****

**AudioWave: A Microservices-Based Audio Sharing Platform**

**Design and Implementation of a Full Microservices Architecture**

*Submitted in partial fulfillment for the degree of*

**Bachelor of Science**

*in*

**Software Engineering**

*for*

**The Software Engineering Department**

*at*

**Braude - College of Engineering, Karmiel**

*By*

**Ward Zidani**

**Ahmad Bsese**

*Under guidance of*

**Zeev Frenkel**

*Project code*

**24-1-D-44**

**Summer 2024**

**Abstract**

*Within our project we develop AudioWave an audio sharing platform based on user-generated content. We look into solving some problems that, eventually, every successful platform faces as it grows its user base. These problems include the difficulties of serving users in times of dynamically changing traffic, such as traffic peaks, and the ability to control the flow of error detection and fault singulation. Another problem the project aims to solve is the complexity of the future of integrating new features. We implement a certain architectural pattern to the backend, an architecture based on the microservices architecture. The project’s architecture is divided into separate microservices, each microservice has a main responsibility, with some minor ones. The main rule of implementing and dividing the system into microservices is that a single client request must require exactly one service to be running, meaning that if a service must not wait for an inter-service message to complete a client request. Our solution provides a fast and seamless experience of listening to audios by dividing the audios into smaller, more manageable chunks. Each chunk is a predetermined amount of seconds long, all the audios follow this predetermined chunk time-length. When a user requests to play an audio, even if not from the beginning, the backend returns the appropriate chunks of where the audio player is in the audio dynamically, doing this makes the platform faster and more performant.*

**Table of Contents**

[**1. Introduction 1**](#_99lunqdrnj3b)

[**2. Background 1**](#_u3d8ke7oq4sl)

[2.1. General 1](#_d9qv08v3ape5)

[2.2. Terminology 3](#_1byg0k3i6y51)

[**2.3. Related Solutions 4**](#_xnbb7aoyocf0)

[**3. Expected Achievements 8**](#_wml01vvyfjau)

[3.1. General 8](#_5os3xnxbcu5s)

[3.2. Technical 8](#_p5ja7jswk28i)

[3.3. Product 8](#_15f4wxmxao4e)

[**4. Requirements 9**](#_gfphgzfkn61o)

[4.1. Product’s Target Audience 9](#_2nebv1qv3uia)

[4.2. Project’s Target Audience 9](#_u83q7wom1i0c)

[4.3. Functional Requirements 10](#_9rf76fhzyu1m)

[4.4. Non-Functional Requirements 11](#_a4zsdzmvaotg)

[4.5. Diagrams 12](#_qbje3olyaj4r)

[**5. Engineering Process 17**](#_5i9d10eun0wm)

[5.1. The Process 17](#_5cklkrt4j78b)

[5.2. Product 26](#_nt5atpuh4nx7)

[5.3. Backend Design 32](#_o88grzycrhr0)

[**6. Testing Plan 36**](#_r5fco28vn5wa)

[**7. References 40**](#_yrolg24mu8h5)

# Introduction

In recent decades, the world wide web has seen an immense growth in traffic for publicly facing platforms, thanks to the increasing availability of access to the internet and the decreasing costs of personal smart devices that can access the web. It has reached a point where being connected and using the internet has become a quintessential part of billions of people’s lives around the world. People depend on the internet for entertainment, education, finance, transportation, and more, so much so that websites such as youtube can expect hundreds of millions of visits a day [1]. Every one of these visits take around a few milliseconds on average, and when taking into account that millions of these visits can take place in a small time frame, the time to process all these visits can add up and cause immense loads on the website which in turn will hinder its performance.

For such companies as Google, Youtube, Facebook, etc., it is vital to provide the fast and responsive experience that they have become accustomed to in our age of immediate access to information (the internet, basically), so that these companies can maintain a large user base to allow them to monetize their platform.

Large platforms often offer a multitude of features of various purposes. Features such as user management is a common feature in websites that is secondary to a main proprietary feature. Such features are constantly being added or change during the production life of the platform, and when a platform begins to receive large amount of traffic on constant basis, the company will need to optimize the platform to be able to withstand this load and also have the ability to add new features without the risk of affecting the overall performance.

In this project book we will design an audio sharing platform for user-generated content. The platform will attempt to solve the problem of changing traffic peaks through the separation of system contexts as to allow for the independent scaling of parts in the system that are facing the immense traffic loads.

[Link to Github Repository](https://github.com/WardZid/AudioWave)

# Background

## General

The world of software is a relatively new space when looking at it from the consumer point of view. Whereas in our current age software is so deeply ingrained in our daily lives so much that we depend on it for many trivial or mundane tasks.

Software can take many forms, ranging from highly complex systems that governments use for managing a whole country’s population taxing and revenue calculations to building agriculture simulations using probability theory, machine learning algorithms, and graphical algebra to the most simple clock widget on a smartphone where its sole purpose is to show the time.

In the early ages of computers and computer software, software was developed with the sole purpose of solving a specific scientific problem or decision problems using the Turing Machine [2]. Later, we saw the rapid advancements in the world of computer science where we saw the introduction of programming languages that made creating software much more approachable. Around the late 1980’s, we saw the release and continuous expansion of the world wide web network [3], that allowed computers to transfer data at higher speeds than ever imagined before, which in turn, saw the rise in accumulation of data that needed storage. The rise of the internet (world wide web), saw an influx of users that needed access to shared data storages such as books and articles, and of course, the average computer could not hold all the data just so the user could access a part of it, so that led to increase use of client to storage platforms where the user could access the desires pieces of data remotely without needing to store on their machine, which fueled the growth of larger data storages that could be monetized and invested in larger storage and client request traffic handling capabilities [4]. And so, software entered the consumer age. This is where the fun begins.

In the software products industry, public facing platforms (such as Facebook, Youtube, Google, etc.) are expected to handle large amounts of user traffic. Such platforms require copious amounts of processing power in order to provide the end-user with the requested resources in the shortest amount of time possible.

The following sections of this book will detail our findings in the field of data transfer and management, and how we plan to build an audio sharing platform that will implement, what is in our opinion, the most suitable way to set up a platform that addresses the challenge of handling large amounts of public traffic while keeping the response times as minimal as possible, which would be in a microservices architecture.

Microservices architecture is a software design approach where a certain software is built from multiple, smaller, independent, and loosely-coupled “micro”-services. Each one of these services is responsible for a specific business functionality, *NOT* technological requirements. An example of this would be a flight booking website where the functionality of searching for optimal flights has its own service in the backend and the functionality of viewing and managing already booked flights is handled by a separate service. This is in contrast to a monolithic architecture where a software is built completely in the same codebase.

Applications of the microservices architecture, or just simply the idea of separating services by their responsibilities as to allow individual scalability and maintainability, can be found in some of the largest platforms on the internet, such as Google, Facebook, Spotify, Soundcloud, and many more platforms that provide a type of client-user service.

## Terminology

**User-generated Content (UGC)**: Content uploaded to a platform by users seeking to share their audio content. This content would be the driving factor of the platform as it is meant to be what online users will want to access on short-notice demand.

**Scalability**: The ability to increase or decrease the capacity of load handling of a running service to meet the requirements of the current state of traffic.

**Vertical Scalability**: The increase or decrease of dedicated resources for a running service instance. E.g.: RAM, CPU power, Wattage, or anything that can assist a service to increase its work output.

**Horizontal Scalability**: The increase or decrease in the number of running instances of a service.

**Flexibility**: The capability to add, remove, or alter components in a way that does not hinder or interrupt other parts of a system.

**Domain Driven Design (DDD)**: An approach that seeks to align software systems with the domain they represent. It emphasizes close collaboration between domain experts and developers to create software that accurately models the problem domain. [5]

**Microservices Architecture**: An architectural approach to a software system where the system is divided into smaller loosely coupled services with a specific responsibility.

**Bounded Context**: A specific boundary within which a particular model or concept applies. It defines the explicit context in which a domain model is valid, ensuring that terms and concepts have clear and consistent meanings within that context. [5] (Evans, 2004, 335)

**API Gateway**: A single point of entry for all clients of the platform before the main handling of appropriate responsibility has occurred.

**Virtual Private Cloud (VPC)**: A secure, isolated private cloud hosted within a public cloud.

**Frontend**: Mainly used in the context of development, frontend is the portion of a platform that a user-client of the platform interacts with. Basically, the part that a user “sees” when they visit a certain software application.

**Backend**: Mainly used in the context of development, backend is the portion of the platform that sits behind the scenes and is responsible for the processing user requests, storing the system’s data, processing said data as needed, providing said data when requested by a user, running analytics, and keeping the data secure. In addition, it is important to note that backend systems usually sit in a centralized location on powerful computers that all the clients can request from.

**Progressive Web Application (PWA)**: An installable web application – a website that you can install on your device [11].

# 

## Related Solutions

Here we will look at two aspects of related works and similar platforms within two different scopes. The first being the platform’s business functionality, meaning what the platform does for its intended users’. The second being the architectural approach of which these platforms have decided to build their backend services in order to handle the large amounts of traffic that they were expecting.

### Audio Sharing

There exist many audio sharing platforms out there that satisfy various market needs. Although there are many of these platforms not any that were found to satisfy the corner that we are looking to occupy, that corner being an open platform for anyone to upload audios of all sorts of purposes such as recorded lectures, podcasts, spoken-word content, or anything that can have an audio file (within the rules and limits of the platform).

#### Overview of Existing Platforms

* **SoundCloud**: An audio sharing platform that focuses on a social and open UGC where anyone can go to share their music [26].
* **Bandcamp:** An music sharing and distribution platform where “Indie” musicians or just any musicians with minimal resources can go share and distribute their music in an attempt to reach a larger audience [27].
* **Spotify:** A music streaming platform that serves the public mostly free music on-demand streaming of professionally produced music [28].
* **Audioboom:** An open spoken-word content platform mainly for podcasts with a dedicated listener-base [29].

#### Comparison with Existing Platforms

This project aims to target and empower all audio listeners without focusing just on a sole genre of audio such as music.

* Unlike Spotify, anyone can upload audio content and is not restricted to being a pre-approved “artist”.
* Unlike Bandcamp and Soundcloud, the platform is not centered on music or any specific genre of audios, although there will be categories for the listeners to choose from.
* Unlike Audioboom, content creators are not expected to have a paid subscription to upload and share content.

### Microservices Architecture

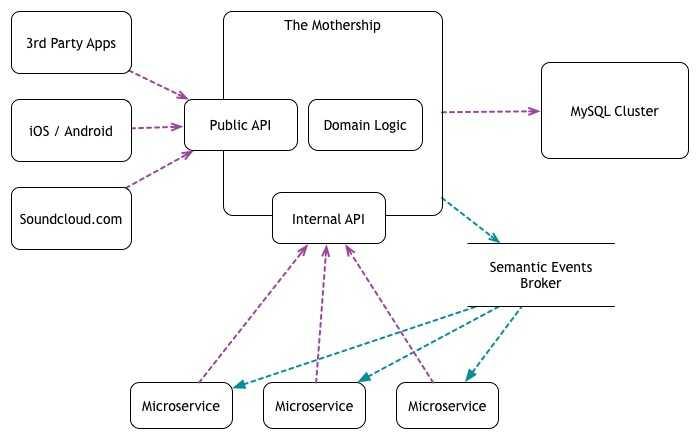
Looking at the state of software in the previously mentioned platforms, we can see that many, if not all, of the platforms use a microservices architecture for their backend. Although Bandcamp and Audioboom havent publicized their architecture, Spotify and Soundcloud definitely have, and both of these platforms boast about their usage of this architectural approach.

Soundcloud started off with a monolithic architecture, as its user-base grew, soundcloud faced difficulties scaling dynamically to handle the loads. Souncloud had to switch up their platform to a more modular microservices architecture with the help of the domain driven design pattern “Bounded Context” to help draw the limits of its microservice contexts in relation to each other [6]. As simply dividing the services without predetermined definitions and boundaries of said definitions would result in an entangled and obfuscated mess of responsibilities.

Soundcloud is at a point where its users upload more than 12 hours of audio every minute [7], on the climb to this number, Soundcloud has always been a monolithic based system, but climbing up to these numbers, the requirements grew more taxing on the system and needed a way to make it more scalable and flexible, to which they decided the best way to move forward and remedy the issue of growing traffic is rebuild the system using the microservices approach. Some reasons stated by the engineers themselves is: [7]

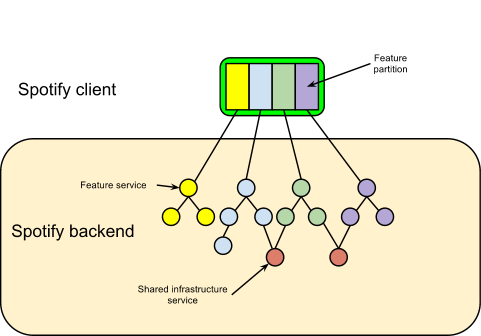
* Ability to provide a high-quality streaming service.
* Simplified the deployment of the app.
* No single point of failure.
* Enabled event sourcing to deal with shared data.
* Code Reusability.
* Freedom in the use of technology.
* Higher availability.

Soundcloud calls its microservices architecture system “The Mothership” and it works by accepting all requests into a single point of entry at a public API endpoint and only after does it route the requests according to their respective destinations (Fig. 1)[7].





Spotify is also a large backer of dividing its backend infrastructure into separate components in the microservices approach. Spotify separates its microservices by features (Fig. 2)[8]. Which as stated by their engineering team: “Feature partitioning gives scalability, reliability and an efficient way of focusing team efforts” [8].

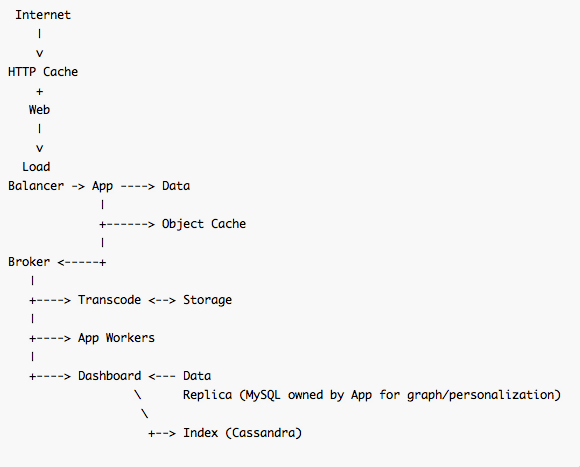


### Comparison with Frontend

Spotify's frontend has evolved from a minimalistic design to a feature-rich interface, adapting to advancements in technology and user preferences. Over the years, enhancements have focused on social integration, cross-platform compatibility, and personalized discovery features. Mobile-first design principles and seamless integration with smart devices have been prioritized, ensuring a consistent and engaging user experience across different platforms. Overall, Spotify's frontend evolution reflects its commitment to providing a seamless, personalized, and immersive music streaming experience for users worldwide.

SoundCloud's frontend (Fig 3) evolution has focused on fostering community engagement and empowering creators through features such as social interaction tools, monetization options, and integration with third-party platforms. Recent updates prioritize user-friendly navigation, personalized recommendations, and enhanced discoverability, maintaining SoundCloud's position as a vibrant music-sharing ecosystem for creators and listeners.





Bandcamp's frontend evolution has centered on empowering independent musicians through customizable storefronts and direct fan support, fostering a vibrant community of music enthusiasts. Recent updates prioritize user-friendly navigation, personalized discovery, and seamless integration with social media, maintaining Bandcamp's position as a leading platform for discovering and supporting independent music.

# Expected Achievements

## General

The expected outcome of the project is to delve into the inner workings of a microservices based platform designed with scalability, flexibility, and maintainability in mind as to demonstrate, hands on, the positive and negative attributes of such an architectural approach. By the end of this project we expect to have, in our hands, a fully working audio sharing content with the capability to host large amounts of UGC while maintaining its latency and response times at low value.

## Technical

The project should be able to dynamically route requests to the appropriate services and also utilize a load balancer to route requests equally between all running instances of a service. Although hosting large traffic is a problem that companies and startups face only a while after their launch, a company that plans to go big should look to be prepared for a scenario like this before it occurs. When a company that runs a public-facing platform does not prepare the platform for big traffic, it risks unmanaged loads and outages that push users away which, in turn, risks the failure of the platform. So building it from the start as a load-bearing ready platform will ensure it is ready for future peaks.

An important functionality to implement is load-security where the system, specifically the public API service, would be able to identify DDOS attacks or suspicious requests. When such scenarios are identified, the system would blacklist the clients.

## Product

The platform envisioned in this paper would fill a niche in the market, as we are in an age where the average human is multitasking more than ever before [9]. Especially when costs of living are rising rapidly [10], pushing people to pursue higher education or just simply learn new skills to advance their careers, which is an aspect our platform focuses on, educational audios. The platform also provides the opposite, audios for relaxation, whether it is book reading, relaxing sounds such as rainfall and white noise, spoken-word documentaries, and so on.

We expect to have a complete, ready for production system with fully compatible frontend clients and a robust backend system. The frontend would include two applications, a ReactTS web application and a Flutter mobile application. Both frontends are expected to be fully featured and both are expected to use the same data API endpoints. The backend would be a fully complete microservices based architecture with a public API Gateway as a single port of entry of the VPC where the rest of the services would reside.

# Requirements

In this section, we will define the scope of requirements for both the platform (*as a product*) and the project (*as a scientific article*). We list the audience categories and the requirements of a microservices architecture based platform. These requirements are a sum of collected information from sources such as popular features in already popular websites, feature requests in forums, and our own experiences as users over the years.

## Product’s Target Audience

* **The User**: The platform’s target client is anyone with a device that can access the internet. The user can be any or both a content creator and a content consumer.
* **The Content Creator**: This would be the person that has some kind of audio content to share with the world. This person can be a news blogger, a storyteller, a lecturer, a musician, or anyone that desires.
* **The Content Consumer**: This would be the person that desires some kind of audio resource (that would be uploaded by the content creator).

## Project’s Target Audience

* **The Software Architect:** This person is responsible for designing software systems. They would be interested in finding software solutions that best fit the requirements of a given platform. It is their responsibility to provide an architecture that would streamline development in a way that would ensure flexible, dynamic scalability and smooth deployment of the platform.
* **The DevOps Professional:** Automation of integration and deployment are at the core of what DevOps engineers do. DevOps engineers are responsible for setting up a project so that it can be developed at large scale by multiple different teams with minimal deployment lag. Their responsibilities include the automation of builds and deployments of different services asynchronously, containerization and orchestration of all the separate services in the microservices domain, and monitoring and logging of all the separate services regardless of how they are built.
* **Developer Communities:** There is no shortage of developers and SE/CS students that look to understand the world of software architecture and study all the popular architecture designs for a multitude of reasons, of which is furthering one’s career or just in the noble pursuit of knowledge.

## Functional Requirements

* **User Account Management:**
  + The system should manage user accounts for the end-users.
  + The system should verify the account with an email link.
  + Users should be able to log in to their accounts securely using an email and password.
  + Users should be able to alter their personal information that is tied to their account.
* **Search Functionality:**
  + Any user should be able to search for public audios using keywords, author names, or subject.
  + The system should present the list of relevant search results in a user-friendly manner.
* **Audio Player:**
  + Any user should be able to listen to public audios on the platform.
  + The system should let the user control the audio player using play, pause, or skipping back and forth.
* **Content Management:**
  + The system should be able to process and organize user-submitted audio content.
  + Users should be able to alter the info associated with the audios that they, themselves, uploaded to the platform.
  + The system should allow for the user to set their content privacy to public, private, unlisted, etc.
* **Content Streaming:**
  + The system should be able to dynamically load audios to the end-user while the user is in the process of listening as to provide a seamless and responsive experience.
  + The system should pre-load an initial part of the next audio in queue for a seamless and responsive experience when skipping.

## 

## Non-Functional Requirements

* **Scalability:** The platform’s services should be able to scale horizontally in order to meet a minimum performance threshold without wasting resources and computing power
* **Maintainability:** The platform, being split by “business” requirements where each lives on its own microservice, will result in separate, *smaller*, and less complex codebases which simplifies maintenance.
* **Flexibility:** The platform, being split by “business” requirements where each lives on its own microservice, makes adding, altering, or removing features much easier and more malleable per the developers’ desires.
* **Performance:** The platform is expected to reach a large audience. For the platform to succeed as a product, it is important that the end-user receives the can listen to his desired audio within a reasonable timeframe in order to keep them engaged.
* **Security:** User data and uploaded data must be kept secure in order to maintain the trust of the user and keep them on the platform.
* **Reliability:** Each feature will run on its own microservice, whereas if a failure occurs in certain microservice, it wouldn't affect others.

## Diagrams

### Use Case

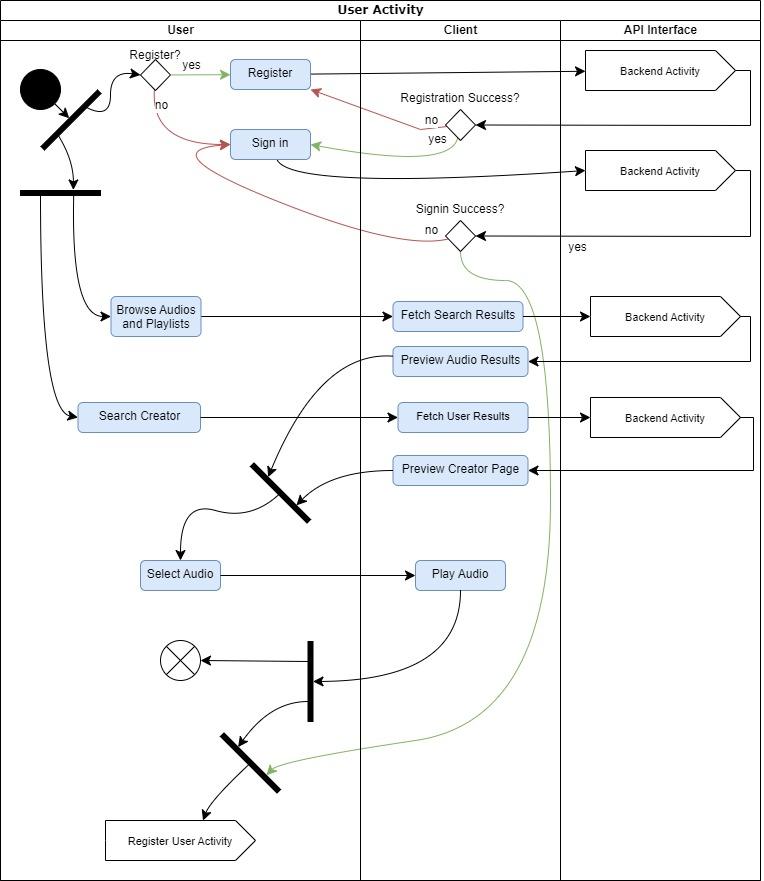
The following diagram (Fig. 4) demonstrates the scope of each type of user in the platform, the two kinds being a guest (non-registered) and a registered user that owns an account.





### Activity Diagram

The following activity diagrams roughly represent the flow in which the platform would operate. The first (Fig. 5.) be of the actions that any user can perform, that being guest users and registered ones alike. The second (Fig. 6.) would show the flow of actions a registered user can perform through their usage of the platform.





### 

### 

### 

### Sequence Diagram

The following Sequence Diagram (Fig. 7.) demonstrates how, in the backend, the Gateway Service will transfer the requests to the appropriate services.



### System Architecture

The system architecture presented (Fig. 8) shows the multiple services and their connections to each other.

### 

### 

# Engineering Process

There are two main aspects to discuss in the process of understanding the best ways to implement the system, especially when moving from a theoretical point of view to an actual working implementation, the client-side and the server-side. These two main components are separated by responsibility yet need to work together seamlessly.

## The Process

### Frontend

The world of web development is rapidly changing with new tools and software frameworks being released and altered all the time. But the trend shows that however different they might be, they are slowly converging into similar patterns of abstraction.

Since our platform is looking to be highly compatible for the everyday listener and especially focusing on not tying down the user to their computer just to listen to their desired audios, it is decided that we will implement our main client-side app as a mobile application. Alongside the mobile application, we will also create an in-browser web application for an extended reach of users. In addition, we will look more into making the web application into an installable PWA, thus utilizing the advantages of a desktop application, such as push notifications and offline service workers, without needing to build standalone,platform dependent desktop applications.

For the mobile application, there are two major discussions. The first is should the application be built in a single-platform paradigm? Or should it be built in a multi-platform paradigm? And if the latter is chosen, which framework should be used? Both of these will require a dive into the advantages of each while considering the requirements of the platform that is being built.



When considering single-platform applications, a major component to keep in mind is that since the release of the modern smartphone, there have been a multitude of mobile phone operating systems in the market, each needing a separately built app for it to work on the system. Doing so, building an app for every single mobile operating system, would require large resources in terms of development manpower for the building and maintaining of the separately built apps. Thankfully, over the past couple of decades, we saw a concentration of mobile operating systems into two main ones, Google’s Android and Apple’s IOS, with Android being the base OS of an overwhelming part of the smartphone brands, while IOS is the proprietary OS for Apple’s handheld mobile devices (Chart 1). If we were to take this as an indicator, it would mean that we would have to develop two apps only, although both apps will be identical in features, each app will be built using its target OS’s dedicated app building tools. For android devices, that would be Android Studio using either the Java or Kotlin languages, and for IOS devices, that would be XCode using the Swift language. Having apps for Android and IOS would make it so we would have covered over 99% of the smartphones in the world [13]. The advantages of this building single-platform applications is that the apps would be rendered directly by the OS and deliver a native experience to the respective OS, as well as being able to access the device’s resources less restrictively [14, p.1205]. Despite this, the single-platform paradigm is less popular today than the multi-platform paradigm, even with all the multi-platform disadvantages (Table 1), single-platform development’s requirements scare away most companies since most companies are profit hungry.

# 

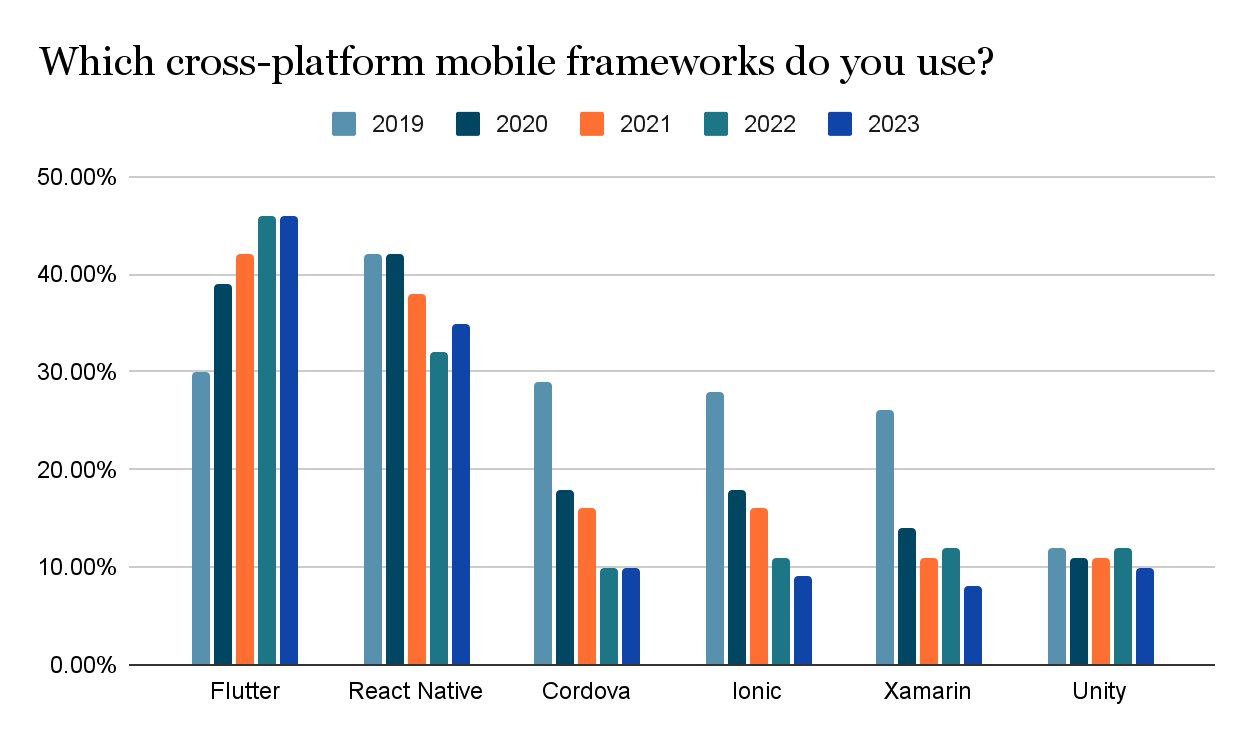
When considering multi-platform applications, things become somewhat simpler. The Worry of different operating systems becomes a problem for the creators of the multi-platform paradigm frameworks’ creators. All that remains is worrying about testing for specific relevant device form factors in the market. When going the multi-platform direction, building apps becomes more of an abstraction of the single-platform development tools, meaning that the tools that are created on top of the single-platform tools, except they have their own languages and rules. The most attractive factor of such frameworks, is that only one app is needed for all phones and OSs’ (depending on the framework), which hugely simplifies development and maintenance. As much as this seems like the ultimate solution, there do exist drawbacks where single-platform apps would excel, drawbacks such as the loss of an OS’s native experience in the graphical user interface, and more importantly, performance [15]. Performance is the biggest bump in the road in the multi-platform paradigm, since not only the development tools are an abstraction on top of the native tools, but also the apps themselves, whereas the apps have to pass through a whole layer of abstraction to be translated to the OS. Seeing that performance is the main hiccup to which multi-platform developed apps face, choosing between multi-platform development tools might come down to be decided by their performance.

Comparing these paradigms, it is evident that both offer significant advantages and disadvantages (Table 2) over the other. It is a fact that single-platform apps have a leg-up in terms of speed and performance [15], but it also comes with the downsides of requiring at least double the amount of development resources such as developers, time, project managers, integration pipelines, and more. Multi-platform apps, on the other hand, might be slower and less performant, but when considering the costs and time saved in developing the apps, it becomes a no-brainer for companies, especially companies with tight budgets and/or deadlines, since the difference could mean the success or potential decline of the app being developed. So for a small team of developers that are tight on deadlines and pursuing to release a high quality application. The potential advantages (Table 3) greatly outweigh the potential disadvantages (Table 1) of the multi-platform paradigm.





Assuming the chosen development paradigm would be a multi-platform option, we arrive at another crossroads where we must choose between frameworks. Frameworks are plentiful in number and provide a variety of development features to attract developers. Hand-picking the more notable frameworks, it is evident that there aren’t many out there and some are more popular for very specific reasons. Many of the frameworks, such as react native, gain a large part of their popularity due to the fact that they support converting already built react web applications into almost native apps. Due to that previous statement, it is important to consider the context of the platform being built. According to JetBrains’ yearly Developer Ecosystems Survey [16], 49% of mobile developers use cross-platform frameworks, of which 81% (by 2023) are using the top two options exclusively (Chart 2). From the accumulated trend data, we are compelled to look into why frameworks such as Flutter and React Native have such a hold on the developers. Flutter with its engine seems to be much more performant whilst React Native needs an intermediary Javascript translator [18].





### Backend

Since the focus of this project is to build a dynamically scalable and flexible system with a microservices architecture specifically, the question becomes more of what is the best way to implement this.

When we talk about microservices versus monolithic architecture, they are completely different architectures that serve varying requirements. Both have their pros and cons. With microservices, you're breaking down your application into smaller, independent pieces, which sounds great, because each of these pieces can be developed, deployed (Fig. 9), and scaled independently. This means faster development, easier maintenance, and fewer headaches when things go wrong. Plus, if one part fails, it doesn't bring down the whole system [19].



**Scalability** is another big win for microservices. Imagine you have a public-facing UGC platform where traffic can spike at any moment. With microservices, you can allocate resources where they're needed most, ensuring smooth sailing even during peak times [20].

But why does a microservice architecture ensure that scaling is smooth? To understand this, we need to understand how traffic works. Traffic to a service usually consists of two messages, an incoming *REQUEST* and a returning *RESPONSE* message. A request message, simply called a request, is a message that a client application, such as a website, sends to a centralized back-end server that holds the resources of the website with the expectation of a response, the response that would be appropriate to the purpose of the request. To understand the purpose of requests, we can look at a common standard for request structures such as HTTP. HTTP is an application layer protocol that defines the basic types of requests, these are GET, POST, PUT, and DELETE, and each has a specific structure to help the receiving party to identify the purpose of the request [23]. Let's focus on the GET, as it is the most basic. GET requests have the job of retrieving data for the client, such as when an article is opened by the user on a news website, of the many possible requests, a GET request is sent to the server to retrieve the desired article text, title, and anything else that the client needs, per what is agreed upon by the developers. The client sends the request that would include some networking headers and some way of specifying what resource, article in our example, is needed. The backend server receives that message, finds the requested article and forms a *RESPONSE* message. Response messages, or simply called a response, is the message reply of a party to another that made a request. Furthermore, when the server constructs the response with the appropriate data, it is sent to the client that requested the resource so it can be displayed for the user.

Now, how does this relate to “traffic”? Traffic is the coming and going of these messages to and from a machine in a network, and since we previously specified that our backend service is a centralized machine that holds data and serves the data to clients, it is responsible for serving all the clients of a specific platform. It is also important to note that when a service receives a request and goes through the process of constructing and returning a response, it is naturally required to do some kind of computational work that will require some amount of computational power resources, and when a backend service is responsible for many frontend clients that all want to make requests, even the smallest and simplest requests can overwhelm the machine at times of peak traffic. To solve this, the simplest way that can be thought of is to just strengthen the backend machine’s computational hardware, such as adding faster processors and ram, but the problem with simply adding more hardware is running this hardware is that it will expend valuable monetary resources, like electricity, or simply shorten its lifespan, and buying the hardware is even more expensive.

A solution to resource wasting would be to add dedicated hardware at times of traffic peaks and to remove the hardware later, programmatically or physically, called *VERTICAL SCALING* (Fig. 10). What that does is it ensures a lengthier lifespan of the hardware and saves monetary power costs and maintenance. But yet again that introduces another problem, which is that when the hardware is added or removed, the software will need to reboot and momentarily go offline, leaving users without being able to use the platform. A solution for this would be running another instance of the backend service and routing part of the incoming requests to the new instance, where both running instances of the service software would be running concurrently, this would distribute the load between the instances, and relieve the request bottleneck while completely abiding downtime and will keep the users engaged with a seamless experience [24]. This is how we achieve seamless dynamic scaling in a system and it is called *HORIZONTAL SCALING* (Fig. 11).





Despite this,the previously mentioned statements still don’t necessarily show why a microservices architecture is needed when dynamic scaling is absolutely achievable for a monolithic system. It can be improved. A monolithic system would have all the features and functionalities of the platform implemented into one codebase, meaning that each instance would have multiplied all the functionality request capabilities with each instantiation, which introduces a drawback of the monolithic architecture, that drawback is that some functionalities don't need to be scaled, since they don't receive as many requests as others, for example, for a streaming platform such as our audio sharing platform, a user signs up an average of once in their lifetime, but they will play large amounts of audio minutes. A user will also log in only once every session, but listen to possibly tens of minutes. This means that the requests for audio dta will outweigh user management requests greatly, but every time we scale the monolithic system horizontally in the case of audio listening request spikes, we are using up resources for user management functionalities being multiplied without needing to. This is where we introduce a core aspect of the microservices architecture, which is the breaking up of the backend system into separate services, each service is responsible for a specific business functionality of the platform. There can be a Audio Data Service, and another User Management Service, and each one of those would have its own codebase and run as separate programs, which allows for separate scaling by only adding instances of the services facing higher traffic. For each one of these services, there would live a dedicated load balancer to direct requests evenly between the instances. This way it is possible to only scale a portion of the system, leading to fewer resources being wasted on parts of the program that dont need to be multiplied.

**Maintainability.** A large benefit of separating the services is maintainability. Each service can be maintained and updated separately. New versions of a service can be released without affecting or needing to alter other services at all, if there happened to be some kind of fatal security bug or any similar issue that would require rolling back to a previous version, it can be done without rolling back updates and features that were added after the bugged version. This helps keep the system, even at faulty times, easier to maintain with fewer drawbacks and sacrifices.

**Reliability** is also a huge point of interest for those using or looking to implement a microservices architecture in their system. Splitting up a monolithic system into separate services increases fault damage and fault maintainability. This is not something that needs to be implemented directly, but it is something that comes naturally with the splitting up of the services. Assume a fatal fault occurs in service A that completely crashes it, since service B runs separately, it remains mostly unaffected, other than if it needed a resource from service A, but properly defined service contexts are built by assuring that services are able to function without needing needing others, for the most part. Although sometimes there does exist a need for inter-service communication, it should be avoided as much as possible.

So we split up the system and introduced load balancers. How does a request know which service to go to without needing to redesign the frontend client’s API calls’ infrastructure? For that we need to introduce a central, single point of entry, that would be responsible for routing the requests appropriately. Here we introduce the *API GATEWAY*. The API Gateway acts as a single point of entry to the backend system (Fig. 12), its job is to receive the requests, take a quick look at them to understand their context, and route them to their proper service for further handling [25]. The gateway would be a running program that has a public API and can be reached from the client using its public IP Address or more commonly, a domain name. The API Gateway is a crucial part of a microservices architecture since it can take many responsibilities, such as authentication, cyber-security handling, and even caching data for common requests to reduce response time.



**Freedom of Frameworks**. Microservices give developers the freedom to choose the tools and languages that work best for each service. So, while one team might be using Node.js, another might be jamming with Java. This flexibility means each service can be optimized for peak performance [21]. Let’s take for instance a service that is responsible for analyzing data, finding trends, or just anything that has to do with handling large amounts of data, the team developing this service would most likely use the Python programming language with tools such as numpy and pandas. Another service might have speed and efficiency requirements, the team developing that service might choose to use high performance languages like C, C++, or Rust. Another advantage is choosing between types of databases for each service, a service might want to use a more concrete approach for storing data in a relational database like MySQL or MSSQL, while another might not want to be tied down with indexes and mandatory columns and choose a more efficient no-sql approach.

But, like all good things, microservices come with their own set of challenges. Distributed systems can be tricky to manage, with issues like network delays and keeping everything in sync. Plus, managing a multitude of different services requires well-built infrastructure and monitoring [19]. This infrastructure will include:

* API Gateway (as a reverse proxy[[1]](#footnote-0))
* Separate microservice per decided context
* Inter-microservice domain messaging

Setting up a custom API Gateway can be a challenging task and is not often the best way to go, as most API gateway tools in the market are easy to set up and offer most of the features a system might need. The gateway is expected to provide functionalities such as [30]:

* Dealing with security, throttling, and caching
* Routing requests to backend microservices
* Discovery of running microservice instances
* Health checking the microservices instances

Building these from scratch would sidetrack the project since these features are complex and require a great deal of time and effort while not providing any benefit to the vision of the project. Two intriguing options are Kong Gateway and Traefic’s API Gateway. A simple comparison of these on stackshare (Table 4), shows that, even though Traefik is essentially simpler to implement and offers built in service discovery, Kong offers higher performance and a more hands-on approach of which is the aim of the project. Kong also allows the defining of specific services in the network, this is useful because the project is aimed at implementing the parts of the microservices as a system, not worrying about DevOps tools and practices.

|  | **Kong** | **Traefik** |
| --- | --- | --- |
| Flexibility | More flexible with plugins, but requires separate service discovery. | Easier to use, built-in service discovery, good for cloud-native environments. |
| Routing and Load Balancing | Offers a wide range of algorithms for configuration. | Automatic configuration based on service discovery. |
| Health Checking and Security | Basic health checks, supports TLS termination. | More advanced health checks, automatic certificate provisioning. |
| Overall | Powerful and customizable, but requires more setup for beginners. | Easier to set up and use, good for dynamic environments. |
| Table 4. Comparison between Kong’s and Traafiks API Gateway [31] | | |

Presented solution. Properly defining contexts of each microservice is what this project aims to improve. A certain backend service *MUST* be capable of completing a client’s request independently, if it is not capable of doing so, the context of the service has been incorrectly defined. Although sending asynchronous inter-service messages using message-brokers is permissible, it must not await a response.

In the end, it's all about balance. By weighing the pros and cons and following best practices, we can harness the power of microservices to create an audio sharing platform that's both responsive and reliable.

## Product

### Features

#### Upload audio:

Users can upload audio files, so when the file upload, we have two main databases to save the data files and the audio filecontent:

First the user must provide details about the file such as title, description and other details that must the user insert, so these details are saved to the metadatabase with other details that relate to the file that the system saves automatically for instance: user name, id file name, date.

Second, the audio file content is saved to the audiofile database as blocks, each file is stored in the audio file database as blocks of seconds, to reduce the file size, so when other users start to listen to this audio file, the system begins transferring the data to the user in blocks of seconds. This idea enhances the ease of transferring data to the user directly as parts, without a relay while listening to the audio file.

#### Resume upload:

If the upload performance is interrupted for any system reason like disconnect from server or wifi lost, the user can still continue the upload system without the need to perform the whole process again from where it was stopped, we will depend on file specific details to build an algorithm to perform this operation.

#### Login and registration operations:

Every time a user successfully logs in, their client will receive a JWT access token to be bundled with all their requests. This taken will have a set expiration date.

#### Tags:

For each file, the uploader can add tags, these tags will sit under the main category that can show up for interested listeners of the same tags. It's not mandatory to add tags for each file, but if an uploader wants to increase their chances for higher viewership of their content, adding the appropriate tags will help them reach their targeted audience more precisely since the tags will be used to recommend similar audios for their respective listeners.

#### Search technique

In our system there is a comfortable way to perform a search, users can choose author search or title search or tags search or leave it as general search.

#### Download files

Users also can download files so they can listen to the files in areas outside network coverage.

#### History:

For user comfort, the platform gives a history log to the user.

#### Time period:

While playing an audio content, users can adjust the audio timeline by clicking on a specific tab to move the audio forward or backward.

#### Play list:

By clicking on add to playlist button, the audio will be added to the playlist.

## 

### Client Design

Designing the client’s implementation strategy brings to the question of what is the plan to meet the requirements of the platform. Taking into account both Functional and Non-Functional requirements, as to narrow down our design decisions that helps us design it in a shorter time and hopefully implement it quicker.

As per our largest concern, we need the platform to be fast, with many close-to-native features such a background playing audios and offline listening, and per the research done the decision it seems most suitable to go with ***Flutter***. Flutter is fast, performant, and universally loved by developers for its ease of use and intuitiveness.

The first order of business is to look at the big picture of the platform. The platform has a central feature that all others will be built around, and that is listening On to discussing the features from the first screen in the app.

1. **Main Pages**: This part will show the main pages of the app. These are the pages accessed as the landing pages of each item in the navigation bar.

| The ***Home*** page is the landing page of the app and can be accessed through the first item in the navigation bar. This screen consists of a top section that displays “topics” relevant to the user. The main section of the page is an infinitely scrollable list of recommended audios to the user based on their past listening. |  |
| --- | --- |

| The ***Search and Explore*** page can be accessed through the second item in the lower navigation bar. This page will display a text-input search at the top. Under the search, there are three sections. First is a horizontally scrollable trending collection of audios that are not related to the listener’s habits. Second is a horizontally scrollable collection of trending playlists that are also not relevant to the user. Third and last is a two column, vertically scrollable collection of topics. |  |
| --- | --- |

| The ***My Listening*** page can be accessed through the third item in the navigation bar. At the top of the pages are playlist looking components that will hold the “Liked Audios” of the logged in user and the “History” of audios listened to in the past.  Under the top components is a vertically scrollable, two-column list of playlists created by the user. |  |
| --- | --- |

| The ***My Profile*** page can be accessed through the fourth item in the navigation bar and shows the logged in user’s “Creator Page”..  At the top sits the profile picture and the user’s Display Name (username) as chosen by the user.  In the top right is a “More” button that when selected shows more options such as Sharing a link to the profile and editing the profile |  |
| --- | --- |

1. **Screens**: The following screens are the screens that can be reached through various buttons or selections.

| The ***Audio Player*** screen is the screen for the main audio controls. This screen allows the user to play/pause, skip a time period or return back, like the audio, and add the audio to a playlist. |  |
| --- | --- |

| The ***Upload New Audio*** screen is what pops up in the profile when choosing to upload an audio.  The user will enter a title and a description, they will choose the visibility option, upload the audio file, and add a thumbnail. |  |
| --- | --- |

| The ***Login*** screen is what is displayed when entering a page that requires the user to be logged into their account. |  |
| --- | --- |

1. **Components**

| The ***Minimized Audio Player*** is a floating rectangular component that, when an audio is chosen for playing, will be situated at the bottom of the screen just above the navigation menu.  The audio bar contains The title of the currently playing audio and a play/pause control.  When the bar is clicked, excluding the play/pause, the full screen Audio Player slides up. |  |
| --- | --- |

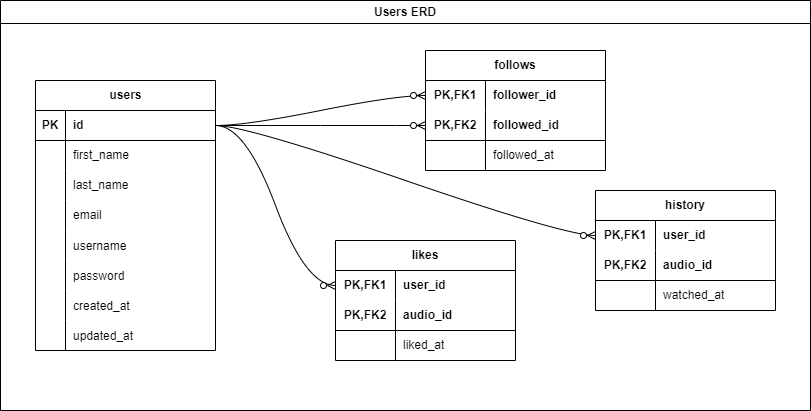
## Backend Design

These features will be split into contexts that are defined by their ability to function independently for a specific request. Meaning that a request needs exactly one service instance to be running to be completed.The features with their requirements that will be implemented are as such:

* Registration
  + Requires access to the user information
* Login
  + Requires access to the user information
* Edit personal account information
  + Requires access to the user information
* Upload Audios
  + Requires access to the audio metadata
  + Requires access to the audio chunk
* Manage Uploaded Audios (Hide/Delete)
  + Requires access to the audio metadata
* Play Audios
  + Requires access to the audio metadata
  + Requires access to the audio chunks
* Search for audios by query
  + Requires access to the audio metadata
* Create playlists
  + Requires access to playlist information
  + Requires access to audio metadata
* Manage Playlists (Add/Remove Audios)
  + Requires access to playlist information
  + Requires access to audio metadata
* Like Audios
  + Requires access to users information
* Follow Creators
  + Requires access to users information

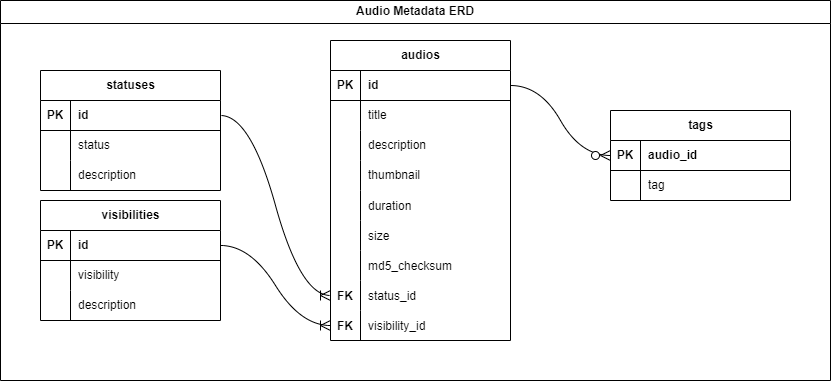
The platform will store the following:

**User Data** (Fig. 13) is the information related to a user such as username, password, email. It would be mostly suitably set up in a secure relational database. So for its high-security closed-source nature [32], *Microsoft SQL Server* is the best option. \*Note: All passwords will be encrypted using the SHA hashing algorithm, each password with its unique Salt.



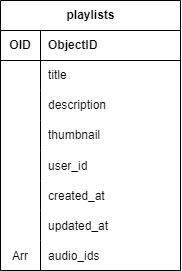
### 

**Audio Metadata** (Fig. 14) is the information about the audios that includes such things as titles, length, uploader, MD5 Checksums, original audio file metadata, and more. Seeing as there is a need for object storage such as original file metadata that would be in the ID3 file format. For this reason, *MS SQL Server* will be used for the simple reason that our base requirement is needing a structured database and its great integration with .NET Core.



### 

**Playlist Information** (Fig. 15) is fairly straightforward, playlists will have just the title, creator, thumbnail, description, and a list of audio UIDs. For the functionality of holding the audio UIDs as an array of a playlist, *MongoDB* will be used for its NoSQL property that allows for array objects-types.



### 

**Audio Files** require file structuring as to hold the file chunks of the audios. Audios will be divided into smaller, more manageable chunks. These chunks will be given, as a unique identifier, the UID of the audio in the metadata service and the sequence number of its placement in the file. The best option for this would be a distributed file system, but as that is not an available resource for this project, the best option would be the cloud storage service *Amazon’s S3*. The Storage will be accessed through S3’s API calls. AudioWave will, as a base, only support the upload of MP3 format files. The audio file chunks will have, as a unique filename, the ID of the audio in the metadata database + the sequential number of the chunk in the audio. An example would be “1111-4” where 111 is the id in the audio metadata db and 4 is the fourth 5-second section of the audio, the 15 through 20 seconds of the audio.

**The Services**:

The system will be split up into services per previously mentioned contexts of data storage, other than for the audio file storage that will be accessed by two separate services. A ***File Management Service*** that will serve for uploading and possible altering of files, and a ***Playback Service*** that will serve the audio chunks per requests.

The users database will be accessed through a dedicated ***Users Service*** that would be responsible for creating new accounts and completing login actions, as well as adding liked audios to a user’s account and managing flower connections.

The Audio Metadata will be accessed by a ***Metadata Service*** that will be responsible for inserting new audio data and retrieving information about these audios. This service is also expected to manage recommendations for a user by tag relations.

The playlists and their contents will be accessed and managed by its own ***Playlist Service*** service since it is not bound to any other functionalities. The playlists will hold a list of audio UIDs that would be used to fetch audios later.

For all the backend services, *.NET Core*will be used. Dotnet core is a robust backend web service framework that offers high speeds along with many helpful, and straightforward dependencies. Dependencies for such things as database SDKs, compression, encryption/hashing.

That said, all backend microservices will live in their own dockerized environments as to keep the system portable and ready for future integrations into such technologies as container orchestrations (Docker Swarm, Kubernetes, etc.) with the option of adding load balancers, and general cloud infrastructures.

Per the limited available timeframe of the project, no proprietary API Gateway will be developed. *Kong Gateway* will be used as a generic API Gateway that can later be switched with a proper gateway that fits the needs of the system.

Every system requires inter-service communication, so for that, *RabbitMQ* is the best solution for its multitude of features and its simplicity.

# Testing Plan

The primary objectives of testing are to ensure the functionality, performance, and cover maximum users experience to discover mistakes. This plan covers the testing of key features and components of the main body. We will focus on the testing to cover manual and automated testing techniques. Functional testing, and usability testing, The test strategy will follow a black-box testing methodology to simulate user performance and validate expected outcomes.

### Testing with Desktop or laptop computers. These computers should meet minimum hardware ability.

**Testing with Mobile Devices**: For testing mobile applications, we'll verify the availability of various mobile devices representing different platforms (iOS and Android). These devices will be used to test the compatibility and functionality of the mobile applications across different devices.

**Software Requirements:**

For the operating systems, we gotta make sure this testing setup works with all kinds, like Windows, macOS, Android, and iOS. This way we cover all the bases and test the platform on the systems people actually use. When it comes to web browsers, we need to check that everything runs smoothly on the big ones, like Chrome, Firefox, and Safari. Testing on different browsers helps us catch any weird stuff that might happen depending on which browser folks are using. And for testing tools, we'll make sure we have the right ones, like Selenium for web tests and XCTest for iOS stuff. These tools are super important for running tests automatically and making sure everything works like it should. By double-checking all this hardware and software stuff, we make sure our testing setup is good to go. This way, we can really put the audio sharing platform through its paces and make sure it works great on all kinds of devices and systems.

| Test | Module | Testing function | Expected result |
| --- | --- | --- | --- |
|  |  |  |  |
| 1 | User management | User Registration | New user registered successfully. |
| 2 | User management | User login | Registered user loged in successfully |
| 3 | Audio management | Audio upload | Uploaded audio file successful |
| 4 | Audio management | Audio playback | User can play uploaded audio file |
| 5 | Search | Keyword | Search using keywords return relevant results |
| 6 | Search | Author | Search using author name returns relevant results |
| 7 | Search | Subject | Search using subject returns relevant results |
| 8 | Content management | Edit content | User can edit uploaded audio content |
| 9 | Content management | Delete content | User able to delete uploaded audio content |
| 10 | Performance | Load testing | Platform should maintain stable performance under high load |
| 11 | Performance | Stress testing | Platform should handle stress scenarios without crashing |
| 12 | Audio management | Upload resume | User should be able to resume upload audio if it was uncompleted |
| Stress platform performance testing | | | |
| 12 | User management | Concurrent user registration | Multiple users should be able to register same time successfully |
| 13 | User management | Concurrent user login | Multiple users can log in same time successfully |
| 14 | Audio management | Concurrent audio upload | Multiple users should be able to upload audio files same time successfully |
| 15 | Audio management | Concurrent audio playback | Multiple users should be able to play audio files same time successfully |
| 16 | Search | Simulated user searches | Multiple users should be able to use search tools same time successfully |
| 17 | Content management | Concurrent content editing | Multiple users should be able to edit content same time successfully |
| 18 | Content management | Concurrent content deletion | Multiple users should be able to delete content same time successfully |
| 19 | Performance | Scalability testing | Gradually increase the number of concurrent users and measure system performance to ensure scalability |
| 20 | Performance | Response time testing | Measure the platform's response time under different load conditions to ensure it meets performance requirements |
| 21 | Performance | Resource utilization testing | Monitor CPU, memory, and network usage during load testing to ensure optimal resource utilization |
| 22 | Performance | Stress testing | Apply extreme load conditions to the platform to identify its breaking point and assess how it recovers |

# 

# References

1. Similarweb.com (2024 *updated regularly*). “[Youtube Traffic Analytics](https://www.similarweb.com/website/youtube.com/#overview)”
2. Turing, Alan Mathison (1936). "[On Computable Numbers, with an Application to the Entscheidungsproblem](https://www.wolframscience.com/prizes/tm23/images/Turing.pdf)" (PDF). Journal of Mathematics.
3. "[InfoWorld](https://books.google.co.il/books?id=wDAEAAAAMBAJ&q=compuserve+to+mci+mail+internet&pg=PT31&redir_esc=y#v=snippet&q=compuserve%20to%20mci%20mail%20internet&f=false)". 25 September 1989.
4. Quickbase. “[A Timeline of Database History & Database Management](https://www.quickbase.com/articles/timeline-of-database-history)”.
5. Evans, E. (2004). “[Domain-driven Design: Tackling Complexity in the Heart of Software](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)”
6. Jayamini Samaratunga (2021). “[Microservices Architecture used by SoundCloud](https://samaratungajs.medium.com/microservices-architecture-used-by-soundcloud-58200ac192e8)”
7. Phil Calçado (2014). “[Building Products at SoundCloud —Part I: Dealing with the Monolith](https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith)”
8. Spotify Engineering (2013). “[Backend infrastructure at Spotify](https://engineering.atspotify.com/2013/03/backend-infrastructure-at-spotify/)”
9. Kevin P. Madore, Ph.D. and Anthony D. Wagner, Ph.D. (2019). “[Multicosts of Multitasking](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7075496/)”.
10. Greg Daugherty (2023) “[History of the Cost of Living](https://www.investopedia.com/ask/answers/101314/what-does-current-cost-living-compare-20-years-ago.asp)”
11. Ophy Boamah (2024) “[What are Progressive Web Apps? PWA Guide for Beginners](https://www.freecodecamp.org/news/what-are-progressive-web-apps-pwa-guide/#:~:text=Progressive%20Web%20Apps%20(PWAs)%20are%20simply%20installable%20web%20applications%20%E2%80%93%20websites%20that%20you%20can%20install%20on%20your%20device%2C%20much%20like%20you%20would%20install%20an%20app%20from%20an%20app%20store.)”
12. Maurizio Naldi (2016) “[Concentration in the mobile operating systems market](https://arxiv.org/pdf/1605.04761)”
13. Statcounter.com (2024 *updated regularly*) “[Mobile Operating System Market Share Worldwide](https://gs.statcounter.com/os-market-share/mobile/worldwide)”
14. Luis Corral, Andrea Janes, Tadas Remencius (2012). “[Potential advantages and disadvantages of multiplatform development frameworks – A vision on mobile environments](https://www.sciencedirect.com/science/article/pii/S1877050912005303)”
15. Biørn-Hansen, A., Rieger, C., Grønli, TM. et al (2020). “[An empirical investigation of performance overhead in cross-platform mobile development frameworks](https://doi.org/10.1007/s10664-020-09827-6)”. Empir Software Eng 25, 2997–3040 (2020).
16. JetBrains.com (2023 *updated yearly*). “[The State of Developer Ecosystem 2023](https://www.jetbrains.com/lp/devecosystem-2023/)”
17. JetBrains.com (2023 *updated yearly*). “[General Development Trends](https://www.jetbrains.com/lp/devecosystem-2023/development/#:~:text=Which%20cross%2Dplatform%20mobile%20frameworks%20do%20you%20use%3F)”
18. Vlad Prudnikov (2023). “[Flutter vs React Native: Which Framework Wins in 2024?](https://blog.flutter.wtf/flutter-vs-react-native/#:~:text=When%20it%20comes,slowdown%20in%20performance.)”
19. Sam Newman (2015). “[Building Microservices: Designing Fine-Grained Systems](https://books.google.com/books/about/Building_Microservices.html?id=jjl4BgAAQBAJ&redir_esc=y)”
20. Lewis, J., & Fowler, M. (2014). “[Microservices: a definition of this new architectural term](https://martinfowler.com/articles/microservices.html)”
21. Chris Richardson (2018). “[Microservices Patterns: With examples in Java](https://books.google.com/books/about/Microservices_Patterns.html?id=UeK1swEACAAJ&redir_esc=y)”
22. Mehmet Ozkaya (2023). “[Architecture Comparison: Monolithic vs Microservices](https://medium.com/design-microservices-architecture-with-patterns/architecture-comparison-monolithic-vs-microservices-4109265c4806#:~:text=requirements%20and%20capacity.-,Architecture%20Comparison%3A%20Monolithic%20vs%20Microservices,-By%20this%20article)”
23. Henrik Nielsen , Roy T. Fielding , Tim Berners-Lee (1996). “[Hypertext Transfer Protocol -- HTTP/1.0](https://www.rfc-editor.org/rfc/rfc1945.html#section-1.1:~:text=.%20%20Introduction%0A%0A1.1-,Purpose,-The%20Hypertext%20Transfer)”
24. Nadejda Alkhaldi, Kirill Stashevsky (2023). “[What is software scalability, and why should your company take it seriously?](https://itrexgroup.com/blog/what-is-software-scalability/)”
25. J. T. ZHAO, S. Y. JING, L. Z. Jiang (2018). “[Management of API Gateway Based on Microservice Architecture](https://iopscience.iop.org/article/10.1088/1742-6596/1087/3/032032/pdf)”
26. SoundCloud . “[SoundCloud Terms of Use: Description of the Platform](https://soundcloud.com/terms-of-use#:~:text=Terms%20of%20Use.-,Description%20of%20the%20Platform,-The%20Platform%20is)”
27. John, Tozzi (2011). “[Helping Indie Musicians Market Their Tunes](https://system.privco.com/company/bandcamp#:~:text=Bandcamp%2C%20Inc.%20is,for%20independent%20music.)”
28. Spotify. “[About Spotify](https://newsroom.spotify.com/company-info/#:~:text=Get%20Spotify-,About%20Spotify,-Spotify%E2%80%99s%20platform%20revolutionized)”
29. AudioBoom. “[About Us](https://audioboom.com/about/us#:~:text=Sign%20In-,About%20Us,-We%20make%20podcasts)”
30. Sanjay Gadge, Vijaya Kotwani (2017). “[Microservice Architecture: API Gateway Considerations](https://mainlab.cs.ccu.edu.tw/presentation/pdf/(2017)Microservice-Architecture-API-Gateway-Considerations.pdf)”
31. Stackshare Comparison .“[Kong vs Traefik](https://stackshare.io/stackups/kong-vs-traefik)”
32. Microsoft. “[Licensing Resources and Documents](https://www.microsoft.com/licensing/docs)”

1. A reverse proxy is a server that sits in front of web servers and forwards client (e.g. web browser) requests to those web servers. [↑](#footnote-ref-0)