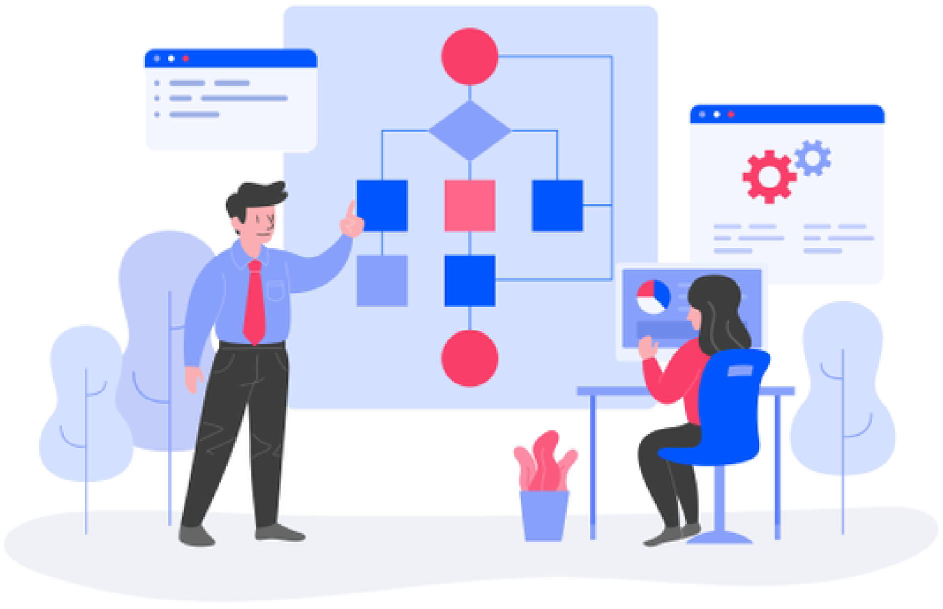
Software Architecture Document

Mariela Gocheva

30 September 2022



Version History

Version	**Changelog**	**Date**	**Rationale**
v1.0	• -	November 4, 2022	Issued

Contents

1 Introduction

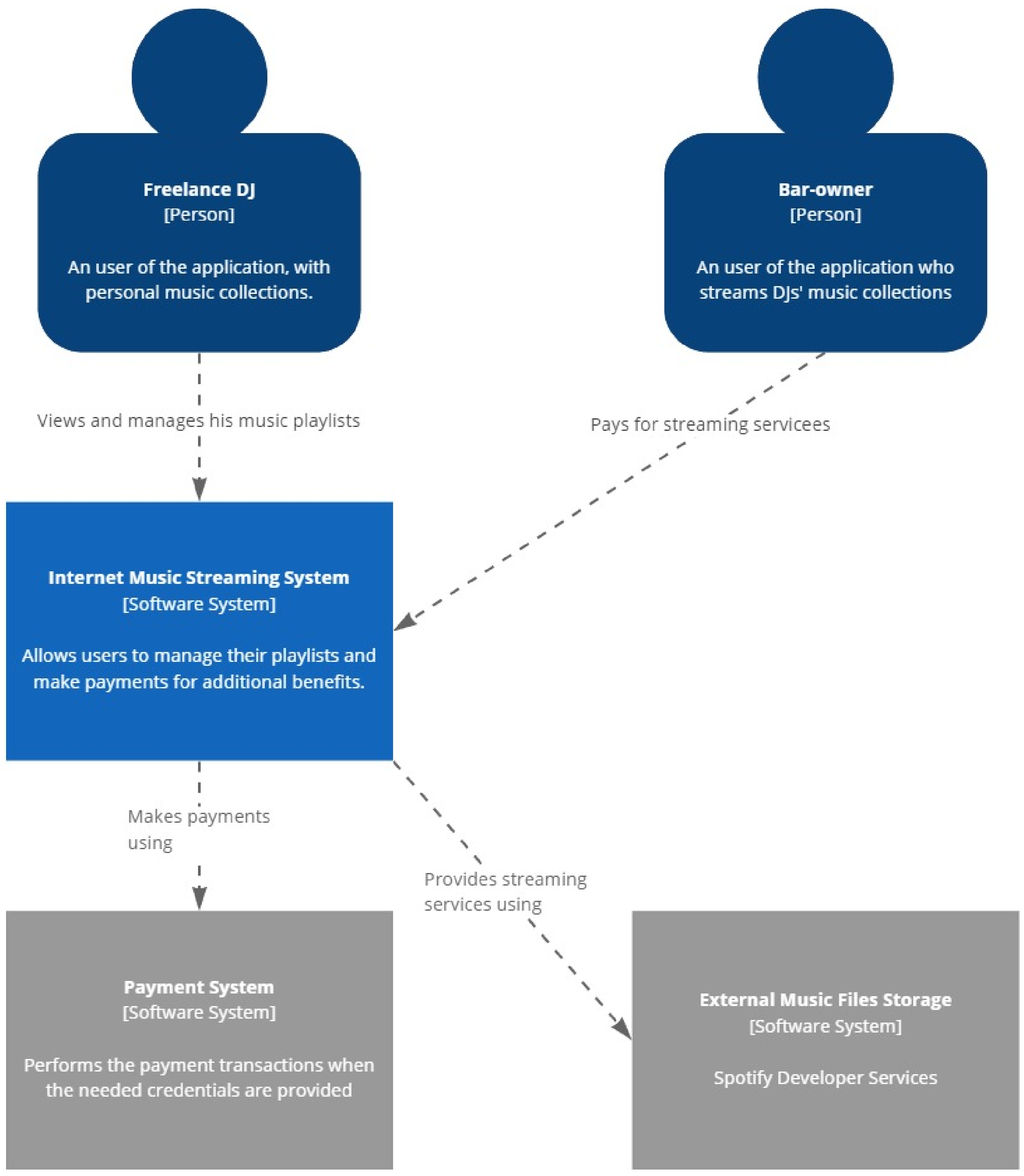
The purpose of this document is to provide a detailed architecture design of the new music collection streaming platform DJingle by focusing on four key concepts: system context, containers and tech choices, components and diagrams of the outlook of the software. These attributes were chosen based on their importance in the design and construction of the application.

This document includes a general description of the system, a scope definition, the logical architecture of the software, the layers and top-level components as well as justification of the technical choices made. Each of the aforementioned attributes will be described through a comprehensive sections followed by an architectural overview, which includes different types of diagrams and a full description of patterns and tactics that will be used to address the architecturally significant processes. Finally, a conclusion and a summary of the technical choices will round out the document.

The intention of this document is to help for the development process of determining how the system will be structured at the highest level. Moreover, the project leader and the rest of the stakeholders can use this document to validate that the developer is meeting the agreed-upon requirements during the evaluation of the developer's efforts.

2 System Context

2.1 Diagram C1


Freelance DJ
[Person]
An user of the application, with personal music collections.
Bar-owner
[Person]
An user of the application who streams DJs' music collections
Views and manages his music playlists
Pays for streaming servicees
Internet Music Streaming System
[Software System]
Allows users to manage their playlists and make payments for additional benefits.
Makes payments using
Provides streaming services using
Payment System
[Software System]
Performs the payment transactions when the needed credentials are provided
External Music Files Storage
[Software System]
Spotify Developer Services

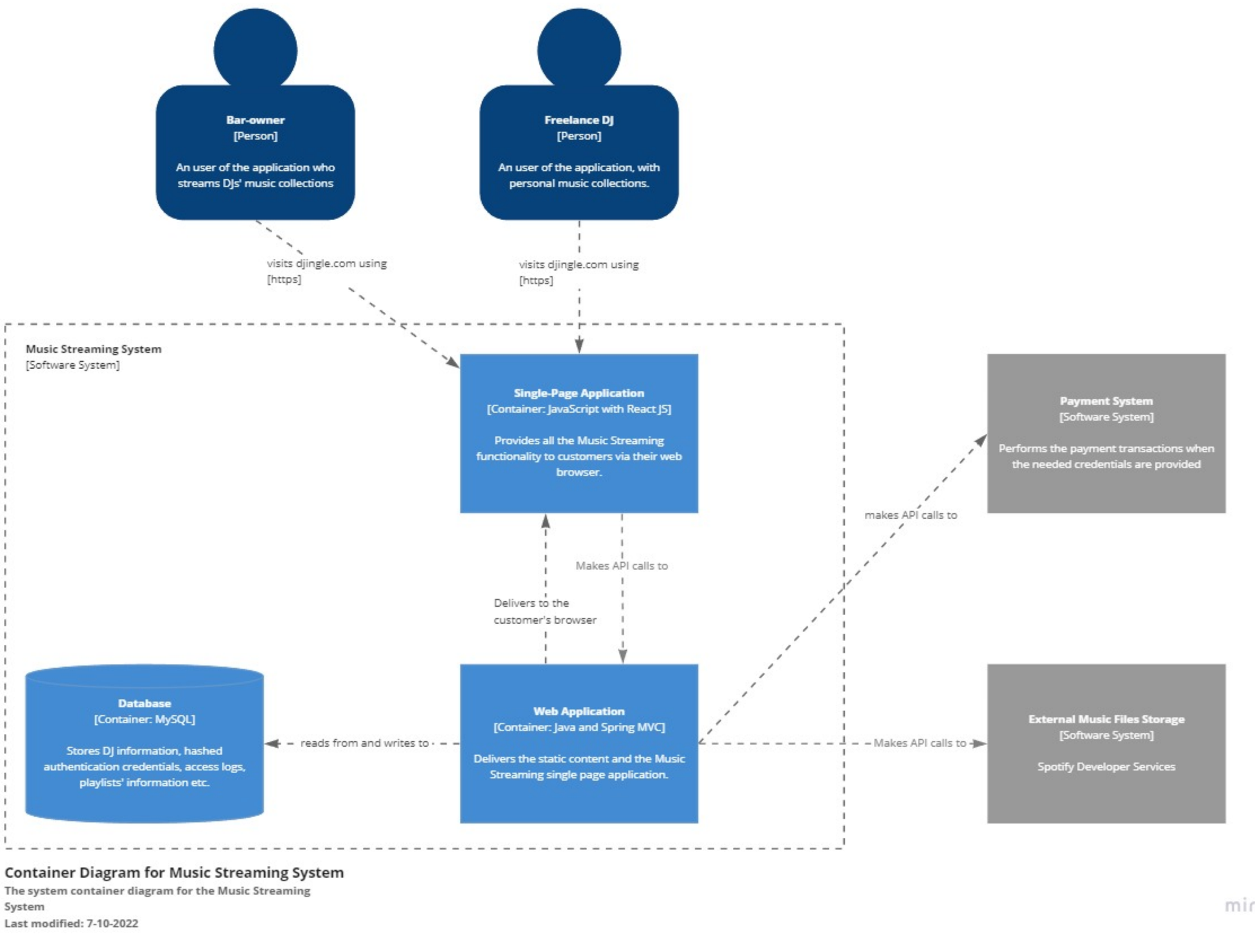
System Context Diagram for Music Streaming System
The system context diagram for the Music Streaming System
Last modified: 26-09-2022

2.2 Technical Choices

******************************* External systems justification *******************************

3 Containers and technical choices

3.1 Diagram C2

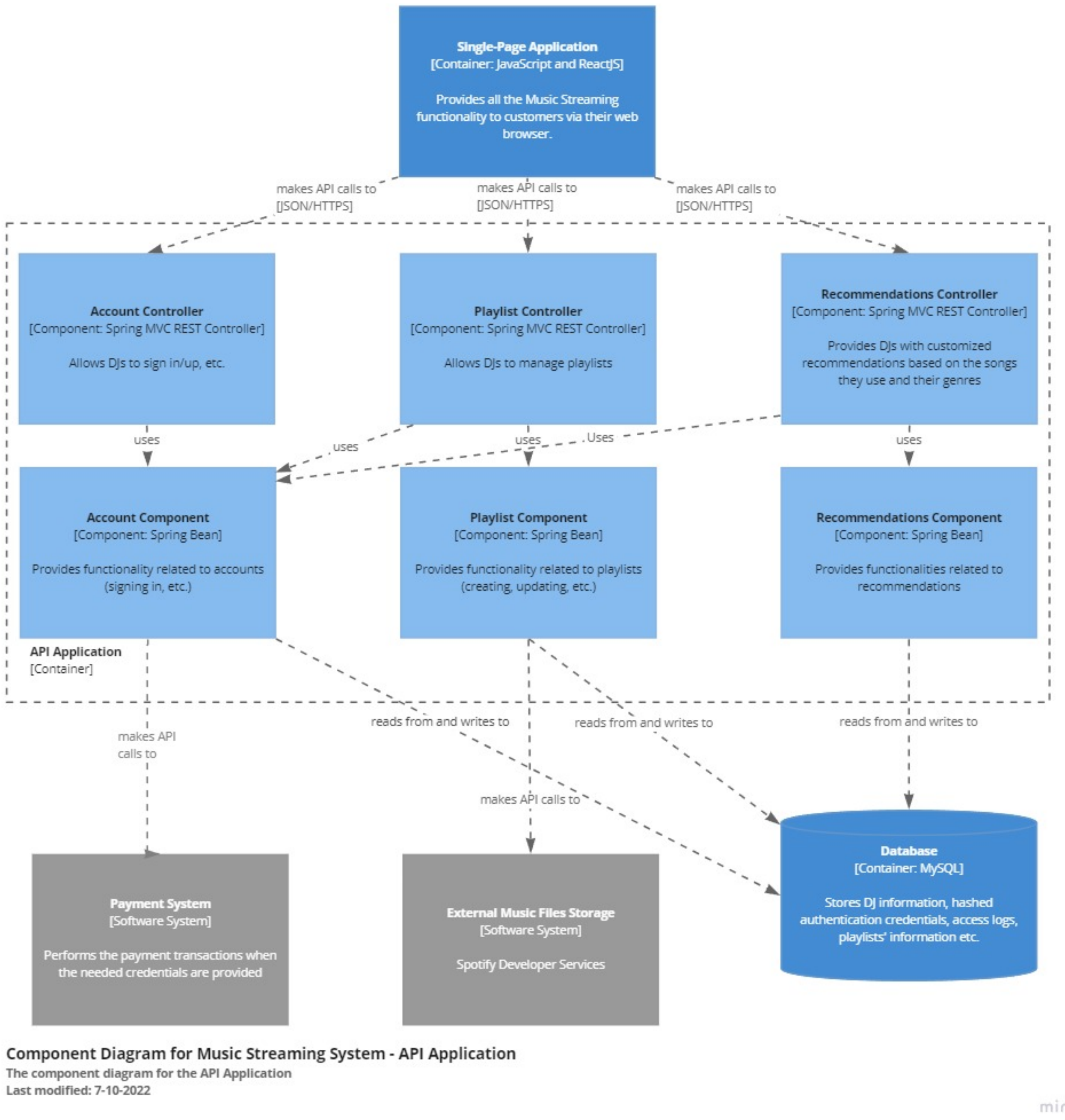

Bar-owner
[Person]
An user of the application who streams DJs' music collections
Freelance DJ
[Person]
An user of the application, with personal music collections.
visits djingle.com using [https]
visits djingle.com using [https]
Music Streaming System
[Software System]
Single-Page Application
[Container: JavaScript with React JS]
Provides all the Music Streaming functionality to customers via their web browser.
Makes API calls to
Delivers to the customer's browser
makes API calls to
Payment System
[Software System]
Performs the payment transactions when the needed credentials are provided
Database
[Container: MySQL]
Stores DJ information, hashed authentication credentials, access logs, playlists' information etc.
reads from and writes to
Web Application
[Container: Java and Spring MVC]
Delivers the static content and the Music Streaming single page application.
Makes API calls to
External Music Files Storage
[Software System]
Spotify Developer Services
Container Diagram for Music Streaming System
The system container diagram for the Music Streaming System
Last modified: 7-10-2022

3.2 Technical Choices

******************************* Java, JavaScript and React Justification, MySql justification *******************************

4 Components

4.1 Diagram C3


Single-Page Application
[Container: JavaScript and ReactJS]
Provides all the Music Streaming functionality to customers via their web browser.
makes API calls to [JSON/HTTPS]
makes API calls to [JSON/HTTPS]
makes API calls to [JSON/HTTPS]
Account Controller
[Component: Spring MVC REST Controller]
Allows DJs to sign in/up, etc.
Playlist Controller
[Component: Spring MVC REST Controller]
Allows DJs to manage playlists
Recommendations Controller
[Component: Spring MVC REST Controller]
Provides DJs with customized recommendations based on the songs they use and their genres
uses
uses
uses
Uses
uses
Account Component
[Component: Spring Bean]
Provides functionality related to accounts (signing in, etc.)
Playlist Component
[Component: Spring Bean]
Provides functionality related to playlists (creating, updating, etc.)
Recommendations Component
[Component: Spring Bean]
Provides functionalities related to recommendations
API Application
[Container]
makes API calls to
reads from and writes to
reads from and writes to
reads from and writes to
makes API calls to
Database
[Container: MySQL]
Stores DJ information, hashed authentication credentials, access logs, playlists' information etc.
Payment System
[Software System]
Performs the payment transactions when the needed credentials are provided
External Music Files Storage
[Software System]
Spotify Developer Services
Component Diagram for Music Streaming System - API Application
The component diagram for the API Application
Last modified: 7-10-2022

4.2 Technical Choices

******************************* Springboot *******************************

5 How the SOLID principles are guaranteed?

In the matter of creating quality and maintainable code, proper implementation of the SOLID principles is mandatory. This approach assists the developers as it provides clear and unambiguous guidelines for creating cogent and well-organized solutions with guaranteed maintainable structure.

To ensure the implementation of the Single Responsibility Principle I decided to use feature-driven development. This technique implies the separation of each feature into one particular class. Using this strategy, I can guarantee that each class will have only one main responsibility and a single reason to change.

Regarding the Open Closed Principle, I am going to guarantee the possibility to extend classes without modifying them with creating the classes in such manner that is going to be amenable to an insert of interfaces. Moreover, when a certain behaviour needs to be reused by different classes across the system, it will not be possible to reuse its functionalities for other types of repositories if it's included in a repository class. In that case, that behaviour can be separated in a different component to be easily processed again. Additionally, in some occasions it should be possible to enable/disable a new behaviour in accordance with some user settings or other conditions which can be accomplished with separating the different behaviour in several classes which will made the code easier to test and maintain.

The Liskov Substitution Principle is going to be enforced in my solution with the creation of superclasses which will have similar behaviour as their subclasses. This approach will ensure that a superclass's object can be replaced by a subclass's one. The correct implementation of the Liskov Substitution Principle can also be guaranteed during the testing state of the development as I can execute some parts of the code and provide only subclasses' objects to check whether they're going to meet the correct performance.

In relation to the Interface Segregation Principle, I am going to be careful with situations where creating an interface needs to be implemented in more than one class. I will ensure that in case of some of its operations are not being used in all of the implementation classes and cause empty method implementations, the interface will be split in several ones so they can be independent and suitable for its relying classes. These interfaces will be client-specific and will reduce the need of required changes in the system.

The Dependency Inversion Principle concerns commonly discussed and studied topics as its incorrect implementation leads to detrimental consequences over the code structure. Usually, if the Liskov Substitution Principle is properly enforced, the Dependency Inversion will also be followed. Nevertheless, to guarantee its enforcement I'm going to introduce interface abstractions between lower-level and higher-level software components to remove the dependencies between them without affecting any other classes.

6 Feature-driven approach justification

When it comes to software development and code implementation good organization, as well as clarification about all of the methodologies and mechanisms put in use, have vital importance regarding the quality of the final product. Therefore, inevitably I faced the junction of choices regarding the major decision whether I'm going to use the feature approach or I'm going to use the service/management strategy.

Thus, I decided to go for the feature-driven development as it provides easily distinguishable classes which have a focus on a specific purpose while at the same time this approach also redounds to achieving clearer and well-organized code. Another benefit of the feature-based implementation is that in the beginning state of the development the minimal viable product is easier to get reached as it may require only a few functioning features. Moreover, feature-driven development prevents one crucial problem which is commonly faced while using the service/management approach – it eliminates the possibility of forming 'GOD' classes which concentrate a lot of responsibilities, control and oversee many different objects, and effectively do everything in the application. That issue results into producing unmaintainable code and disastrous scattered software architecture.

On a final note, another benefit of the feature-driven development which made me choose it as an implementation strategy for my individual project is that in my opinion it provides the opportunity of guaranteeing more easily that the code is going to have a complete test coverage due to its fined structure which allows the testing of each possible case to be performed per feature.

7 Persistence per component

8 Interfaces or API Documentation