**BABEȘ-BOLYAI UNIVERSITY CLUJ-NAPOCA  
FACULTY OF MATHEMATICS AND COMPUTER** **SCIENCE**

**SPECIALIZATION MATHEMATICS  
AND COMPUTER SCIENCE**

**DIPLOMA THESIS  
Audio Streaming in React and  
.NET Core**

**Supervisor  
Dr. Conf. STERCA Adrian**

**Author  
CUCIUREANU Vlad**

**2023**

**UNIVERSITATEA BABEȘ-BOLYAI CLUJ-NAPOCA  
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ**

**SPECIALIZAREA MATEMATICĂ-INFORMATICĂ**

**LUCRARE DE LICENȚĂ  
Audio Streaming in React and  
.NET Core**

**Conducător științific  
Dr. Conf. STERCA Adrian**

**Absolvent  
CUCIUREANU Vlad**

**2023**

**Abstract**

The paper aims to demonstrate how an audio streaming tool can be built using mainstream, state-of-the-art technologies like React and .NET Core. While these two tools were chosen for this paper’s implementation, there are many popular tools which are able to deliver such functionalities, with vast communities of passionate users willing to provide any required support.

The first chapter proposes one of the many options for achieving the desired goal, while presenting pros and cons for other solutions.

The second chapter is split into three sections which present the application in theory, the decisions taken and technologies used to develop this thesis’ tool. The first section lays out the client-server architecture used. The second section presents the backend, offering a technical description of the project structure, the way the blobs are being handled and served and of the architectural decisions taken. The last section goes in depth into the tools used to build the web application and into how unopinionated tools like React can be leveraged.

The third chapter showcases the example implementation of the application and provides a step-by-step description on how it was built. This includes a presentation of the design system used.

The last chapter is the author’s conclusion regarding this paper’s proposal towards solving the presented issue using the proposed tools. That includes a reflection on the developer’s experience working with said tools.

**Contents**

1. **Introduction**
   1. The problem
   2. Motivation
   3. Other solutions
   4. Proposed Solution
2. **Used technologies**
   1. Client-Server Architecture
      1. Overview
      2. Communication channel
      3. Implementation details
   2. Back-End
      1. API
      2. Data & Storage
   3. Front-End
      1. Technologies & Frameworks
      2. React
      3. State Management
      4. API Interaction (Fetch)
      5. UI / UX
      6. Other notable tools
3. **Practical application**
   1. Design
      1. Wireframing
   2. Implementation
      1. Scaffolding
      2. API Interraction
      3. State management
4. **Conclusion**

**Chapter 1**

**Introduction**

* 1. **The problem**

Nowadays, if one desires to consume audio content, most people would point towards audio streaming platforms such as Spotify, Apple Music, SoundCloud, Deezer or newer, lesser known options like Tidal or Amazon Music. And why wouldn’t they?  
 All these platforms offer value that surpasses the cost of their service. Endless music catalogs accompanied by a plethora of podcasts hooked up to an algorithm built for recommending content that the user might like based on previous listening sessions.But once one might be interested in something more or something different than all these platform’s offerings, a lack of flexibility and a lack of freedom arises.  
 The problem becomes clearer when the user desires something more than the platforms’ main offerings. Then, a sense of lack of flexibility, privacy, freedom and customizability arises.

**1.2 Motivation**

The aforementioned platforms have a habit of collecting large amounts of user data, including listening habits, personal preferences and demographic information. For some, that is a deal-breaker, as reducing one’s digital footprint, as a habit, is gaining traction among growing communities of privacy-focused enthusiasts, including myself.  
 Another concern is the monopoly that rose with mainstream audio streaming platforms. Such control over the market makes available options limited, with the user being given limited control over the content they are able to benefit from. Some users want the freedom to curate and manage their own music libraries that might include rare or independent music that might not be available on popular streaming platforms.  
 For some, other aspects of the services are dissatisfying. Some users complain about frequent ads interrupting music playback, while others complain about limited audio quality options, lack of customization and user interfaces that don’t cater to specific preferences or needs. In such situations one does not have many options.

**1.3 Other solutions**

For a few years now, most (to not say all) households owning a computer became the norm, with personal computing power becoming more and more accessible to people with all kinds of budgets. From such a context, an interesting preposition was born: “What if you could self-host all these platforms that we use and pay for monthly?”.  
 While such an idea would seem out of reach for the average human, such ideas became more and more popular, as seen in the growth of large online communities such as r/selfhosted and r/homelab. What once was a laborous task, configuring such services became more and more accessible and straight-forwards, with self-hosted services becoming almost plug-and-play via tools like Docker and more.

Music Player Daemon (known under it’s acronym, MPD) is one of such self-hostable services. MPD was released in 2003 by Max Kellerman and has since become a go-to solution for a self-hostable audio streaming service. It is powered by FFmpeg and therefore offers compatibility with all mainstream audio file formats. It also offers almost any option anyone could feasably desire to enable in an audio player. The issue is one of it’s strengths, actually. MPD touts itself as being GUI-agnostic and GUI-independent. While this opens a plethora of options for the user interface, it resulted in a fractured set of front-ends that simply don’t add up to the user experience Spotify offers.

Another note-worthy service would be Jellyfin. While Jellyfin is in fact a multimedia system, it functions as a music streaming service too. Taking a stance opposite to MPD’s, which only offers one thing and does it well, Jellyfin tackles all issues all at once and offers a more kitchen-sink solution. The issue with Jellyfin, though, is that it’s main inspiration is Plex, a non-free private multimedia streaming platform and therefore also underdelivers on the audio side.

**1.4 Proposed solution**

The proposed solution for the problem presented in section 1.1 is a web application, which focuses on deliving a similar enough experience to a mainstream audio streaming platform, such as Spotify. The reasoning behind it being that web standards are stable nowadays, that web applications are mostly platform-agnostic, therefore opening doors to quick cross-platform feature delivery, and that offering such an application in an open-source medium could open doors for further extension and personalization to the user’s desire.

In other words, this proposed sollution provides the following features:

* Authentication: allow other users to join the network and have access to streaming songs from your personal catalog.
* Catalog management: upload owned song files for later streaming and organize them by album and artist.
* Self-hostable: make hosting the application effortless using Docker.

All the features described above should be wrapped by a familiar and intuitive web interface, in order to make searching for the desired song a simple, enjoyable, frictionless task.  
 The process of building such an application is described in the following chapters in detail. In the first chapter, the technologies and the architecture of the application will be described (client-server interraction, REST APIs, domain-driven design, React design principles and more). The following chapter describes the application itself: how it was designed, implemented and a few extra details.

**Chapter 2**

**Used technologies**

* 1. **The problem**
     1. **Overview**

The client-server architecture implies the application being split into two complementary parts that are traditionally described in the following way:

1. The server is responsible for handling requests from the client. It is commonly authoritative due to being in control of handling all resources and interractions between client and database, and also securing sensitive information via authorization and authentification. One could argue that the server is the brain of the application.
2. The client provides the user with an interractive interface through which the human can interract with the application. That usually implies both visualizing and acting upon underlying data.

Such an architecture is usually used in order to put the load on the server instead of the client (due to the fact that the server is usually better performing and has better availability) and in order to centralize data in one location to prevent costly searches.