Development prep: tech stack/CI:

- ### Develop MVP/devops- for your own feature:

First, we can create a new Angular component called LikedSongsComponent that will be responsible for displaying the user's liked songs.

Next, we can create a service called LikedSongsService that will handle all the API calls related to the user's liked songs. The service will have a method called getLikedSongs() that will make a GET request to the backend API to fetch the user's liked songs.

We can then inject the LikedSongsService into the LikedSongsComponent and use the getLikedSongs() method to retrieve the user's liked songs and display them on the page.

For example, the code for the LikedSongsComponent might look like this:

```
import { Component, OnInit } from '@angular/core';
import { LikedSongsService } from '../services/liked-songs.service';


@Component({
  selector: 'app-liked-songs',
  templateUrl: './liked-songs.component.html',
  styleUrls: ['./liked-songs.component.css']
})
export class LikedSongsComponent implements OnInit {
  likedSongs: any[];


  constructor(private likedSongsService: LikedSongsService) { }


  ngOnInit() {
    this.getLikedSongs();
  }


  getLikedSongs() {
    this.likedSongsService.getLikedSongs().subscribe(songs => {
      this.likedSongs = songs;
    });
  }
}
```

And the LikedSongsService might look like this:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class LikedSongsService {
  private apiUrl = 'http://localhost:8080/api/liked-songs';

  constructor(private http: HttpClient) { }

  getLikedSongs() {
    return this.http.get<any[]>(this.apiUrl);
  }
}
```

Of course, this is just a rough example to illustrate how we can develop an MVP for the Liked Songs feature. The actual implementation would likely involve more features and functionality.

• Integrate any libraries into stack:

Library:

- Angular Material for UI components.

Integration Steps:

- Install the Angular Material library using **npm**.
- Import the required Angular Material components and modules into the Angular app.

- Use Angular Material components in the Liked Songs feature, such as **mat-list** to display the list of liked songs and **mat-icon-button** to add/remove a song.

• Integrate API tokens/dummy calls:

1. Obtain API credentials: To integrate the Spotify API, we need to obtain API credentials from the Spotify developer dashboard. This will typically involve registering an application and obtaining a client ID and secret key.
2. Create an API service: In Angular, we can create an API service that will handle all API calls for the Liked Songs feature. We can use RxJS to handle the HTTP requests and responses, and we can inject the HttpClient service into our API service to make requests to the Spotify API.
3. Create a dummy API: To test the integration of the Spotify API, we can create a dummy API that returns sample data. This can be useful to ensure that our code is functioning correctly and that we are able to retrieve the data that we need from the Spotify API.
4. Replace dummy calls with actual API calls: Once we have successfully integrated the Spotify API and tested our code with dummy calls, we can replace the dummy calls with actual API calls. We can use the API service that we created earlier to make requests to the Spotify API and retrieve data about the user's liked songs.
5. Handle authentication and authorization: To ensure that our API calls are authenticated and authorized, we need to handle authentication and authorization in our code. This may involve sending the user's access token with each API request or using OAuth 2.0 to obtain an access token on behalf of the user. We can use libraries like ng2-cookies or ngx-cookie-service to store and manage user authentication tokens.

• Propose data model, constraints, architecture:

Data model:

The data model for the Liked Songs feature will need to store information about the songs that the user has liked. This information may include the song title, artist, album, duration, genre, and any other relevant metadata. Additionally, we may want to store information about the playlists that the user has created and the songs that are included in each playlist.

To represent this data, we can use a relational database such as PostgreSQL. We can define two main entities: **Song** and **Playlist**. The **Song** entity can have attributes such as **id**, **title**, **artist**, **album**, **duration**, and **genre**. The **Playlist** entity can have attributes such as **id**, **name**, and **description**. To represent the many-to-many relationship between songs and playlists, we can create a third entity called **PlaylistSong**, which will have foreign keys to both the **Song** and **Playlist** entities.

Constraints:

To ensure data consistency and integrity, we can define some constraints for our data model. For example:

- The **id** attribute for each entity should be unique.
- The **title** attribute for each **Song** entity should not be null.
- The **duration** attribute for each **Song** entity should be greater than 0.
- Each **Playlist** entity should have at least one **Song** entity associated with it.
- The same **Song** entity should not be associated with a **Playlist** more than once.

Architecture:

To implement the Liked Songs feature, we can use a combination of JHipster, Angular, Spring Boot, and PostgreSQL. The architecture can be divided into three main components:

1. Frontend: The frontend can be developed using Angular and can include features such as displaying the list of liked songs, playing a snippet from a song, removing songs from the list, and adding to a playlist.
2. Backend: The backend can be developed using Spring Boot and can include features such as retrieving song data from the Spotify API, storing song and playlist data in the PostgreSQL database, and providing endpoints for the frontend to interact with the data.
3. Database: The database can be developed using PostgreSQL and can include the entities and constraints defined above.

The frontend and backend can communicate with each other using RESTful API calls, with the frontend sending requests to the backend and the backend sending responses back to the frontend. To ensure a smooth and efficient communication between the frontend and backend, we can define a clear API contract that specifies the expected request and response formats, HTTP methods, and endpoints. This can be achieved using tools such as Swagger or OpenAPI. Additionally, we can implement security measures such as JWT authentication to protect the API endpoints from unauthorized access.