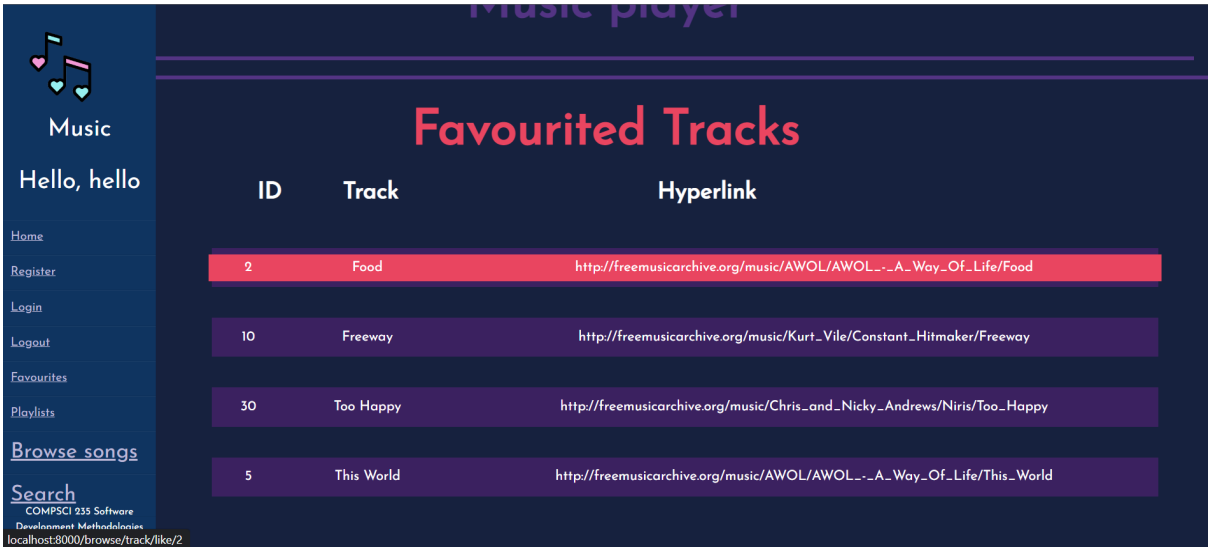
The track class references genre objects from the genre class. The track class is a composition of genre and other objects.

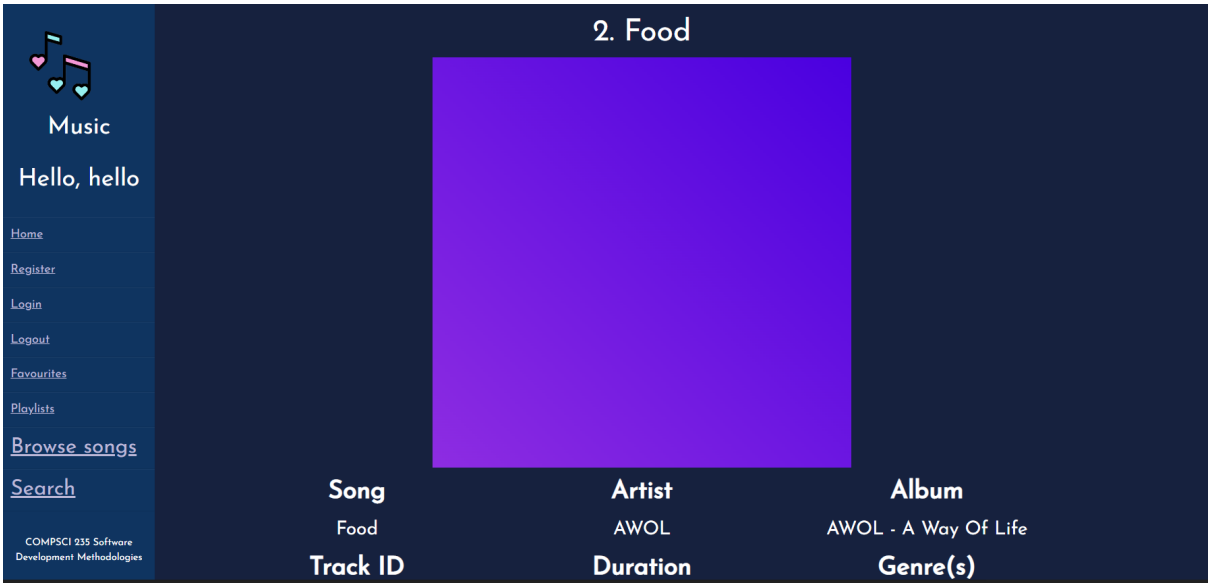
```python
class Genre:

    def __init__(self, genre_id: int, genre_name: str):
        if type(genre_id) is not int or genre_id < 0:
            raise ValueError('Genre ID should be an integer!')
        self.__genre_id = genre_id

        if type(genre_name) is str:
            self.__name = genre_name.strip()
        else:
            self.__name = None
```

Liked songs:

Logged-in users can browse tracks and click a like button to like songs they enjoy and save them for later. These liked songs will then be visible in their “Favourites” tab. Users can browse these favourited tracks and click on them for more details. The favourited track includes a hyperlink users can copy and share with their friends. Users can also unlike songs to remove the track from their favourites.







This feature uses the like_track, unlike_track, and track_page blueprints.

```python
@track_blueprint.route('/track/<int:track_id>', methods=['GET'])
def track_page(track_id):
    prev_track, track, next_track = utilities.get_selected_track(track_id)
    if track:
        track_to_show_reviews = request.args.get('view_reviews_for')
        if track_to_show_reviews == None:
            # No view-reviews query parameter, so set to a non-existent track id.
            track_to_show_reviews = -1
        elif track_to_show_reviews is not None:
            # Convert track_to_show_reviews from string to int.
            track_to_show_reviews = int(track_to_show_reviews)
        if track_to_show_reviews is None :
            track_to_show_reviews = -1
        else:
            track_to_show_reviews = int(track_to_show_reviews)
        view_review_url = url_for('track_blueprint.track_page',track_id = track_id,  view_reviews_f
        add_review_url = url_for('track_blueprint.review_on_track', track_id = track_id)

        next_track_url = url_for('track_blueprint.track_page',track_id  = next_track.track_id) if r
        prev_track_url = url_for('track_blueprint.track_page', track_id = prev_track.track_id) if p

        like_track_url = url_for('track_blueprint.like_track', track_id = track_id)
```

```python
@track_blueprint.route('/track/like', methods=['GET', 'POST'])
@login_required
def like_track():
    user_name = session['user_name']
    user: User = get_user(user_name, repo.repo_instance)
    if request.args.get('track_id') == None:
        track_id = 2
    else:
        track_id = int(request.args.get('track_id'))

    prev, track, next = services.get_track(track_id, repo.repo_instance)
    services.add_liked_track(track, user_name, repo.repo_instance)
    tracks = services.get_liked_tracks(user_name, repo.repo_instance)

    return render_template('profile/favourites.html',
    title='Liked Tracks', track = track, tracks = tracks, user = user
    )
```

```python
@track_blueprint.route('/track/unlike', methods=['GET', 'POST'])
@login_required
def unlike_track():
    user_name = session['user_name']
    user: User = get_user(user_name, repo.repo_instance)
    if request.args.get('track_id') == None:
        track_id = 2
    else:
        track_id = int(request.args.get('track_id'))

    prev, track, next = services.get_track(track_id, repo.repo_instance)
    services.remove_liked_track(track, user_name, repo.repo_instance)
    tracks = services.get_liked_tracks(user_name, repo.repo_instance)

    return render_template('profile/favourites.html',
    title='Liked Tracks', track = track, tracks = tracks, user = user
    )
```

These blueprints follow the single responsibility principle. Blueprints organise the application into distinct components. The like_track blueprint is responsible for handling requests concerning liking tracks and the unlike_track blueprint is responsible for handling requests concerning removing liked tracks, while track_page blueprint is responsible for creating HTML pages. These request handlers are responsible for one component each and call the service layer (services) which is responsible for fetching/updating/processing data from the repo (complies with the repository pattern). The functions in the repo all have one responsibility or purpose in the program, thus complying with the single responsibility principle. The repository pattern is a simplifying abstraction over data storage that allows us to break down the domain model from the database. The tracks are displayed on the

browser using templates that render HTML documents used to display in the user's browser. The template extends layout which is a use of template inheritance.

Personalised tracks:

As the title implies, we have the ability for users to get song recommendations based on the songs they have added to their favourites. For this feature we used the profile blueprint. This blueprint handles rendering the favourite tracks of the user and recommending the user songs based on songs that were liked by said user. By doing this, we adhere to the single responsibility principle as we have created and used the profile blueprint only for this purpose.

Here is a snippet of the blueprint:

```
if liked_tracks:
    recommended_tracks, random_track= services.get_recommended_tracks(user_name, repo.repo_instance)
    stripped_recommended_tracks  = []
    [stripped_recommended_tracks.append(track) for track in recommended_tracks if track not in stripped_recommended_tracks]
    # print(request.args.get('track_id'))
    if request.args.get('track_id') and  int(request.args.get('track_id')) in [track.track_id for track in liked_tracks]:
        random_track = services.get_track(int(request.args.get('track_id')), repo.repo_instance)[1]
```

This screenshot shows the blueprint calling the service layer from our blueprint.

```
def get_recommended_tracks(user_name, repo: AbstractRepository):
    user : User = repo.get_user(user_name)
    liked_tracks = get_liked_tracks_list(user_name, repo)
    if liked_tracks:
        random_track = liked_tracks[0]

        if random_track.genres:
            recommended_tracks = repo.get_filtered_tracks(random_track.album.title, "album")
            recommended_tracks += repo.get_filtered_tracks(random_track.genres[0].name, "genre")
            recommended_tracks += repo.get_filtered_tracks(random_track.artist.full_name, "artist")
            return recommended_tracks, random_track
```

```
def get_liked_tracks_list(user_name, repo: AbstractRepository):
    user : User = repo.get_user(user_name)
    if user is None:
        raise UnknownUserException
    tracks = user.liked_tracks
    if tracks is None:
        raise NonExistentTrackException
    return tracks
```

This screenshot shows the service layer calling the abstract repository, other methods in the service layer and the domain model.

```
@abc.abstractmethod
def get_track(self, id:int):
    raise NotImplementedError
```

```
@abc.abstractmethod
def get_user(self, user_name):
    raise NotImplementedError
```

```python
def get_track(self, id:int) -> Track:
    track = None
    prev_track = None
    next_track =None

    try:
        track = self.__track_index[id]
        track_index = self.tracks.index(track)
        prev_track = self.tracks[track_index-1] if track_index - 1 >= 0 else None
        next_track = self.tracks[track_index + 1] if track_index + 1 < len(self.__tracks) else None
    except KeyError:

        pass
    return prev_track, track, next_track
```

These screenshots show how the method we call is abstract and how the memory repository focuses on the databases, infrastructure and processing data. These screenshots show how our design patterns coincide with the dependency Inversion principle as we have used. As our service layer and memory repository depend on the abstract repository interface, ‘repository’, and we are calling its abstract methods to get liked tracks and users. The recommendation work as follows:
The service layer uses the domain model and abstract repository interface to process requests and returns it to the blueprint.
The recommended songs are based on what the user has liked, it choses a liked song and gets all tracks by the same artist, on the same album and of the same genres and renders them.