## Design Document

Contents

[Design Document 1](#_Toc120310321)

[C1 Diagram 2](#_Toc120310322)

[C2 Diagram 3](#_Toc120310323)

[C3 Diagram 4](#_Toc120310324)

[C4 Diagram 5](#_Toc120310325)

[How is SOLID guaranteed? 6](#_Toc120310326)

[API Endpoints 7](#_Toc120310327)

[CI Setup diagram 8](#_Toc120310328)

# C1 Diagram

In the C1 component of the C4 architecture, the customers should have different roles that serve different purposes! DJs will be able to upload songs and playlists and listeners will be able to listen to them

# C2 Diagram

The application has a front end and back end. The front end interacts with an external API from SoundCloud where users can connect their accounts and the backend interacts with the database

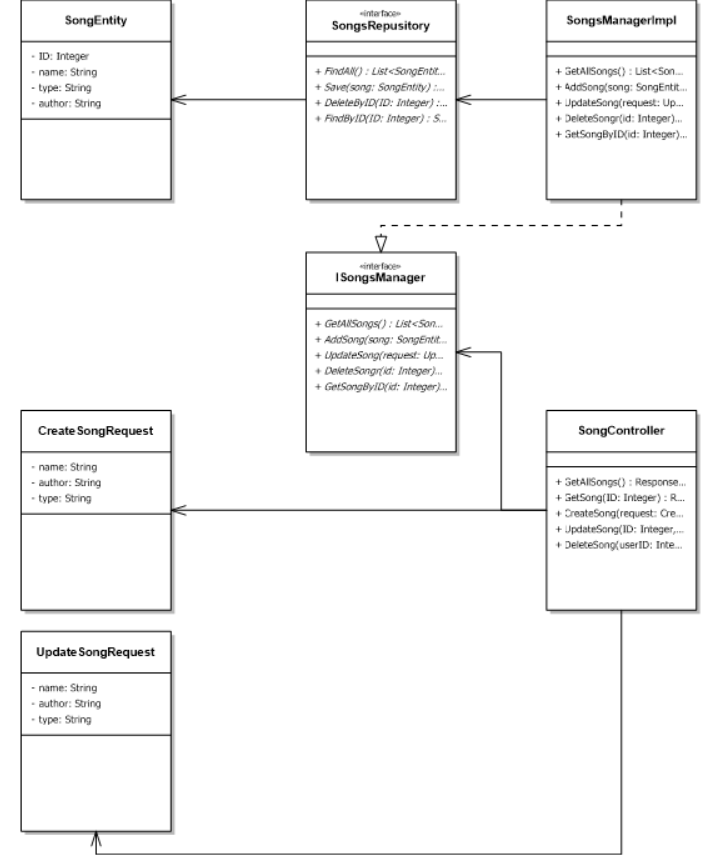
# C3 Diagram

The backend consists of:  
1) A REST controllers which takes care of the API operations

2) Manager classes (SongsManager) that will provide the functionality and interact with the repository

3) JPA repository(SongsRepo) that interacts with the database

# C4 Diagram

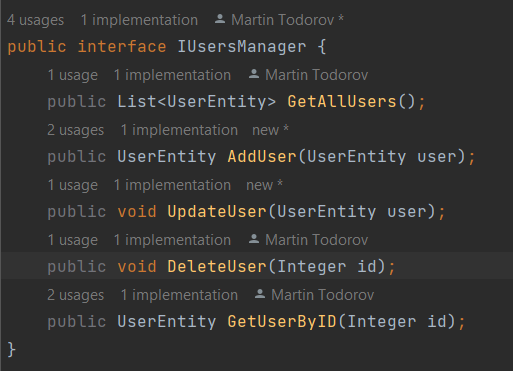


# How is SOLID guaranteed?

Solid is a combination of 5 principles that guarantee the quality of the code. After using SOLID the code should be readable and extendable in case another software engineer picks up on it!

1. Single responsibility principle

-One class should serve only one purpose. This does not imply that each class should have only one method, but they should all relate directly to the responsibility of the class. All the methods and properties should work towards the same goal. When a class serves multiple purposes or responsibilities, it should be made into a new class.

Example code snippet:  
  
My user service class only manages users and nothing else  


1. Open closed principle

-Software entities (classes, modules, functions, etc.) should be extendable without actually changing the contents of the class you’re extending. If we could follow this principle strongly enough, it is possible to then modify the behavior of our code without ever touching a piece of the original code

1. Liskov substitution

-any implementation of an abstraction (interface) should be substitutable in any place that the abstraction is accepted. Basically, it takes care that while coding using interfaces in our code, we not only have a contract of input that the interface receives, but also the output returned by different classes implementing that interface; they should be of the same type

1. Interface segregation

-we should break our interfaces into many smaller ones, so they better satisfy the exact needs of our clients.

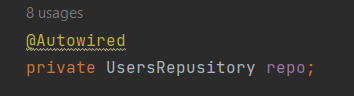
Similar to the Single Responsibility Principle, the goal of the Interface Segregation Principle is to minimize side consequences and repetition by dividing the software into multiple, independent parts

Example: I am only using interfaces for one purpose without repetition

1. Dependency inversion

-By applying the Dependency Inversion Principle, the modules can be easily changed by other modules just changing the dependency module. Any changes to the low-level module won’t affect the high-level module

Example code snippet:



# API Endpoints

|  |  |  |  |
| --- | --- | --- | --- |
| URL | Resource | Operation |  |
| /users | Users | GET | Read a list of all users |
| /users | Users | POST | Create a new user |
| /users/1 | Users | GET | Read the user with ID 1 |
| /users/1 | Users | PUT | Update the user with ID 1 |
| /users/1 | Users | DELETE | Delete the user with ID 1 |
| /songs | Songs | GET | Read a list of all songs |
| /songs | Songs | POST | Create a new song |
| /songs/1 | Songs | GET | Read the song with ID 1 |
| /songs/1 | Songs | PUT | Update the song with ID 1 |
| /songs/1 | Songs | DELETE | Delete the user with ID 1 |
| /playlists | Playlists | GET | Read a list of all playlists |
| /playlists | Playlists | POST | Create a new playlist |
| /playlists/1 | Playlists | GET | Read the playlist with value 1 |

# CI Setup diagram