ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BACHELOR IN INFORMATION TECHNOLOGY ENGINEERING

VOICEWORM: A WEB TOOL FOR THE OPTIMIZATION OF VOICE TEACHING

Bachelor Thesis in Web-related Technologies

Advisor:
Dr. Giovanni Delnevo

Candidate:
Aisja Baglioni

Co-advisors:
Dr. Kelvin Olaiya
Dr. Manuel Andruccioli

Session III March 2026
Academic Year 2024-2025

To my grandfather Paolo,
the memory of you reminds me to never give up on music.

Contents

List of Figures

Introduction

Singing instruction occupies a special place in music education. While many other sectors have adopted digital tools to facilitate distance learning, vocal training has long been dependent on direct interaction between teacher and student. The internal nature of the voice, the need for careful listening, and the reliance on real-time sound have made it difficult to envisage an effective transition to a virtual environment. Although music production has evolved greatly over the years thanks to advancements in computer science and sound synthesis, the field of vocal teaching has remained largely unchanged. One-to-one online singing lessons are still considered challenging for both teachers and students. There are three main reasons why lessons are still taught in person: it is easier for the teacher to listen to the human voice and provide feedback; students don't need home recording equipment and a stable internet connection; there is no latency between the piano accompaniment and the student's voice.

Prior to 2020, online lessons were considered an impractical fallback due to technical issues and the substandard learning experience they offered. However, the pandemic radically changed the needs of the education sector, forcing music teachers to find ways to continue working despite the restrictions. During this period, video call software such as Zoom introduced specific audio options to support musicians by counteracting noise reduction. Other software, such as Yamaha Syncroom, Jamulus and Sonobus, focuses solely on audio synchronization and latency reduction for jam sessions and online performances. However, these options are not widely known, and those not accustomed to using personal computers may find them difficult to install. It remains desirable for all types of users to be able to access these kinds of facilities. For the average user, it is much easier to navigate a website than to install, configure and periodically update specialised audio software. Web applications do not require manual installation, work directly in the browser,

and minimize the risk of incompatibility. This makes them inherently more accessible to students and teachers who are not very familiar with computers.

During singing lessons, it is crucial that the teacher and student are synchronized. This is particularly important during vocal warm-ups. In a traditional setting, the teacher accompanies vocal exercises on the piano, and the student repeats them in real time. Online, however, even a slight delay in audio transmission can disrupt this synchronization, as the teacher plays the pattern on one side of the screen while the student hears and sings it with latency, making precise coordination difficult. Vocal warm-ups consist of exercises structured according to the principles of vocal hygiene and the singer's specific technical needs. These exercises are based on short melodic patterns that are repeated several times and progressively modulated by an interval (typically a semitone or a tone) before being articulated in ascending and descending sequences that cover the entire vocal range evenly.

Before beginning the design process, it was necessary to consider the role of the instructor in voice teaching and the delicate balance between technological support and the replacement of human labour. The central question was how much activity could be automated without affecting the personalisation of pedagogical interaction. The intention was not to create a tool that would replace the teacher, but rather one that would support them during online lessons by mitigating the risk of technical issues and allowing them to focus on the qualitative aspects of vocal training. A clear ethical line was therefore delineated: a tool would be beneficial as long as it assisted both parties, without eliminating the necessity for expert supervision.

It was these practical needs that led to the development of VoiceWorm, a tool designed to streamline the creation, organization, and sharing of vocal exercises. Developed using web technologies such as JavaScript, HTML, CSS, and the Vue.js framework, its goal is to optimize the execution of vocal warm-ups during online lessons and individual study. Teachers can consult existing exercises or publish new ones. Each warm-up can then be customized by adjusting the execution range to suit different vocal ranges and functional

requirements. The platform also allows users to change the speed, pause exercises, and export them as audio files to share with students for daily practice. All exercises can be labelled and stored in the system database, enabling users to build personal libraries of organized warm-ups for educational purposes. This structure makes VoiceWorm not only a support tool, but also a useful pedagogical archive for planning progressive and consistent vocal training programs.

This thesis covers the entire development and planning process of the VoiceWorm project. The discussion ends with a requirements validation study carried out with educators from the Voice Evolution Institute, a leading institution in vocal training, recognized for its approach based on scientific evidence and up-to-date teaching methodologies.

Thesis structure

This thesis is organized into the following chapters:

1. ***Introduction***: We introduce the research context and the main objectives of the thesis, followed by an outline of the manuscript's structure.
2. **Chapter 1,** *Voice teaching*: We provide a theoretical overview of vocal pedagogy, including vocal technique and exercises. Subsequently, we describe musical notations such as music sheets, leading into a discussion on online teaching contexts and digital tools for vocal education.
3. **Chapter 2,** *Technological stack*: We describe the technological stack used to develop the VoiceWorm web application, focusing on the web development technologies, the frontend and backend frameworks, as well as the music notation system adopted.
4. **Chapter 3,** *Analysis and design process*: We detail the requirements analysis, the architectural design of the platform, and the adopted solutions to address the technical challenges of the project.
5. **Chapter 4,** *Implementation*: We discuss the development of the frontend and backend modules and the custom abc notation parser.
6. **Chapter 5,** *Requirements validation*: We present the results of the study conducted with voice professionals, analyzing the effectiveness of the tool in real teaching scenarios.
7. ***Conclusions***: We draw conclusions on the contributions given in this thesis, based on the analytics and feedback collected. Finally, we propose potential future developments and improvements for the project.

1 Voice teaching

> Did you know that the human voice is the only pure instrument? That it has notes no other instrument has? It's like being between the keys of a piano. The notes are there, you can sing them, but they can't be found on any instrument.
>
> — Nina Simone

Nina Simone's reflection highlights an essential characteristic of the voice: its nature as a 'pure instrument', as it is a flexible, biological system capable of producing a continuous spectrum of pitches and timbres. The physical principle behind musical instruments is relatively straightforward: a vibration is generated by one system and transmitted through a medium, such as air, and then amplified and shaped by the resonating body of the instrument. The tonal qualities usually remain constant, unlike the human voice, which depends on an interconnected body system [1]. Modern singing pedagogy is based on a scientific approach that considers the body, sound and brain to be three aspects of the same vocal phenomenon. Consequently, a singing lesson is an educational process that guides the singer through stages of motor organization and musical development, rather than just a technical exercise [2]. Teachers must pay close attention to every detail, which can be difficult to achieve in an online context. To understand the limitations of distance learning, it is helpful to consider the structure and tools of traditional face-to-face lessons. Once the use cases have been identified, we can take a look at the digital tools that are already available to support teaching and suggest solutions for future implementation, considering how any shortcomings could be addressed in an online environment.

1.1 Periodization of vocal training

Singing lessons are tailored to the individual needs of each student. Vocal pedagogy is inherently flexible because no two voices are identical [3]; therefore, the content and focus of each session can vary significantly. Despite this variability, most lessons follow a similar framework, alternating between learning technique and putting it into practice [2].

1.1.1 Warm-ups

Vocal warm-ups are an essential, non-negotiable component of this structure as they prepare the voice for efficient and healthy phonation. They facilitate a gradual increase in tissue temperature and blood flow to the laryngeal muscles. This vascular shift improves the viscoelastic properties of the vocal fold mucosa, significantly reducing the tissue's viscosity and thereby lowering the phonation threshold pressure (PTP) required to produce sound [4]. This phase often involves exercises using the semi-occluded vocal tract (SOVT), such as lip trills or humming. These exercises use controlled airflow to increase inertive reactance, which helps stabilize subglottic pressure and square the vocal fold closure without inducing excessive collision stress. These adjustments optimize the conditions for economical and stable voice use, minimizing the risk of injury during high-intensity performance [5, 6]. In a typical lesson, this process is both interactive and guided. The teacher plays progressive melodic patterns accompanied by chords on the piano for the student to listen to and reproduce. Warm-ups usually start in the comfortable middle range before expanding to the higher and lower extremes of the tessitura[1].

1.1.2 Vocal athletics

Vocal athletics are the core of advanced technical training, effectively functioning as the strength and conditioning element of the vocal regimen. They are designed to challenge students by expanding their dynamic range and developing the fine motor control necessary to achieve specific acoustic out-

[1] The range of notes in which the voice sings with greater comfort and tonal quality.

comes in all kinds of situations. Vocal athletics demand excellent coordination to achieve the desired acoustic output safely. Because these exercises involve progressive loads, they necessitate a stable and healthy vocal setup to avoid strain [7].

1.1.3 Cool-downs

Just as athletes require a period of tapering off after intense physical exertion, vocal cool-downs are essential for returning the voice to a state of physiological rest. The main aim of this stage is to reduce any muscle tension that has built up during high-intensity phonation and to return the larynx to its neutral, resting position. Low-intensity exercises such as descending sirens, gentle humming or light SOVT tasks encourage relaxation of the muscles and release constriction in the vocal tract. This process effectively rebalances vocal fold vibration, preventing functional overload. From a pedagogical perspective, the accompaniment plays a psychological role here; typically, the teacher shifts to softer, slower piano patterns with resolving harmonies[2] that ground the student. Focusing on descending melodic contours[3] physically encourages the larynx to lower and the breath mechanism to release, signalling to the neuromuscular system that the period of high demand has ended [6, 7].

Rather than being separate blocks, the warm-ups, vocal athletics and cool-downs form a functional ecosystem within the vocal lesson. This structure ensures that the voice is never subjected to demands without adequate preparation or left in a tense state without release. Together, these three elements define a strategic periodization of vocal load, forming a cyclical process in which each phase supports the others.

[2] A resolving harmony (or resolution) is the move of a musical note or chord from a state of tension or dissonance to a state of rest or consonance.

[3] The melodic contour refers to the overall direction in which a melody rises or falls in pitch over time.

1.2 Vocal qualities and vocal exercises

1.2.1 Vocal qualities

The study of vocal technique encompasses not only breath management and intonation, but crucially the mastery of specific vocal qualities. These qualities are often referred to as 'voice colors', but they are actually specific configurations that produce distinct acoustic results. Pedagogically, these configurations are identified through different parameters. They are often described based on sympathetic resonance[4] sensations (commonly termed chest voice, head voice and mix voice) [8] or defined by their underlying physiological mechanisms, such as the regulation of subglottic pressure or the interaction between the thyroarytenoid (TA) and cricothyroid (CT) muscles[5] [9].

1.2.2 Vocal exercises

Vocal exercises are targeted tasks that are designed to isolate and condition specific laryngeal and vocal tract configurations rather than being mere mechanical drills. These exercises consist of repetitive melodic patterns and are almost always accompanied by the piano, which provides harmonic context and rhythmic stability. Depending on their specific pedagogical objective, these exercises fall into distinct functional groups. They may serve as physiological warm-ups to prepare the voice, as vocal athletics to build strength and coordination, or as cool-downs to aid recovery. They also often function as ear-training tools, refining the singer's pitch accuracy and harmonic awareness.

A complete vocal exercise session is structured according to a systematic logic known as chromatic transposition. First, a specific melodic pattern (the 'motif') is established in a comfortable starting key. Once the student has sung the motif, the teacher changes the key by a semitone. This process

[4]Sympathetic resonance is a harmonic phenomenon in which a passive vibrating body responds to external vibrations.

[5]The thyroarytenoid and cricothyroid muscles are the primary laryngeal muscles that control vocal pitch and register. The TA muscle shortens and thickens the vocal folds. The CT muscle, meanwhile, stretches and thins them.

creates an ascending phase, where the voice is gradually taken into higher frequencies, increasing the tension on the vocal folds and challenging the singer to navigate the register change[6]. Once the upper limit of the exercise has been reached, the direction is reversed and the exercise moves into a descending phase. This downward movement is equally important, as it helps to bring the voice down into the lower register, ensuring a smooth transition and encouraging the singer to maintain stability after changing register.

The range of this transposition is dictated by the singer's tessitura. For instance, a singer with a lower tessitura may start and end a vocal exercise on a lower note than a singer with a higher tessitura (e.g. a bass versus a soprano).

1.3 Sheet music

Sheet music is a symbolic notation system used to record music for faithful reproduction. It evolved from ancient adiastematic notation, which used neumes to indicate melodic contour, to diastematic notation, now standardized as the stave. The stave consists of five parallel lines and four equally-sized spaces, functioning as a coordinate system: the vertical axis represents pitch, while the horizontal axis represents time. The melodic and rhythmic elements of music are noted within it in the form of symbols [10].

1.3.1 Notes and pitch

Notes are the fundamental building blocks of nearly all music. In most European countries, music theory identifies them using the solfège naming convention, which employs the syllables "Do-Re-Mi-Fa-Sol-La-Si". English-speaking countries use the letters A through G to represent the same notes; in this system the note A corresponds to La. In a score, each note is assigned a specific vertical position on the staff, either on a line or in a space, as determined by the clef. If a note exceeds the range of the staff, additional lines called ledger lines are added to accommodate it (see Figure 1) [11].

[6] The change in vocal register (often called 'passaggio') is the moment of transition when the larynx must change its vibration mechanism in order to continue rising or falling in range.

On a piano keyboard (see Figure 2), these notes repeat in a 12-key pattern (an octave) comprising 7 white keys (naturals) and 5 black keys (sharps and flats).

Pitch is associated with the frequency of physical oscillations, measured in hertz (Hz). In Western music, pitches are defined around a central reference of A4, which is currently standardized at 440 Hz. In the 12-tone equal temperament system, the frequency of any note is calculated as:

$$f = 440 \cdot 2^{\frac{n}{12}}$$

where n is the number of semitones from A4.

An octave represents a 2:1 frequency ratio, meaning notes an octave apart share the same name but exist at different pitch levels [12].

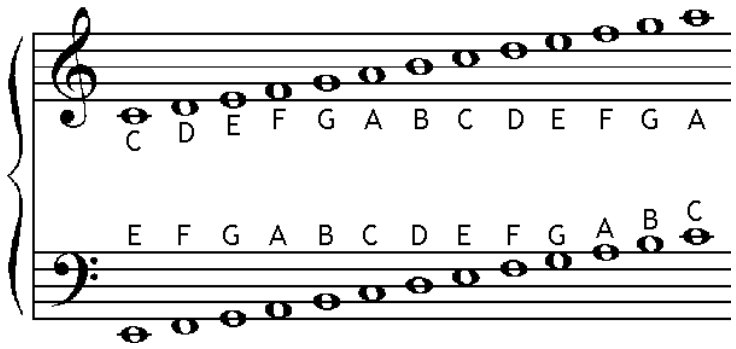


Figure 1: The note names assigned to each line and space of the staff, as determined by the clef symbol at the beginning of the staff [13].

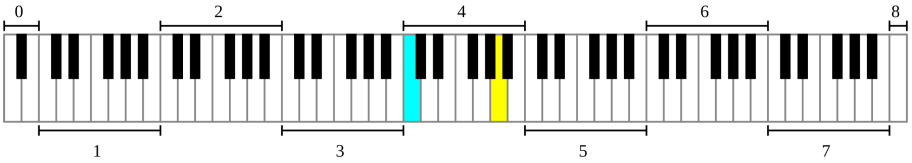


Figure 2: Every octave on a 88-key piano, containing the same sequence of notes but at different pitch levels [12].

1.3.2 Intervals

In music theory, an interval is the difference in pitch between two notes. The smallest interval in a diatonic scale is a semitone, representing the distance between two adjacent keys on a piano. Those smaller than a semitone are called microtones, and are more common in non-Western music.

An interval is defined by two attributes: size and quality. The size of an interval is the number of note names it encompasses, including the two notes that form it. The interval B–D is a 'third' because the notes from B to the D above it encompass three letter names (B, C, and D). The quality of an interval is determined by the number of semitones it contains. For instance, a major third (M3) comprises four semitones, whereas a minor third (m3) comprises three. Intervals can also be augmented or diminished by adding or subtracting a semitone, respectively [14].

1.3.3 Music notation symbols

- **Clef placement**: the absolute pitch of each line is determined by a clef symbol placed at the far left of the stave. For example, a treble clef (also known as a G clef) on the second line identifies that line as the pitch G above middle C[7].
- **Key signature**: located between the clef and the time signature, this arrangement of sharps or flats[8] indicates the tonality of the piece. It shows

[7]Middle C is the 4th C note from the left hand side of a standard 88-key piano.

which notes should consistently be altered (raised or lowered) throughout the music.

- **Time signature**: positioned to the right of the key signature, the time signature indicates the relationship between timing counts and note symbols. The lower numeral shows which note value the signature is counting. The upper numeral shows how many of these note values make up a bar.
- **Bar lines and measures**: vertical bar lines group notes on the stave into measures, providing a clear rhythmic structure.
- **Notes and rests**: these symbols represent the presence or absence of sound; notes indicate pitch through their vertical position and duration through their shape, while rests utilize specific symbols and positions to encode precise intervals of silence.
- **Tuplets**: these symbols indicate irregular subdivisions of the beat, allowing for rhythmic variations such as triplets or quintuplets within standard time signatures.
- **Chords**: when multiple notes are stacked vertically, they indicate that these pitches should be played simultaneously, forming a chord.
- **Dynamics and articulation**: various symbols and markings provide instructions on the volume (dynamics) and manner of playing (articulation) for specific notes or passages, guiding the expressive interpretation of the music [15].

[8] Sharps (♯) and flats (♭) are musical symbols, known as accidentals, that alter a note's pitch by a semitone.

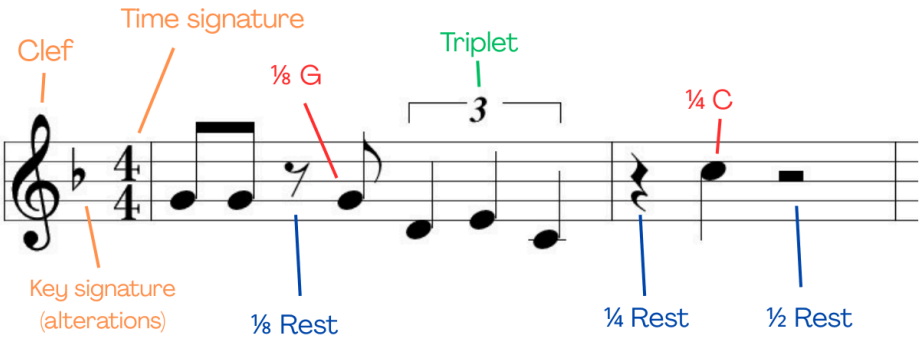


Figure 3: An example of a musical score showing various music notation symbols.

1.4 LL(1) grammars

In the field of musical representation, there are many different types of notation, not just sheet music. Some of these are textual languages that can be interpreted by both humans and machines. From the perspective of formal language theory, each of these systems could be modelled as a formal grammar. To ensure univocal interpretation of symbols, it is possible to assimilate an unambiguous grammar to the properties of an LL(1) grammar, allowing for deterministic syntactic analysis.

1.4.1 Context-free grammars

In formal language theory[9], a context-free grammar G is a quadruple G = (V, Σ, P, S) where:

- V is a finite set of non-terminal symbols
- Σ is a finite set of terminal symbols

[9]Formal language theory is a branch of computer science and mathematics that studies the syntax of languages, defining them as sets of strings over a finite alphabet created by specific grammatical rules.

- P is a finite set of production (or derivation) rules, with rules of the form $V \rightarrow (V \cup \Sigma)*$
- S is the start symbol, with $S \in V$ [16]

Each production rule is of the form $A \rightarrow \alpha$, with A a single nonterminal symbol, and α a string of terminals and/or nonterminals. Regardless of which symbols surround it, the single nonterminal A on the left hand side can always be replaced by α on the right hand side.

A formal grammar is essentially a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements [17].

Example of a derivation in a context-free grammar G = (V, Σ, P, S):

```
- V = {E}
- Σ = {(, ), +, x}
- P = {E → (E), E → E + E, E → x}
- S = E

Derivation of the string (x) + x:
E ⇒ E + E ⇒ (E) + E ⇒ (E) + x ⇒ (x) + x
```

(x) + x is to be considered a string in the language generated by G [16].

Each series of derivations is associated with a derivation tree. For the previous derivation, the associated tree is the one in Figure 4.

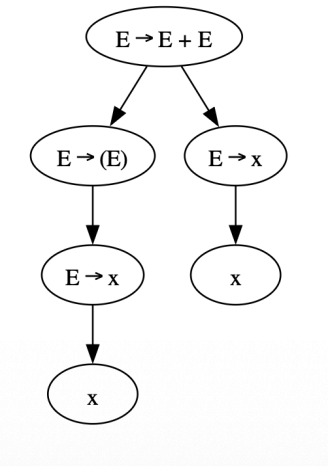


Figure 4: The derivation tree associated with the derivation of the string (x) + x in the context-free grammar G defined above [16].

1.4.2 LL(1)

In theory, we examine the class of context-free grammars that can be parsed in a deterministic and top-down manner with a fixed amount of look-ahead. These grammars are known as LL(k) grammars, where k represents the number of symbols that can be looked ahead to. In the context of compiler construction, the most relevant and widely studied case is k = 1. LL(1) grammars represent a subset of context-free grammars that allow for efficient, unambiguous syntactic analysis. The acronym LL(1) summarizes the operational characteristics of the parser.

- **L**: the first L indicates that the parser processes the input from left to right, meaning it reads the input string sequentially from the beginning to the end.
- **L**: the second L shows that the parser constructs a leftmost derivation of the input string. In a leftmost derivation, the parser always expands the leftmost non-terminal symbol first, ensuring a systematic approach to building the parse tree.

- **(1)**: the (1) denotes that the parser uses a single symbol of look-ahead, without consuming it, to make parsing decisions. This means that at any point in the parsing process, the parser can only look at the next input symbol to determine which production rule to apply.

Formally, a grammar G = (V, Σ, P, S) is defined as LL(1) if, for every pair of distinct productions $A \Longrightarrow \alpha$ and $A \Longrightarrow \beta$, it is possible to distinguish which rule to apply based on the FIRST and FOLLOW sets. The FIRST set of a string α, denoted as FIRST(α), is the set of terminals that can appear at the beginning of any string derived from α. The FOLLOW set of a non-terminal A, denoted as FOLLOW(A), is the set of terminals that can appear immediately to the right of A [17, 18].

1.5 Online vocal teaching

In order to discuss online vocal instruction, we must first ask ourselves if it makes sense to take singing lessons online. Until recently, the music community would have probably responded ‘no’. Historically, online music education was seen as a compromise comprising static resources such as digital sheet music and pre-recorded instructional videos, or asynchronous courses. These tools were considered supplementary support for students outside their physical learning environment and were criticized for lacking the immediate feedback loop essential for skill acquisition.

However, the onset of the pandemic acted as a catalyst, accelerating technological progress in the delivery of vocal training. What started out as a necessity has evolved into a legitimate teaching method. One of the key benefits of this shift is the democratization of access. Digital platforms enable learners to access instruction regardless of their geographical location, eliminating the restrictions imposed by proximity to local teachers. This flexibility enables students to organize their studies around their schedules and progress at a suitable pace, facilitating a balance between artistic development and other commitments. Furthermore, the digital environment offers unique advantages, such as the ability to instantly record lessons for review, integrate

visual aids via screen sharing and foster global interaction among students and educators [19].

1.5.1 Environment and hardware requirements

To ensure effective online vocal lessons, teachers and students must consider their environment and the necessary hardware. A quiet, well-lit space that is free from distractions is essential for clear communication and concentration during lessons. Good acoustics are also important: soft furnishings can help to reduce echo and background noise. A reliable internet connection is also crucial to prevent interruptions and maintain the flow of the lesson. In terms of hardware, a computer or tablet and a high-quality microphone and headphones are necessary for clear audio transmission. To ensure a more faithful, unprocessed signal, it is recommended that external condenser microphones are used in combination with an audio interface. Using headphones instead of speakers also helps to prevent audio feedback and echo, improving clarity for both teacher and student. Platforms that allow users to disable audio enhancements and use 'original sound' modes are particularly valuable for vocal pedagogy, as they preserve the natural acoustic qualities of the voice. A webcam is also important for visual communication, enabling the teacher to observe the student's posture and technique [20].

1.5.2 Latency

Latency is the delay between sound transmission and reception, technically defined as the time interval required for a data packet to travel from the source to the destination and be processed. During online lessons, this delay is cumulative and is introduced at distinct stages of the audio signal chain:

- **A/D conversion and buffering**: the time taken by the audio interface to convert the analog microphone signal into digital data and fill the hardware buffer.
- **DSP and encoding**: the operating system and communication software (e.g., Zoom, Google Meet) process the signal (noise suppression, echo cancellation) and encode it using codecs like Opus.

- **Network transmission**: the travel time of packets over the internet, susceptible to jitter (variability in packet arrival time).
- **D/A conversion**: the decoding and conversion back to analog signal on the receiving device.

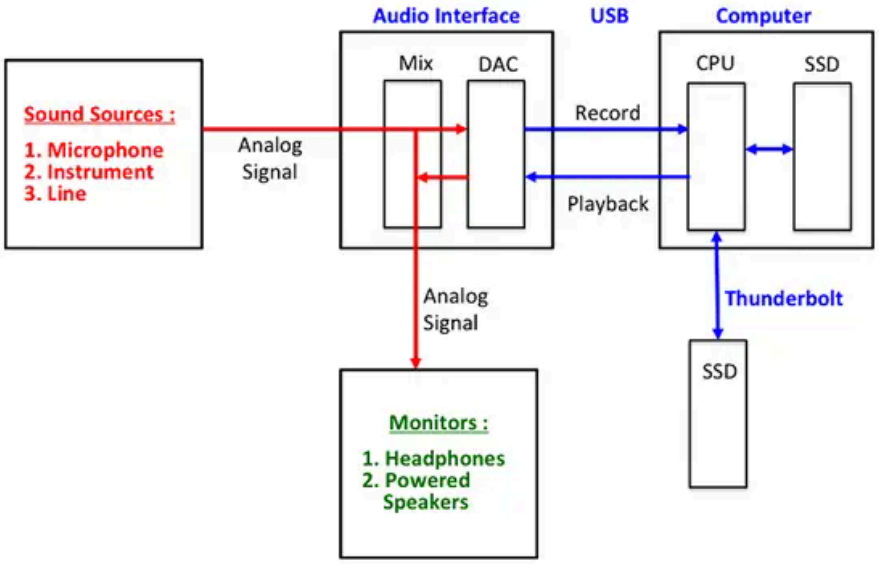


Figure 5: The path of the audio signal in a digital recording setup. Each stage introduces a fraction of latency [21].

Although individual hardware stages may only contribute milliseconds, the aggregate round-trip latency (RTL) over a standard internet connection often ranges from 100 ms to 500 ms. This far exceeds the perceptual threshold for synchronous music-making, which research places at 20–30 ms. Beyond this threshold, the human brain can no longer perceive two sounds as simultaneous, resulting in a 'dragging' beat and the eventual breakdown of rhythmic cohesion [20, 22]. Consequently, unless specialized low-latency software is used, instructors must adopt asynchronous pedagogical approaches. These include turn-taking strategies, where either only one person sings or plays at

a time, or where students sing along to a pre-recorded backing track played directly on their computers.

1.6 Tools and platforms for online voice lessons

Online vocal lessons are affected by issues such as latency and limited real-time interaction, so choosing the right digital tools and platforms is crucial. This section provides an overview of the main existing solutions.

1.6.1 General-purpose platforms

General-purpose video conferencing platforms such as Zoom, Microsoft Teams, and Google Meet are widely used in online vocal instruction thanks to their accessibility, reliability, and ease of use. Messaging applications such as WhatsApp and Telegram are also increasingly being used for online vocal instruction, as they now offer features such as screen sharing, high-quality voice calls and video communication. This makes them suitable for both formal and informal teaching contexts [23]. However, these platforms are primarily designed for spoken communication rather than musical performance. Consequently, they often apply automatic audio processing, including noise suppression, echo cancellation and dynamic compression, which can alter vocal timbre and dynamics. Some platforms offer settings (e.g. ‘original sound’ or ‘music mode’) that allow users to disable or reduce these effects, thereby improving their suitability for vocal pedagogy (see Figure 6).

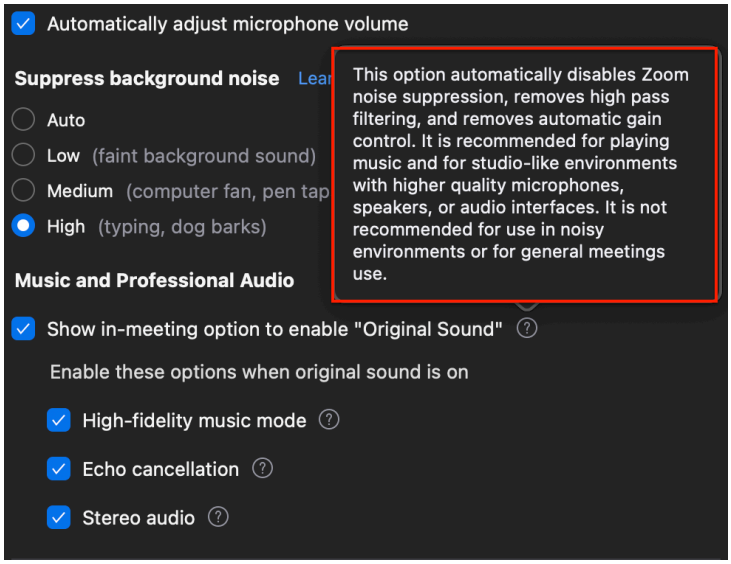


Figure 6: Zoom's 'Original Sound for Musicians' settings, which disable echo cancellation and audio compression [24].

1.6.2 Low-latency music platforms

Low-latency music platforms are specifically designed to enable real-time musical interaction over the internet by minimizing audio delay while preserving sound quality. Examples include Jamulus, JackTrip, Yamaha Syncroom, Soundjack and SonoBus. These platforms are particularly valuable for activities that require precise temporal coordination. Unlike general-purpose video conferencing tools, low-latency platforms prioritize audio transmission over video, often using uncompressed or lightly compressed audio formats to reduce processing delays. However, they typically require a more advanced technical setup, including external audio interfaces, wired internet connections, optimized buffer sizes, and in some cases, network configuration. Despite their higher technical demands, low-latency platforms offer significant pedagogical advantages in contexts where synchronous musical interaction is essential. They are better known and used by musicians for rehearsals and performances because of their timing precision, allowing

for the creation of virtual ensemble concert music (see Figure 7), but they can also be effectively adapted for vocal teaching [25].

Figure 7: The interface of Yamaha Syncroom Ver.2, displaying latency metrics and connection status for ensemble performance [26].

1.6.3 Requirements for an ideal vocal teaching platform

In summary, while these platforms are indispensable for audio-video connections and rehearsals, they are not sufficient for singing lessons alone. General-purpose platforms are often preferable for this type of activity as they support webcams and visual communication, and are generally easier to configure. The limitations of these types of platforms highlight which features would be most beneficial in an online vocal teaching program. One of the main challenges is latency: when exercises are played from the teacher's device, the student hears the accompaniment with a delay, making it difficult to sing in time and receive accurate guidance. An ideal system would address this by enabling students to perform vocal exercises directly on their own device,

with the accompaniment generated or played locally. This architecture effectively eliminates transmission latency, ensuring real-time synchronization between voice and instrument, and facilitating immediate corrective feedback from the teacher.

Furthermore, the system should enable teachers to prepare exercises in advance and organize them into a structured library for streamlined access. Generating these melodic patterns with the help of software would be significantly more efficient than the traditional approach of performing and recording each vocalise manually. Finally, the ability to export audio and sheet music would empower students to use these resources for autonomous warm-ups and practice outside of lessons.

2 Technological stack

Unlike a static website designed primarily for viewing content, an online vocal warm-up system has to be developed as a web application. While traditional websites simply provide information, a web application allows users to interact with and manipulate the data they see [27]. In the context of vocal pedagogy, this means that users can actively play and edit musical exercises in their browser, rather than just viewing a score.

A web-based system for interactive vocal exercises requires a strategic combination of frontend and backend technologies. In this setup, the frontend operates within the user's browser, while the backend resides on the server. This separation allows the frontend to handle real-time tasks such as rendering musical notation and providing immediate auditory feedback, while the backend manages broader system logic, including data persistence, user authentication, and resource management.

At the heart of the system's logic, musical data is represented in symbolic notation. This abstraction is essential, as it enables complex musical structures to be rendered across various devices. In this model, the backend serves as support infrastructure for content delivery and storage, while core musical execution is offloaded to the user's device. The following sections will detail the specific web-related technologies, frameworks and tools that comprise this technological stack.

2.1 Web development technologies

The foundation of any web application rests on a set of core technologies that define how content is structured, styled, and programmed. These technologies are standardized by the World Wide Web Consortium (W3C)[10] and are

[10]The World Wide Web Consortium (W3C) is an international organization founded in 1994, that defines open standards to ensure the web's interoperability and development.

natively supported by all modern web browsers [28]. The following subsections provide a brief overview of these fundamental building blocks.

2.1.1 HTML

HyperText Markup Language, or HTML, serves as the structural backbone of all web content. It is a declarative markup language, meaning it is used to describe the nature and structure of content rather than to execute logic. HTML utilizes a system of elements enclosed in tags to demarcate different parts of a document.

When a web browser loads an HTML file, it parses the markup to construct the Document Object Model (DOM). The DOM is a tree-like memory representation of the document structure, where every HTML element becomes a node. This hierarchy is fundamental to the operation of dynamic web pages, as it provides the programmatic interface that allows other technologies, specifically JavaScript, to access and modify elements within the document after the initial page load [29].

Contemporary web development utilizes HTML5, the current industry standard which significantly extended the language's capabilities to support native multimedia and complex applications without reliance on third-party plugins.

An HTML5 page is built around a hierarchical structure of elements, starting with the `<!DOCTYPE html>` declaration (which specifies that the document is an HTML5 document) and the `<html>` root element, which contains the `<head>` (for metadata and links to external resources) and the `<body>` (for the actual visible content) [30, 31]. This structure is briefly illustrated in the following code snippet:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Vocal Warm-up</title>
```

```
</head>
<body>
    <h1>Exercise 1</h1>
    <div id="notation"></div>
</body>
</html>
```

2.1.2 CSS

While HTML defines the structure of a document, Cascading Style Sheets (CSS) is responsible for its visual appearance. CSS functions as a rule-based language that describes how HTML elements should be rendered. The primary design philosophy behind CSS is separation of concerns, which dictates that a document's visual design should be distinct from its content. This separation enables developers to modify a website's aesthetic properties without altering the underlying HTML structure.

The 'cascading' in the name refers to the priority system that determines which style rule applies when multiple rules match a particular element. This hierarchy enables a modular approach, whereby generic styles are inherited from parent elements, which can then be overridden by more specific styles. Furthermore, CSS enables the implementation of Responsive Web Design (RWD); the stylesheet can detect the characteristics of the user's device, such as screen width, resolution and orientation, and apply different styling rules accordingly.

CSS is usually specified in a separate .css file and linked to the HTML document via a `<link>` tag. Alternatively, it can be embedded directly within the HTML using `<style>` tags [32, 33].

The example below shows a CSS rule set, comprising a class (`.container`) and a type selector (`h1`):

```
.container {
    background-color: #ffffff;
    padding: 20px;
```

```
    border-radius: 8px;
}

h1 {
    color: #333;
    font-family: sans-serif;
}
```

2.1.3 JavaScript

JavaScript functions as the core programming language of the web, transforming static documents into dynamic applications. Unlike HTML and CSS, which are declarative, JavaScript introduces imperative programming capabilities, enabling the implementation of complex algorithms directly within the client's browser.

Its primary role is to manage the application's behavior by handling user events, such as mouse clicks, key presses, or form submissions, in real-time. This allows for the creation of Single-Page Applications (SPAs), where the page content is updated dynamically without requiring a full reload from the server. JavaScript interacts with the DOM to programmatically update the content and style of the page in response to user actions [34].

The following example shows a JavaScript function that selects an HTML element by its ID and updates its content:

```
function updateNotation(content) {
    const element = document.getElementById('notation');
    if (element) {
        element.textContent = content;
    }
}
```

Unlike compiled languages, where the source code is translated into machine-readable binary formats prior to execution, JavaScript is typically delivered to the client as plain text. The script is transmitted to the browser alongside other

structural and stylistic assets, such as HTML markup and CSS stylesheets. The browser interprets this source code exactly as it was written: a human-readable sequence of Unicode characters, analyzed linearly from left to right and top to bottom.

Upon receiving a script, the JavaScript engine first initiates a process known as lexical analysis. This phase involves scanning the long string of characters that comprises the code and converting it into a stream of discrete input elements. These elements are categorized into five distinct types: tokens (the meaningful identifiers and keywords), format control characters, line terminators, comments, and whitespace. This tokenization process is the prerequisite for the browser to construct an Abstract Syntax Tree (AST)[11] and eventually execute the logic.

It is also important to note the ephemeral nature of the execution environment. The results of a script execution are not automatically preserved when the user reloads the page or navigates away. Unless the developer includes explicit instructions to persist data, the application state resets with every new page load [35].

2.1.4 JSON

JavaScript Object Notation (JSON) functions as the ubiquitous standard for data interchange in modern web applications. Although its syntax is derived from JavaScript object literals[12], JSON is a language-independent text format that is now supported by virtually every programming environment.

Structurally, JSON is built on two universal data structures: a collection of name/value pairs (realized as an object, record, or dictionary) and an ordered list of values (realized as an array or vector). It supports primitive data types such as strings, numbers, booleans, and null, which can be nested to create complex data models [36].

[11] An AST is a data structure used by compilers and linters to represent the hierarchical relationship of program elements.

[12] An object literal is a way to define a single object and its properties in one go using a comma-separated list of name-value pairs enclosed in curly braces

When a client-side application requests data, the server serializes this information into a JSON-formatted string. Upon receipt, the browser's JavaScript engine utilizes the native `JSON.parse()` method to deserialize the string into a valid JavaScript object. This seamless integration allows developers to manipulate external data as native variables immediately after retrieval. This process can be reversed; using `JSON.stringify()`, a JavaScript object can be converted back into a JSON string [37].

The following example shows a JSON object representing a musical exercise:

```
{
  "exercise": {
    "id": 101,
    "title": "C Major Scale",
    "tempo": 120,
    "notes": ["C4", "D4", "E4", "F4", "G4", "A4", "B4",
"C5"]
  }
}
```

2.2 Frontend

Before analyzing the frameworks used to manage and customize the web application, it is essential to focus on the system's core: the musical scores. This section first examines the specific technologies adopted for music notation and its rendering within a web ecosystem. It then looks at how these components are integrated into a cohesive web interface.

2.2.1 abc notation

The abc notation is a symbolic language designed to represent music in a plain text format, utilizing a combination of Helmholtz pitch notation[13] and ASCII characters. Introduced at the end of 1993 by Chris Walshaw, it was

[13] This musical naming system, named after Hermann von Helmholtz and born in 1863, distinguishes the same pitch class in different octaves by using a combination of uppercase and lowercase letters, along with sub- and super-prime symbols.

developed primarily to notate folk and traditional tunes of Western European origin, such as English, Irish, and Scottish music, which typically require only a single stave in standard classical notation. However, despite its origins, the format has proven remarkably versatile and has been used to encode complex classical works such as Beethoven's Symphony No. 7[14].

Unlike many computer-based musical languages, abc notation prioritizes human readability. Its ASCII structure allows for immediate performance from plain text, eliminating the dependency on graphical processing. Consequently, it serves as a highly efficient interchange format for web architecture, significantly reducing overhead in data storage and transfer.

Furthermore, abc notation has gained prominence in academic research. In recent years, this has led to its adoption in automated music composition and generation systems, particularly those that use machine learning techniques. One notable example of this application is the folkrnn.org project [38], which uses abc notation for neural network–based music generation [39].

abc notation syntax

The notation relies on a strict syntax that distinguishes between metadata and musical content. An abc tune is formally structured into two primary blocks: the tune header and the tune body, and is terminated by an empty line or the end of the file. This separation enables parsers to pre-configure the musical environment by establishing rules for pitch and rhythm prior to processing the individual symbols representing the melody.

- **Tune header**: this section consists of a series of information fields that provide the metadata for the tune. Each field begins with a single uppercase letter followed by a colon, which identifies the type of information contained in the line. Among the most commonly used fields are:
  - `X:` (Reference number): a unique identifier for the tune within a collection.

[14] A complete abc transcription of the Symphony No. 7, Movement 2 by Steve Allen is available in the official abc notation archives: https://www.ucolick.org/~sla/abcmusic/sym7mov2.html.

- `T:` (Title): the name of the tune.
- `M:` (Meter): the time signature, indicating the rhythmic structure.
- `L:` (Default note length): the base duration for notes in the tune.
- `K:` (Key): the key signature, defining the tonal center and accidentals.

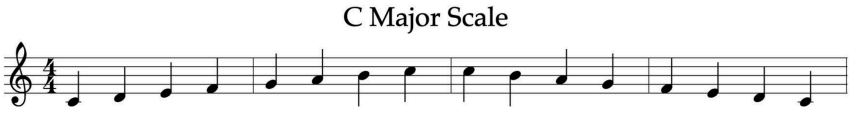
- **Tune body**: immediately following the header, the tune body contains the music code. Formally, music code is defined as any line that is not an information field, stylesheet directive, or comment.
  - **Notes and duration**: pitch is represented by the letters A through G. The duration of each note is calculated relative to the default unit length defined in the header (L:). To modify this duration, a multiplier (e.g. C2) or a divisor (e.g., C/2) is added to the note letter.
  - **Accidentals**: chromatic alterations are indicated by symbols placed immediately before the note they modify: `^` for sharps, `_` for flats, and `=` for naturals.
  - **Octaves**: uppercase letters represent notes in the fourth octave, while lowercase letters indicate notes in the fifth octave. Additional octave shifts can be indicated by appending commas (for lower octaves) or apostrophes (for higher octaves) after the note letter.
  - **Rests**: denoted by the letter `z`, with duration specified similarly to notes.
  - **Bar lines**: vertical lines (`|`) are used to separate measures.
  - **Chords**: multiple notes played simultaneously can be grouped using square brackets (`[ ]`).
  - **Tuplets**: irregular rhythmic groupings are handled via a specific prefix syntax. A triplet is denoted by `(3` followed by the three notes involved (e.g. `(3CDE)`), instructing the renderer to fit three notes into the time normally occupied by two.

Syntax examples

To illustrate how these syntactic elements combine to form a readable score, the following examples demonstrate the direct mapping between the ASCII characters and the standard staff notation.

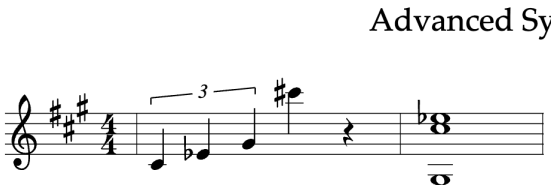
The first example (Figure 8) shows a simple C major scale, written entirely with 1/4 notes.

The second example (Figure 9) shows more advanced syntax features, including accidentals, octave shifts, chords and tuplets.




```
X:1
T:C Major Scale
M:4/4
L:1/4
K:C
|C D E F | G A B c | c B A G | F E D C |
```

Figure 8: The musical notation (top) and the corresponding abc notation (bottom).




```
X:2
T:Advanced Syntax
M:4/4
L:1/4
K:A
|(3C_EG ^c' z|[c_eg,,]4 |
```

Figure 9: The musical notation (top) and the corresponding abc notation (bottom).

2.2.2 abcjs

The abcjs library is an open-source JavaScript rendering engine for interpreting tunes in abc format, available under the MIT license. It was developed by Gregory Dyke and Paul Rosen between 2009 and 2018. As it renders sheet music entirely in the browser, the library is intended for web designers and programmers who wish to display standard music notation on websites. Its purposes include modifying music, creating synthesized audio, styling music sheets using CSS and adding animated effects to drawn music. It should be noted that abcjs does not have a visual score editor, and that every operation must be performed by writing strings in abc notation [40].

renderAbc function

The main entry point for creating standard music notation is `abcjs.renderAbc`. This allows an arbitrary JavaScript string to be converted into an SVG image of sheet music. The full definition of the `renderAbc` call is:

```
var tuneObjectArray = renderAbc(elementArray, abcString,
options);
```

The `renderAbc` function relies on three key parameters that control how music is processed and displayed:

- `elementArray` identifies the target for rendering. It can accept a string ID, a direct HTML element (usually a 'div'), an array combining these items, or an asterisk for invisible rendering[15].
- `abcString` contains musical data in standard abc format and can represent a single tune, multiple tunes or a fragment.
- `options` is an object that allows developers to configure a wide variety of settings to customize the output. These options include various layout and visual styling settings.

visualObjs

The `renderAbc` function returns an array of objects that each represent a tune. This data is the bridge to audio and animation; it must be passed to the `CreateSynth` or `TimingCallbacks` constructors to enable audio playback or coordinate timing events.

Each object contains a property called `visualObj`, which is an array of objects representing the individual elements of the rendered music sheet. The data in each object contains an array of all the music, called `lines`, in which every item is a 'staff system'. This could be one staff for music written for a single instrument, two staves for music written for the piano, and so on. In addition to the `lines` property, the `visualObj` contains various other properties such as the version of abcjs used, text items not associated with music the formatting options applied to the tune.

Regarding the methods available for the `visualObj`, most of them serve to retrieve information on rhythmic and metric data. For example,

[15]The abcjs library includes an invisible rendering mode in which audio is generated without any visual output.

the `getBarLength` method returns the duration of a measure, while the `getBeatsPerMeasure` method returns the number of beats in a measure.

Synthesized sound

The synthesized sound creates an audio buffer similar to a WAV file, which contains the entire piece ready to play. The sounds themselves come from a 'sound font'; each individual note on each instrument is a separate sound file, which is then combined into the audio buffer. An internet connection is required for the sound fonts, the default version of which is available on a GitHub page[16]. However, users can supply their own sound fonts and deliver them locally without a network connection.

All implementations of audio playback need a `CreateSynth`.

```
var synth = new ABCJS.synth.CreateSynth();
```

The process begins with the initialization of the `CreateSynth` object, which triggers the retrieval of all the necessary sound fonts. This operation returns a `Promise` that resolves once loading is complete, providing a detailed report that distinguishes between newly loaded notes, previously cached notes and notes that haven't been loaded, along with the corresponding error. While initial data retrieval may be subject to network latency, the library caches these assets locally. This means that subsequent requests involving similar musical content are processed with significantly reduced delay.

TimingCallbacks

The `TimingCallbacks` class allows developers to set up functions that are called at various intervals. This is useful for synchronizing visual effects with the music, such as highlighting notes as they are played. To use `TimingCallbacks`, an instance of the class must be created and the `visualObj` must be passed to it, along with a `params` object that defines the properties of the callbacks.

[16]The default sound fonts can be found at https://github.com/paulrosen/midi-js-soundfonts.

```
var timingCallbacks = new abcjs.TimingCallbacks(visualObj,
params);
```

Among the properties that can be defined in the `params` object are:

- `qpm`: the number of beats per minute, which controls the tempo of the music.
- `eventCallback`: a function that is called at the start of each event, either a note, a rest or a group of notes, and that takes `ev`, the event object, as a parameter [41].

2.2.3 Vue.js

Vue serves as a progressive JavaScript framework designed for efficient interface development. Its architecture rests on standard web technologies, offering a model that decouples the view from the direct DOM manipulation.

The framework is built around two core features that distinguish it from traditional DOM manipulation libraries:

- **Declarative Rendering**: rather than manually selecting elements and modifying their text content (imperative programming), Vue extends HTML with a template syntax. This allows developers to describe how the UI should look based on the current state, and the framework handles the rendering.
- **Reactivity**: when the application state changes, Vue detects the modification and automatically patches the DOM to reflect the new reality.

```
import { createApp, ref } from 'vue'

createApp({
  setup() {
    return {
      count: ref(0)
    }
  }
}).mount('#app')
```

```
<div id="app">
  <button @click="count++">
    Count is: {{ count }}
  </button>
</div>
```

This example shows a simple Vue application that consists of a button and a counter. The `ref` function is used to create a reactive reference to the `count` variable, which is initialized to 0. The `@click` directive binds a click event to the button, which increments the `count` variable each time the button is clicked. The current value of `count` is displayed within the button using interpolation syntax (`{{ count }}`), allowing the user to see the updated count in real time as they interact with the button.

Single-file components

In Vue projects, single-file components can be used. These are files with a .vue extension that encapsulate a component's logic (JavaScript), template (HTML) and style (CSS) in a single file.

Here is the same example as above, but using a single-file component:

```
<script setup>
import { ref } from 'vue'
const count = ref(0)
</script>

<template>
  <button @click="count++">Count is: {{ count }}</button>
</template>

<style scoped>
</style>
```

API styles

Vue provides developers with two primary paradigms for structuring component logic. With **Options API**, logic is compartmentalized into specific properties such as `data` (state), `methods` (behavior), and lifecycle hooks (e.g., mounted). Inside these options, the component instance is accessible via the 'this' context, which serves as the central hub for accessing reactive state and functions:

```
<script>
export default {
  // Properties returned from data() become reactive state
  // and will be exposed on `this`.
  data() {
    return {
      count: 0
    }
  },
  // Methods are functions that mutate state and trigger
updates.
  methods: {
    increment() {
      this.count++
    }
  },
  // This function will be called when the component is
mounted.
  mounted() {
    console.log("The initial count is ${this.count}.")
  }
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

Alternatively, the **Composition API** adopts a function-based approach. Instead of declaring options, logic is constructed using imported API functions. When working within single-file components, this style is most effectively implemented using the `<script setup>` block.

The 'setup' attribute acts as a compile-time macro. It instructs Vue to process the script using specific transforms that minimize boilerplate code. A significant advantage of this syntax is that top-level bindings, including variables, function declarations and imports, are automatically made available to the template, eliminating the need for them to be returned manually.

Below is the previous component refactored using the Composition API and `<script setup>`:

```
<script setup>
import { ref, onMounted } from 'vue'

// reactive state
const count = ref(0)

// functions that mutate state and trigger updates
function increment() {
  count.value++
}

// lifecycle hooks
onMounted(() => {
  console.log(`The initial count is ${count.value}.`)
})
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

2.2.4 Bulma

Bulma is a free, open-source CSS framework that is entirely based on Flexbox, a layout model that automatically adjusts the width of elements based on the container. However, using Flexbox manually requires writing complex CSS code. Bulma eliminates this complexity, enabling developers to manage layout and responsive behavior with predefined HTML classes alone. Unlike other popular frameworks, such as Bootstrap, Bulma is CSS-only and does not include any JavaScript files [42].

Bulma is also built using Sass (Syntactically Awesome StyleSheets), which is an extension of the CSS language.Sass enables the use of variables and functions, as well as the organisation of style sheets into multiple files. Bulma uses these Sass variables to define colors, sizes, spacing, and other aspects of the framework. The framework is composed of approximately 40 Sass files, each of which handles a specific component or feature.

Bulma's philosophy is based on class readability. Styling is achieved by adding modifier classes to basic HTML elements. These classes typically adhere to the convention of using "is-" for state or style and "has-" for content or characteristics. Because Bulma's syntax is based on HTML classes, the simplest method for styling a page element is adding modifier classes to the element's HTML tag:

```
<button class="button is-primary is-small" disabled>Button</
button>
<button class="button is-info is-loading">Button</button>
<button class="button is-danger is-outlined is-
large">Button</button>
```

The result is shown in Figure 10.

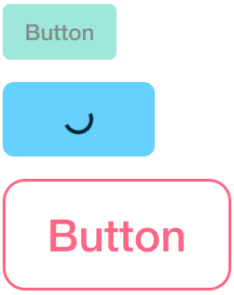


Figure 10: Three buttons styled with Bulma, each with different classes to demonstrate various available styles [43].

CSS Variables

All Bulma components are styled using CSS Variables, which are custom properties that can be defined in CSS and accessed throughout the stylesheet. This allows developers to easily customize the appearance of components by overriding these variables in their own stylesheets. For instance, one way to change the primary colour used in Bulma is to define a new value for the `--primary` variable:

```
:root {
  --primary: #ff5733; /* New primary color */
}
```

CSS Variables defined at a more specific scope will override the ones defined at a more global scope. All Bulma CSS variables are prefixed with 'bulma-', but this prefix can be changed.

Themes

In Bulma, a theme is a collection of CSS Variables. Bulma comes with two themes: `light.scss` (required) and `dark.scss` (optional), both located at `/sass/themes/`. An automated dark mode is included in the framework, and is activated according to the user's system preferences. A custom theme can

be applied by using the `data-theme` attribute on an HTML element or by applying a class to it [43].

2.3 Backend

This section describes the core technologies used to build the server-side logic and the deployment infrastructure.

2.3.1 SQL and SQLite

Structured Query Language (SQL) is the standard domain-specific language used for managing and manipulating relational databases. It allows developers to define data structures, insert records, and perform complex queries to retrieve specific information.

SQL commands fall into two main functional categories that operate at different levels of the database.

The Data Definition Language (DDL) is used to define the structure of the database that will hold the data. It uses commands such as CREATE, ALTER and DROP to create, modify or delete entire databases and their constituent objects (such as tables, indexes and constraints), thereby influencing the relational schema directly.

The Data Manipulation Language (DML) provides the necessary tools to manage data within the previously created structures. It is used to insert, retrieve, update or remove information, thereby interacting with the database's dynamic content [44].

The following SQL snippet demonstrates the creation of a table for storing user data:

```
CREATE TABLE user (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL
);
```

Elements can be retrieved using the `SELECT` statement, which allows for filtering data based on specific criteria.

```
SELECT title, bpm
FROM exercises
WHERE user_id = 1
ORDER BY title ASC;
```

SQLite is a C library that implements a complete and fast SQL database engine. Unlike traditional systems, it operates in a serverless mode, reading and writing data directly to standard disk files without the need for a separate server process. An entire relational database, including tables, indexes, triggers, and views, is contained within a single file. This architecture eliminates the network communication latency typical of the client–server model, making SQLite the ideal solution for environments where simplicity and speed of data access are priorities [45].

2.3.2 Cloudflare Workers

Cloudflare Workers is a serverless platform that allows developers to deploy code across Cloudflare's edge network. Unlike traditional serverless architectures that rely on virtual machines or containers, Workers are built on V8 Isolates [46]. Google V8 is a high-performance engine that implements ECMAScript[17] and WebAssembly[18]. V8 translates JavaScript source code directly into native machine code rather than using an interpreter. This process, known as Just-In-Time (JIT) compilation, allows computers to understand and execute the code rapidly. V8 provides all data types, operators, objects, and functions specified in the ECMA standard. In the context of a browser, the host application (e.g., Google Chrome) provides the DOM, while V8 handles the logic and computation [47].

[17] ECMAScript is a standard for scripting languages (such as JavaScript), intended to ensure the interoperability of web pages across different web browsers.

[18] WebAssembly is a low-level programming language with a compact binary format. It provides other programming languages with a compilation target, enabling them to run on the web.

The key feature that sets Cloudflare Workers apart is its use of isolates instead of containerized processes. An isolate is a lightweight sandbox for code execution. A single runtime instance can host hundreds or thousands of isolates and switch between them seamlessly. Despite sharing a runtime, each isolate's memory is completely separate. This ensures that code is secure and protected from other untrusted or user-written code running on the same machine. Traditional serverless providers spin up a container and a language runtime for each function. Cloudflare only pays for the JavaScript runtime once (see Figure 11) [46].

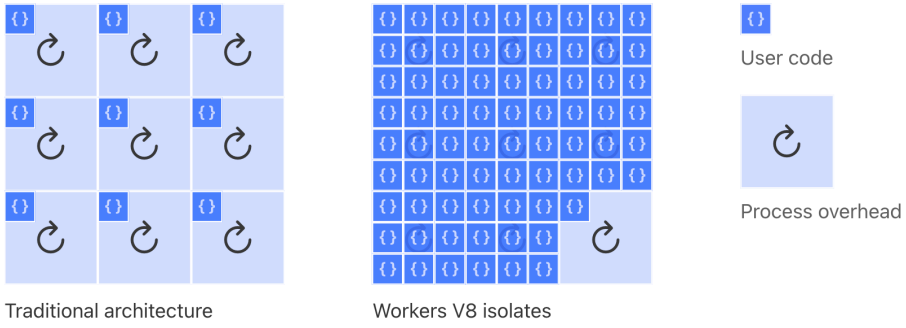


Figure 11: A comparison between the architecture of traditional serverless containers and Cloudflare Workers' isolates [46].

Wrangler

Wrangler is the official command-line interface (CLI) for Cloudflare Workers. It allows developers to manage the entire lifecycle of their serverless applications, from initialization and local development to deployment. Wrangler automates the packaging of code and assets, handles the configuration of environment variables, and manages the creation of necessary resources like KV namespaces or D1 databases. By providing a local development server that mimics the production environment, it enables rapid testing and debugging before pushing code to the global network.

```
# Initialize a new project
npx wrangler init my-worker

# Start a local development server
npx wrangler dev

# Deploy to the Cloudflare global network
npx wrangler deploy
```

D1 Database

D1 is Cloudflare's native serverless relational database, built on top of SQLite. Unlike traditional databases, which require complex connection pooling or manual scaling, D1 is fully managed and scales automatically. D1 is particularly well-suited for applications that require relational data consistency. Developers interact with D1 using standard SQL queries through the Wrangler CLI or the Workers API [48].

```
# Create a new D1 database
npx wrangler d1 create my-database

# Execute a SQL file against the database
npx wrangler d1 execute my-database --file=./schema.sql

# Query the database from a Worker
const { results } = await env.DB.prepare(
  "SELECT * FROM exercises WHERE id = ?"
).bind(id).all();
```

2.3.3 Hono

Hono is a framework designed to help developers create web applications more quickly using Cloudflare Workers. Applications can be developed and published using Wrangler, Cloudflare Workers' command-line interface.

Routers

The most important feature in Hono is its routers. In web development, routing is the backbone of navigation. It is the mechanism that enables web applications to respond to user requests by guiding them to the appropriate content or page based on the URL path. Routing plays a crucial role in defining the user experience, by mapping URL paths to specific content [49].

In Hono there are five routers:

- `RegExpRouter`: Hono's RegExpRouter compiles all the registered routes into 'one large regular expression'. Instead of looping through routes A, then B, then C, it executes that single compiled RegExp once.
- `TrieRouter`: a router based on a Trie (prefix tree) data structure.
- `SmartRouter`: a router that automatically selects the most efficient routing strategy (RegExpRouter or TrieRouter) based on the patterns of the registered routes.
- `LinearRouter`: a simple router that adds the route without compiling strings, suitable for environments that initialize with every request.
- `PatternRouter`: the smallest router in Hono, designed for minimal overhead and basic pattern matching.

Web Standards

Hono is built on top of Web Standard APIs, such as `Fetch`, `Request`, `Response`, and `URL`. This design choice ensures that Hono is not tied to a specific runtime like Node.js, but can run on any platform that supports these standards.

Below is an example of a server that returns 'Hello World'.

```
export default {
  async fetch() {
    return new Response('Hello World')
  },
}
```

Middleware

Middleware functions are a fundamental part of Hono's architecture, allowing developers to execute code before or after a request reaches its final handler. Middleware can be used for a variety of tasks, such as verifying user credentials before allowing access to protected routes.

Hono provides a set of built-in middleware, but also allows for the creation of custom middleware to meet specific application requirements.

```
import { basicAuth } from 'hono/basic-auth'

app.use(
  '/admin/*',
  basicAuth({
    username: 'admin',
    password: 'secret',
  })
)

app.get('/admin', (c) => {
  return c.text('You are authorized!')
})
```

Routing

Hono supports all standard HTTP methods, including GET, POST, PUT, and DELETE. Developers can define routes using these methods to handle different types of requests:

```
app.get('/api/exercises', (c) => c.json({ message: 'List of
exercises' }))
app.post('/api/exercises', (c) => c.text('Exercise created',
201))
app.put('/api/exercises/:id', (c) => c.text(`Update exercise
${c.req.param('id')}`))
app.delete('/api/exercises/:id', (c) => c.text(`Delete
exercise ${c.req.param('id')}`))
```

Context

The Context object `c` is instantiated for each request and kept until the response is returned. It contains the Request object and provides methods for returning responses, such as `c.text()`, `c.json()`, or `c.html()`. It also serves as a store for sharing data between middleware and handlers.

Among the most used properties and methods of the Context object are:

- **`c.req`**: an instance of HonoRequest. It provides access to the incoming request, including headers, query parameters, and body.
- **`c.env`**: in Cloudflare Workers environments, all elements such as KV namespaces, D1 databases, or R2 buckets that are bound to a worker are known as bindings. Regardless of their type, these bindings are always available as global variables and can be accessed via the context `c.env.BINDING_KEY`.
- **`c.text()`**: returns a plain text response.
- **`c.json()`**: returns a JSON response with the appropriate `Content-Type` header.
- **`c.get()` / `c.set()`**: used to pass data between middleware and the route handler. The values of `c.set / c.get` are retained only within the same request. They cannot be shared or persisted across different requests.

The HonoRequest object `c.req` also provides several methods to extract data from the request:

- **`c.req.param()`**: retrieves path parameters (e.g., `:id` in the URL).
- **`c.req.query()`**: retrieves query string parameters.
- **`c.req.header()`**: retrieves request headers.
- **`c.req.json()`**: parses and returns the JSON body of the request [50].

```
app.get('/user/:id', (c) => {
  const id = c.req.param('id')
  const page = c.req.query('page')
  return c.json({ userId: id, page: page })
})
```

3 Analysis and design process

The conception of VoiceWorm stems from the limitations of online teaching discussed in Chapter 1. As mentioned, the main obstacle to online vocal lessons is the lack of real-time synchronization between teachers and students. VoiceWorm addresses this issue by providing a platform where exercises are preloaded and executed locally on the student's device.

The core idea behind VoiceWorm is to shift the generation of the musical accompaniment from the teacher to the student. Rather than transmitting a piano audio signal over the internet, which results in delay and poor audio quality, VoiceWorm transmits the musical code to the student's browser, which synthesizes the sound. The student hears the accompaniment in real time and can sing along in perfect synchronization while the teacher observes and gives feedback to the student via video call.

With this objective in mind, the application was developed as a web-based tool that allows teachers to create, store and search vocal exercises, which then can be shared with students during live sessions or assigned for independent practice.

3.1 The core idea

At the heart of the application is an automation engine designed to mimic the structure of a vocal exercise. Exercises consist of short melodic patterns that are repeated multiple times and transposed by specific intervals to evenly cover certain vocal ranges. VoiceWorm automates this process through its transposition system. Unlike static recordings, which force every singer to replicate the same notes regardless of their physiology, VoiceWorm allows users to define their execution range by setting a starting note and limits for the lowest and highest notes. This flexibility is essential because a vocalization suitable for a bass would be physically impossible for a soprano.

Furthermore, the system is designed to manage the direction of transpositions. A complete exercise traverses the voice in two directions: an ascending phase, in which the voice gradually moves to higher frequencies, and a descending phase, which focuses on bringing the voice down to smooth out transitions. The VoiceWorm engine automatically manages these reversals, ensuring that the pattern never exceeds the boundaries set for the student's voice. Teachers can manually adjust the pitch or playback speed (BPM) until they reach a satisfying range and intensity. The ability to set the number of transpositions and tempo precisely enables the application to adapt to the singer rather than forcing the singer to adapt to the tool.

Vocal exercises are bound by their objective. For this reason, VoiceWorm's core idea is to provide a structured way to categorize and retrieve exercises using a multidimensional tagging system. This system categorizes exercises based on their pedagogical function within a lesson. Exercises can be identified by their musical content, such as specific scales or intervals. The system also incorporates tags based on vocal qualities derived from models like Estill Voice Training[19]. Teachers can quickly locate exercises by filtering for these specific qualities.

3.1.1 Why yet another warm-up application?

Several digital tools currently exist to support vocal training, including the mobile application Warm Me Up [51], the browser-based exercises provided by Virtually Vocal [52], and the interactive tools on ToneGym [53]. While these platforms offer an accessible entry point for vocal practice, they suffer from significant limitations in terms of flexibility and pedagogical depth.

The fundamental constraint of these applications is their reliance on pre-recorded audio files. Because the accompaniment is a static recording, the exercises are not dynamically adaptable. Users are locked into the specific keys and durations that were originally recorded. This rigidity also applies to the usage of the Fach system (the classical classification of voices into

[19]Estill Voice Training is a program established in 1988 by Jo Estill to help singers understand the physiological mechanisms of voice production.

types like Soprano, Alto, Tenor, Bass) as the sole determinant for an exercise's range. This approach overlooks the reality of individual vocal physiology; a singer may limit their range due to fatigue and illness, or they may simply prefer to practice within a more restricted or expanded range on a given day. In pre-recorded systems, granular customization is impossible because the audio cannot be regenerated on the fly to fit new parameters. The overall categorization system in these applications is often reductive. They typically organize content based on generic difficulty levels ('Beginner', 'Intermediate', 'Advanced') or a simple binary distinction between male and female voices.

Finally, a critical omission in most current tools is the absence of visual notation. Although applications like ToneGym have useful features, such as a visual piano keyboard that lights up in real time (see Figure 12), they do not display the actual musical score. Without sheet music, teachers cannot easily follow the exercise progression without listening to the entirety of the audio. VoiceWorm fills these gaps by using client-side synthesis instead of recordings and by allowing users to experiment with transpositions. It also integrates a much more complete tagging system.

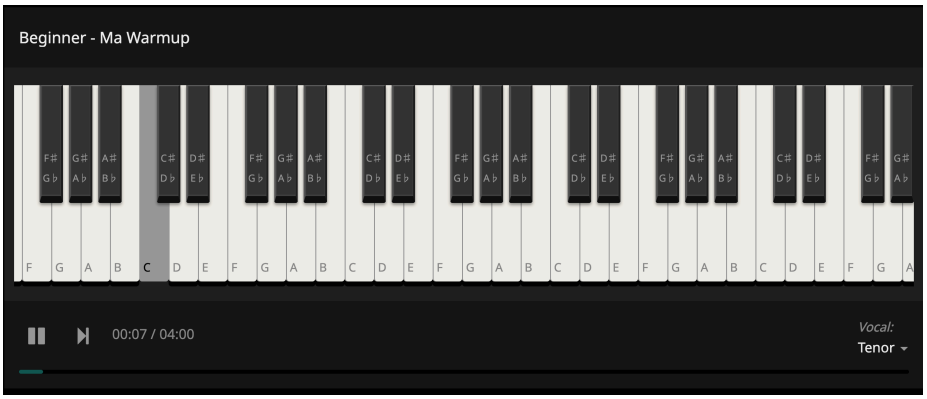


Figure 12: The interface of ToneGym's Vocal Warmup Tool [53].

3.2 Design process

3.2.1 Functional requirements

The goal is to develop a web-based application that allows users to optimize the execution of vocal warm-ups during online lessons and individual study. The typical scenario involves a singing teacher who needs to provide specific exercises to a student, overcoming the latency issues inherent in remote connections. The ideal users of the system are voice professionals, teachers, and students who need to manage their vocal training routine effectively. The application must allow the insertion, visualization, customization, and storage of vocal exercises. Furthermore, it must allow for the consultation of a personal library of organized warm-ups.

User Account Registration

To access the personalized features of the application, such as saving custom configurations or contributing new exercises to the public library, the user must establish a persistent identity within the system. The registration process collects essential credentials which are validated by the authentication service. When accessing the system for the first time, users must enter their personal registration details:

- Email address
- Password
- Password confirmation
- Username

Once users have entered their details, the system stores them in the database so that they can be used for subsequent logins.

User Login

This feature allows the system to authenticate the user via the account created during the registration process. During login, the system displays fields for the user to enter their email address and password. The system then verifies the data by comparing it with the data in the database. If the data is correct,

the user is authenticated and can use the application. If the authentication data is incorrect, the system will prompt the user to correct it.

Exercise Creation

This feature allows users to contribute new content to the platform. The system provides a specialized editor where the user can define the musical pattern using a text-based notation. In addition to the musical content, the user must provide this metadata to ensure the exercise is correctly categorized:

- Title: a descriptive name for the exercise.
- Tags: selection of pedagogical functions, musical structures, or vocal qualities.
- Public/Private status: the user can choose whether to share the exercise with the community or keep it in their private collection.

The transposition and playback functionalities can also be used in this editor, allowing the user to verify the correctness of the notation before saving. Once the creation process is complete, the system stores the new exercise in the database.

Exercise Visualization and Customization

This feature allows users to view an exercise and adjust its parameters according to their needs. The system provides a visual interface that displays the musical notation or structure of the exercise. Users can interact with the exercise by modifying the following parameters:

- Tempo (BPM): the speed of execution.
- Vocal range limits: setting the initial note and the highest and lowest notes ensures the exercise stays within the user's comfortable tessitura.
- Playback controls: the ability to play, pause, or reset the exercise. The user can also save the generated audio and image files.
- Favoriting: the ability to save an exercise to a personal collection for quick access.

If the user is the owner of the exercise, they can also edit the exercise's metadata, such as its name, tags, and musical pattern, or delete it entirely.

Library Management

All exercises are accessible on the website through three distinct pages:

- **Public library**: a collection of exercises shared by the community, searchable by title, author or tags.
- **Personal library**: a private space on the user page where users can find the exercises they have created.
- **Favorites**: a list of exercises that the user has bookmarked for frequent use.

All exercises are stored in the database and can be distinguished by their name, owner, tags and visibility status.

User Profile Management

Users can view and manage their exercises from their profile page. This section provides a centralized dashboard for their created content. From this page, the user can also change their password or delete their account.

User Logout

This feature allows the user to terminate their current session securely. By selecting the logout option, the system invalidates the authentication token, ensuring that the user's account can no longer be accessed from the current browser without re-entering credentials.

Tutorial

This feature guides the user through the main interface components, explaining how to use the transposition engine and how to adjust the exercise parameters. This section also provides an overview of the musical notation used in the application. The tutorial is designed as a static page, in order to be non-intrusive, allowing users to consult it only if necessary.

3.2.2 Website overview

In this section we show the structure of the VoiceWorm application and illustrate the user interaction patterns through user flow maps.

Login and Register

If a user is not authenticated, they will land on the login page. From here, they can navigate to the registration page if they do not yet have an account. After a successful registration, the system redirects the user to the login page. Once authenticated, the user is redirected to the main dashboard, where they can begin exploring the library or navigating the website (Figure 13).

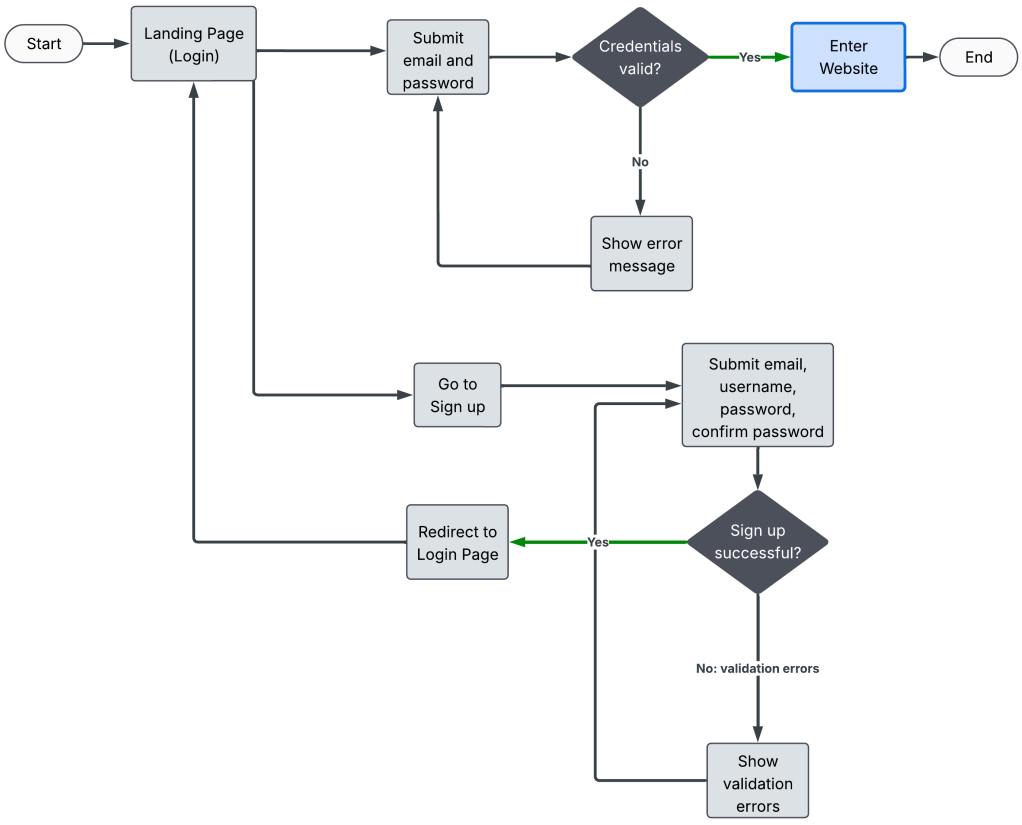


Figure 13: The login and registration user flow

Create exercise

From the navigation bar, the user can access the editor, where they input the musical pattern and metadata. Before saving, the user can preview the exercise and test it through transpositions and playback, to ensure that everything

is correct. Once satisfied, the exercise is saved to the database and becomes available in the user's personal library (Figure 14).

View exercise

When a user selects an exercise from the library, they are taken to the visualization page. Here, the system renders the musical score and provides the interface for adjusting the transposition parameters. The user can also add or remove the exercise from their favorites. If the user is the author of the exercise, they also have the option to modify the original pattern or metadata (Figure 15).

User page

The user profile page serves as a central hub for managing personal content and account settings. From this view, the user can browse their own created exercises and perform account management tasks such as updating the password or permanently deleting their profile (Figure 16).

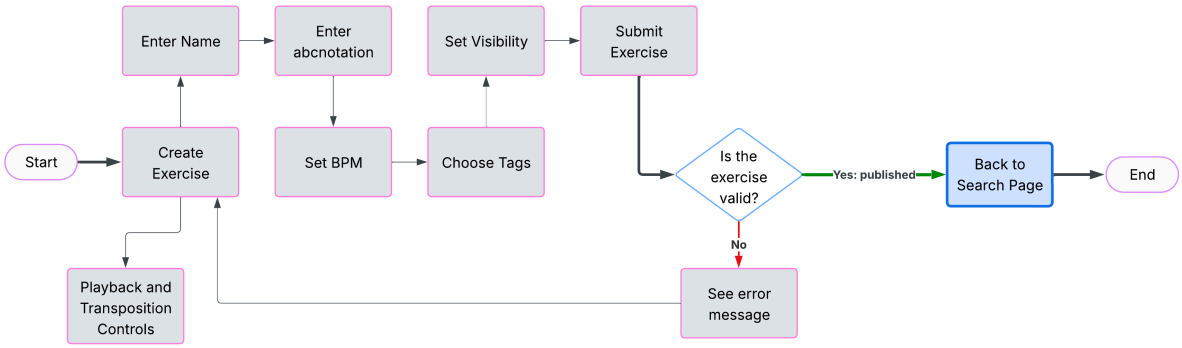


Figure 14: The exercise creation user flow.

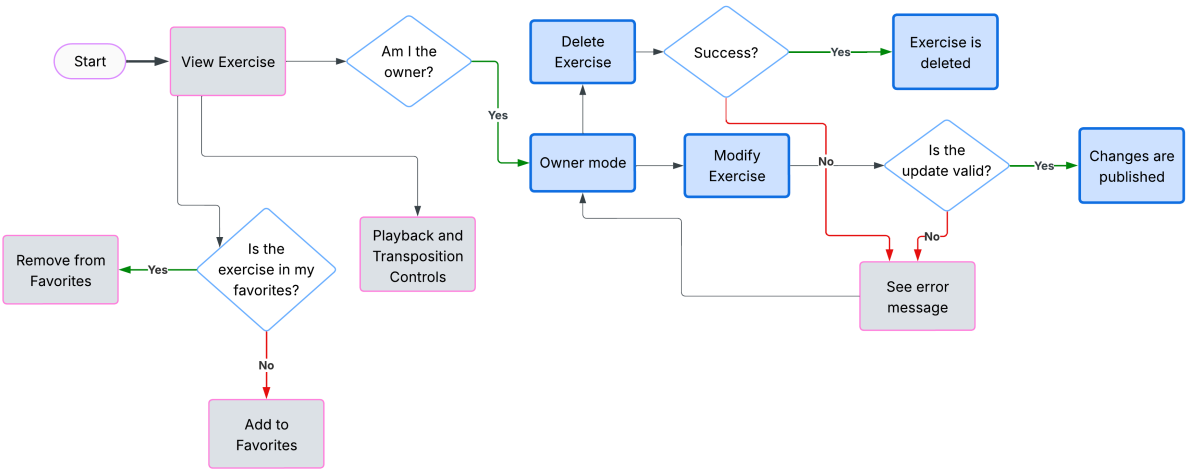


Figure 15: The exercise visualization and customization user flow

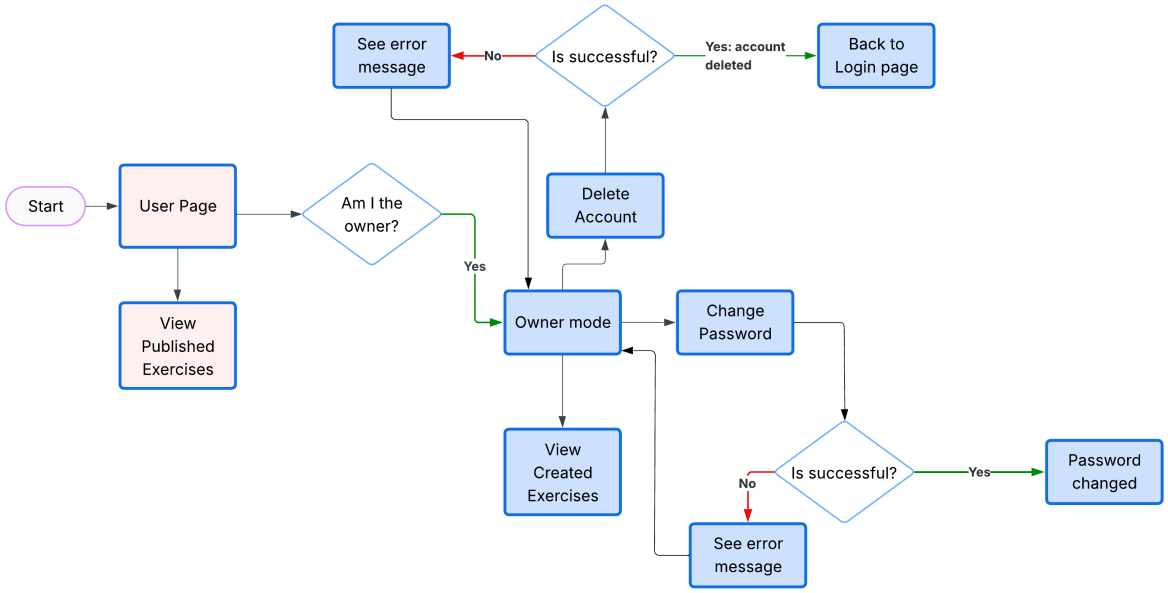


Figure 16: The user profile and personal library management flow

Navigation and Sitemap

The application's architecture is organized into several main sections accessible via a persistent navigation bar. The sitemap shows all the available pages on the website (Figure 17), which are summarized as follows:

- **Search/Home**: the landing page for authenticated users, providing a searchable database of all community-shared exercises.
- **Create**: the interface for creating and testing new vocal patterns.
- **Exercise View**: the dedicated page for performing and customizing a specific exercise.
- **User View**: the dedicated page for viewing a user's public information and exercises.
- **Profile**: the private dashboard for managing personal exercises and user settings.
- **Favorites**: a list of exercises that the user has bookmarked for frequent use.
- **Tutorial**: a guide explaining the notation system and application features.
- **Login/Register**: the entry points for user authentication.

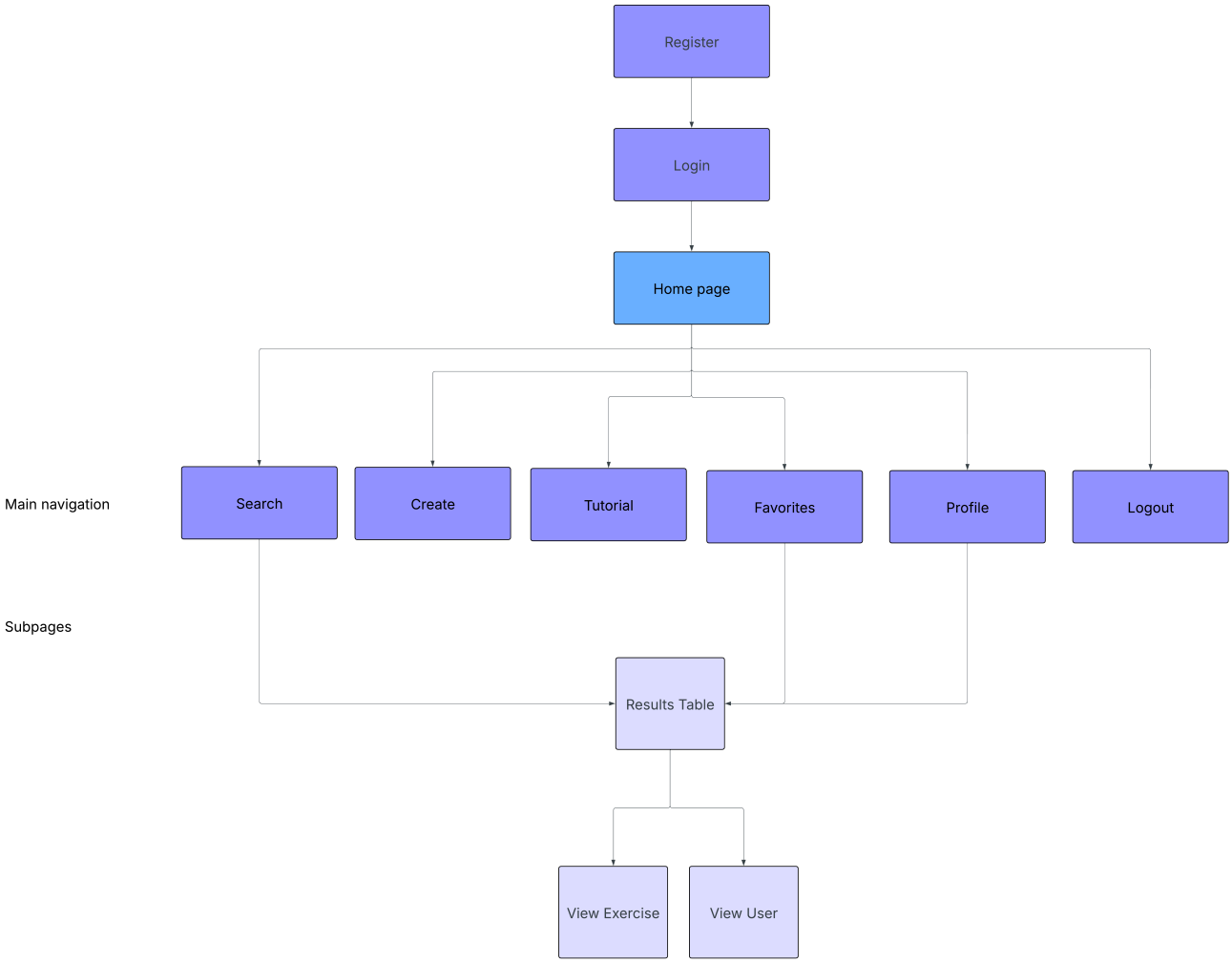


Figure 17: The sitemap of the VoiceWorm application

3.2.3 Database design

The system revolves around the **User** entity, which stores authentication details and profile information. Each user can create multiple exercises, establishing one-to-one ownership relationships. The **Exercise** entity stores core musical data, including the notation string, title, and visibility status. In addition to musical data, it tracks parameters set by users for playback and transposition, such as the BPM, the number of ascending and descending

steps, and the exercise range extremes. The **Tag** entity provides the metadata necessary for the multidimensional categorization system. These three entities all have unique identifiers (ID) as their primary keys.

A many-to-many relationship exists between Exercises and Tags to support the categorization system. This allows a single exercise to be associated with multiple pedagogical or musical descriptors while enabling each tag to be applied to numerous exercises. Furthermore, the favoriting functionality is implemented through a many-to-many relationship between users and exercises, enabling users to curate their own collections without affecting the original exercise data. This structure is illustrated in the entity-relationship diagram below (Figure 18).

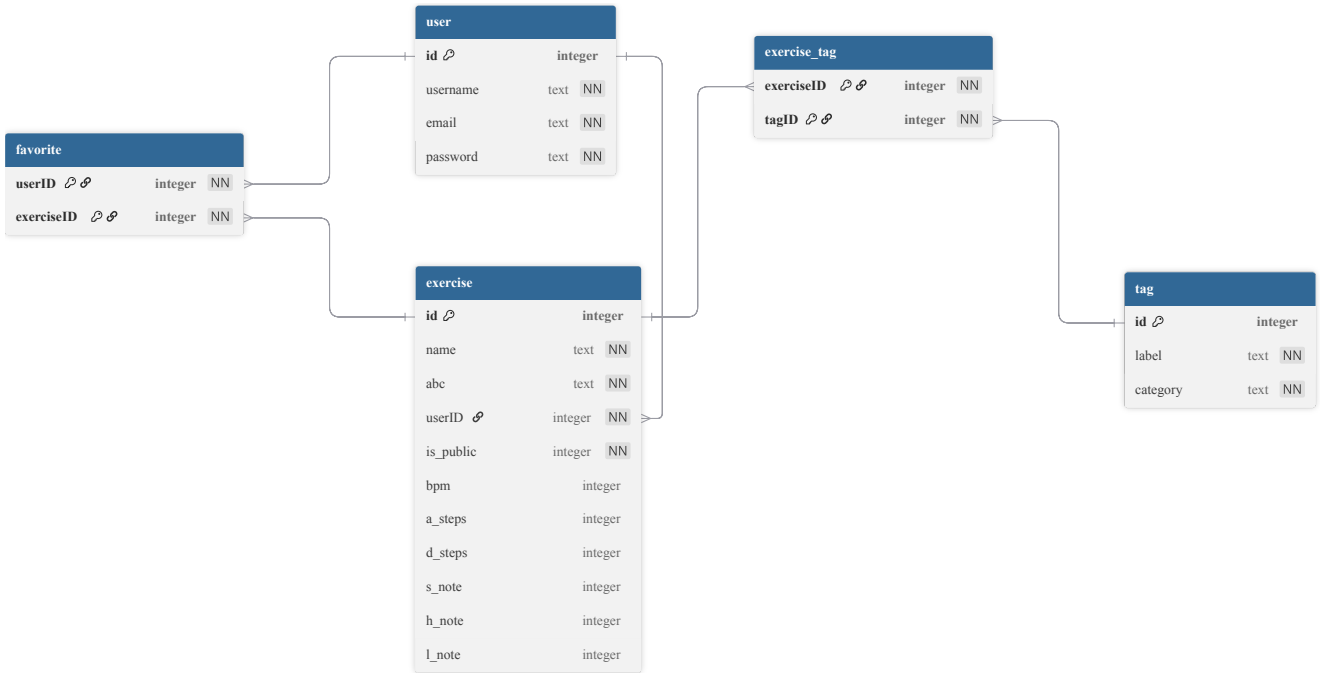


Figure 18: The ER diagram of the VoiceWorm database

- **User**: represents the primary entity, containing unique identifiers (IDs), credentials (emails and hashed passwords), and profile metadata (usernames). It maintains a one-to-many relationship with the exercises

created by the user and a many-to-many relationship with exercises marked as favorites.

- **Exercise**: stores the core musical data (abc notation string) and metadata (title, visibility). It also persists the specific execution parameters such as the base BPM, the transposition steps (ascending and descending) and the transposition boundaries (starting note, upper limit, and lower limit). It also contains the unique identifier of the user who created it (userID).
- **Tag**: a categorical entity used to classify exercises. All tags have a category, which is where a label belongs. For example, 'Minor Third' and 'Major Third' will be grouped under an 'Interval' macrocategory. Tags are linked to exercises via a many-to-many relationship.
- **Exercise Tag**: a table that links exercises to their respective tags, enabling the multidimensional categorization system.
- **Favorite**: a table that facilitates the many-to-many relationship between users and exercises, allowing users to bookmark content without duplicating data.

3.3 Development process

3.3.1 Preliminary choices

The initial development phase focused on identifying the necessary software libraries to meet the application's fundamental requirements of displaying musical scores in a browser and playing them back as audio.

Initially, the idea was to work with a music notation editor similar to MuseScore [54] that would allow users to view and edit sheet music in their browser. This option would have included the ability to import and export sheet music in MusicXML format. MusicXML is the de facto standard for exchanging digital scores between different music notation software programs. It is an XML-based[20] format that descriptively represents all the information

[20]XML (eXtensible Markup Language) is a markup language that defines rules for encoding documents in a format that can be read by humans and machines, using custom tags to structure the data.

in a score, including notes, duration, clefs, key signatures, lyrics, and more. Several solutions for rendering MusicXML were evaluated, including OpenSheetMusicDisplay [55] and Verovio [56]. OpenSheetMusicDisplay is an open-source JavaScript/TypeScript library that renders music scores in MusicXML format directly in the browser. It acts as a bridge between MusicXML and VexFlow, a JavaScript framework that is used to draw the score. Verovio is a tool that takes a score file in text format (MusicXML or MEI[21]) and translates it into a readable graphical representation, typically SVG [57]. In parallel with the rendering process, we analyzed the need for a converter to transform MusicXML into MIDI[22] and for a synthesizer to enable audio playback.

One of the major critical issues that emerged at this stage was the absence of complete open-source libraries for direct music editing via browser (drag-and-drop of notes). Third-party solutions such as Flat.io[58] and Scorio[59] were evaluated. In particular, Scorio Editor was the subject of an in-depth analysis: although it initially seemed promising for MIDI playback management and MusicXML import, a more detailed technical investigation revealed significant limitations. Analysis of network traffic and source code highlighted a dependency on non-transparent server-side components (probably Java applets). Both editors also imposed restrictive licenses for import/export functionality in their free versions.

To ensure the project's sustainability and independence, the approach based on proprietary graphic editors was abandoned in favor of textual notation. After considering the idea of creating an ad hoc musical grammar containing only the necessary elements for the project, abc notation was chosen. Supported by the abcjs library, this solution enables rendering, audio generation

[21] The Music Encoding Initiative (MEI) schema is an extensible open standard mark-up language for music notation.

[22] Musical Instrument Digital Interface (MIDI) is an international standard protocol created in 1982 that allows electronic musical instruments, computers, and other hardware/software devices to communicate with each other.

(WAV/MIDI), and vector export (SVG) to be managed directly on the client side while offering a lightweight format for storing exercises.

3.3.2 Transposition and playback engine development

At the heart of the application lies the management of vocal exercises, for which features not found in typical music players are required. These include the ability to transpose a melody progressively in both ascending and descending directions. Although score rendering, audio playback and animation have been entirely delegated to abcjs, the logic of automatic transposition posed the greatest complexity.

While transposing a score once is fairly easy, abcjs can only perform a visual transposition of the entire score, meaning a new abc string is not generated. To overcome this problem, we initially used an external utility called 'abc-notation-transposition' [60]. This utility takes a valid abc string and an integer number of semitones as input. The advantage is that it transposes the string before rendering it with abcjs. This was ideal since the plan was to select an initial abc string and perform a series of concatenations, each representing a step in the transposition process. However, this tool could not separate the body of the abc string from its header. Thus, each concatenation added an entire score structure (complete with clefs, key signatures, and time signatures) to the previous one. Technically, this caused the parser and the synthesizer to interpret the sequence not as a single modulating piece, but as a collection of separate, distinct tunes. Consequently, the audio engine had to stop and re-initialize the playback buffer for each new section, creating an audible gap between transpositions. This lack of continuity between one transposition and the next would have made performing the exercise unbearable.

First, we developed a function within the exercise views that separated the main part of the tune from the header. Then, we implemented another function to increase the number of semitones by the chosen amount for each notation element containing notes. However, testing revealed issues with managing key signatures, as the system failed to distinguish intervals in keys other than

C major. The view files were also very large and difficult to read because of the size of the script.

As it was not possible to rely on existing solutions, a custom parser had to be developed. This component analyzes the grammar of abc notation and selects the essential elements required for the tool to function. Rather than reading the entire abc text as a single entity, this script breaks it down into logical components that can be manipulated mathematically, particularly with regard to transposition. The parser recreates the musical logic from the individual elements and considers key changes, thus avoiding issues related to ‘naive transpositions’.

3.3.3 Backend architecture and containerization

Once all the issues related to the core transposition functionality had been resolved, development of the application proceeded more smoothly. Initially, VoiceWorm was a single view incorporating rendering, transposition and playback mechanisms. After ensuring that the core of the application was functioning properly, we began implementing the other views. This required the development of a solid and functional backend, particularly with regard to authentication and saving exercises. The authentication system does not store passwords as plain text, but uses hashing algorithms to transform them into irreversible strings before storing them in the database. The login process involves generating session tokens that are exchanged between the client and server to authorise subsequent requests.

To facilitate retrieval of exercise data from the database, each exercise was given a name, and all user-entered settings were tracked at the time of submission. These settings include the starting, highest, and lowest notes, expressed in semitone intervals, which the frontend then breaks down into notes and octaves. The database has a basic design built around user and exercise entities, paying attention to the many-to-many relationships represented by favorite exercises and exercise tags.

To ensure reproducibility of the development environment and facilitate project evaluation, the entire application was containerized using Docker. We

defined a Docker Compose configuration file to orchestrate the various services that make up the application. The containerized architecture is divided into three main services:

- Frontend: the container that hosts the application.
- Backend: the container that runs the server-side environment and manages connections to the local database.
- Reverse proxy: an Nginx[23] container that acts as a single entry point, routing requests to the frontend or backend based on the URL path.

This configuration solves compatibility issues between different operating systems, allowing the entire technology stack to be started with a single `docker compose up -d` command.

As an alternative, the frontend and backend can be started locally using the `npm run dev` command. This command is usually employed to launch a development server that monitors file modifications and automatically refreshes the application.

3.3.4 Deployment

The final phase of the development cycle involved deploying the application in a production environment, making VoiceWorm publicly accessible so that voice professionals could test it. To maintain a separation of concerns between the frontend and backend, we opted for an application hosting solution that natively integrates with the Cloudflare ecosystem. The deployment process is Git-based (or continuous deployment).

- The source code is pushed to the remote repository.
- Cloudflare Pages/Workers detects the changes and automatically starts a build process.
- During the build, the Vite/Wrangler build tool compiles the components and optimizes the static assets.
- Once compilation is complete, the artifacts are distributed across Cloudflare's edge network.

[23] Nginx is an HTTP web server, reverse proxy, content cache, load balancer, TCP/UDP proxy server, and mail proxy server.

The production process involved configuring an architecture based on specific subdomains. The domain architecture was structured as follows:

- Frontend (voiceworm.aisja.it): this subdomain hosts the user interface. It is a static site that communicates with the backend via asynchronous HTTP requests.
- Backend (api.voiceworm.aisja.it): this subdomain acts as an API gateway. All logical requests (e.g., authentication, exercise retrieval, and data saving) are directed here. This endpoint does not return graphical interfaces, but rather, structured responses in JSON format that can be consumed by the frontend.

4 Implementation

As previously mentioned, VoiceWorm is a web application with a modern, full-stack architecture. The project is organized into two main directories: `app`, which contains the Vue.js frontend application (Vue 3 with Composition API), and `backend`, which houses the Cloudflare Workers logic and database migrations. The root directory also contains the Docker configuration files for reverse proxy containerization. At the core of the project is the custom transposition engine (`app/lib/abcp.js`), made possible by integrating a custom-built abc notation parser.

4.1 Parsing

The parsing logic is responsible for transforming a string representation of musical notation into a structured object model that can be manipulated and regenerated. This process is broken down into several layers, from low-level character management to high-level structural aggregation. The parser file is configured as an LL(1) parser.

4.1.1 Input management

The foundation of the parser is the Input class, which wraps the raw string data. The class manages its traversal through the input string by maintaining an index property that tracks the current position. As it processes data, the `next()` method consumes the input by returning the current character and advancing the cursor, while the `peek()` method facilitates predictive parsing by allowing inspection of the character without advancing the index. To ensure stability throughout this process, utility methods such as `hasNext()` and `allRemainingAfterPeek()` validate operations to prevent exceeding the bounds of the input string.

```
class Input {
    constructor(string) {
        this.input = string;
        this.index = 0;
        this.length = this.input.length;
    }

    peek() {
        if (!this.hasNext()) return null;
        return this.input[this.index];
    }

    next() {
        if (!this.hasNext()) return null;
        return this.input[this.index++];
    }

    hasNext() {
        return this.index < this.length;
    }

    allRemainingAfterPeek(func) {
        for (let i = this.index + 1; i < this.length; i++) {
            if (!func(this.input[i])) return false;
        }
        return true;
    }
}
```

To validate the input, an `assert(cond)` function ensures that the parser is correctly processing the expected tokens, throwing an error if the internal state becomes inconsistent.

4.1.2 Musical properties

Before complex entities such as notes or chords can be created, the parser must identify and process the fundamental properties that define sound.

- **Duration**: in abc notation, a duration is expressed as a fraction, in relation to a predefined base duration (L:). The duration is simplified to its lowest terms thanks to a `gcd` utility function.
- **Pitch**: the Pitch class maps alphabetical characters (a-g) to semitonal numerical values using a static mapping constant `TONE_MAPPER`. The parser distinguishes between uppercase and lowercase letters. An uppercase letter (e.g., 'C') is interpreted as the base note ('c') with an octave modifier of −1, while the lowercase letter is the standard reference (0).
- **Octave**: the Octave class handles explicit register modifiers. The parser counts occurrences of apostrophes (') to go up an octave and commas (,) to go down. The resulting value is converted to semitones by multiplying the modifier by 12, allowing for easy calculation of the absolute pitch.
- **PartialAccidental**: this class deals exclusively with syntactic recognition. Using the `VISUAL_TO_SHIFT` table, it maps the visual symbols of abc notation (e.g. `^, _, =, ^^, __`) to integer shift values.
- **Accidental**: this class applies the symbol to the musical context.
  - Explicit Accidentals: if a PartialAccidental is present, the class applies the shift and updates the alterations status for that note, affecting subsequent notes in the same measure.
  - Implicit Accidentals: if there is no symbol, the class consults the alterations map (which contains the key signature or previous accidentals) to determine the correct shift to apply.

4.1.3 Music primitives

The core of the notation engine relies on the accurate representation of musical primitives: sound and silence. In the system's object-oriented architecture, these are modeled respectively by the Note and Rest classes.

Rest class

The Rest class represents a period of silence. Structurally, it is the simplest primitive in the system, requiring only a temporal dimension without frequency information. The class encapsulates a single property, `duration`. The parsing strategy utilizes a lookahead mechanism. The static `hasFirst` method

identifies the character literal ‘z’ as the unique token for rests. Upon validation, the parse method consumes the identifier and immediately delegates the remaining string processing to `Duration.parse`.

Note class

The Note class is the central atom of the musical score. Unlike the Rest class, the Note must manage a multidimensional state including pitch class, octaval register, chromatic alteration, and duration. However, to facilitate algorithmic manipulation, the class calculates a derived property: `this.data.tone`. This property represents the absolute chromatic pitch of the note as an integer, calculated via the summation of its component parts:

$$Tone = Pitch + (Octave \cdot 12) + Accidental$$

The `Note.parse` method enforces a strict grammatical order for musical symbols. The parser expects tokens in a specific order: accidental, pitch, octave, duration.

```
class Note {
    constructor(pitch, octave, duration, accidental) {}

    static hasFirst(_char) {
        return true;
    }

    static parse(input, alterations) {
        let partial_accidental =
PartialAccidental.parse(input);
        assert(Pitch.hasFirst(input.peek()));
        let pitch = Pitch.parse(input);
        let octave = Octave.parse(input);
        let duration = Duration.parse(input);
        let accidental = new Accidental(partial_accidental,
pitch.note, alterations);
        return new Note(pitch, octave, duration,
accidental);
```

```
    }

    generate(){}
    transpose(semitones, alterations) {}
}
```

4.1.4 Composite musical components

Composite musical components include the Chord and Tuplet classes.

Chord class

The Chord class represents a vertical aggregation of musical events, sounds occurring simultaneously. In the provided implementation, a chord is defined as a collection of elements sharing a single duration. The parser identifies chords via the bracket delimiters ([ ]). The `Chord.parse` method initializes an array and enters a loop that continues until the closing bracket is detected.

Tuplet class

The Tuplet class handles irrational rhythms. The syntax for a tuplet is denoted by an opening parenthesis '(' followed immediately by a number `n`. Unlike the Chord parser which looks for a delimiter, the Tuplet parser is counter-based. It loops exactly `n` times, consuming valid musical tokens (Chord, Rest, or Note) until the quota is met.

4.1.5 Aggregation and containers

The system's highest level of abstraction is managed by the Bar and Score classes.

Bar class

The Bar class represents a single measure of music, defined by the '|' delimiter. It serves as a local container for musical events, aggregating Note, Rest, Chord, and Tuplet objects into a linear sequence. A critical responsibility of the Bar is managing the scope of accidentals. In standard music notation, an accidental typically applies only until the end of the current measure. When `Bar.parse` is called, it accepts a set of `globalAlterations` (the key

signature). The method immediately creates a deep copy of these globals using `structuredClone(globalAlterations)`. Any subsequent accidentals encountered within the bar modify this local alterations copy, ensuring that temporary shifts do not leak into subsequent measures.

Score class

The Score class represents the complete musical work. It is the root of the object graph, containing an ordered list of Bar objects, along with global metadata like key signature (represented as alterations). The `Score.parse` method processes the input stream in three distinct phases:

- Prefix: consumes initial whitespace.
- Body: iteratively parses Bar objects until no more bars are found.
- Suffix: consumes trailing whitespace.

The Score maintains the 'truth' of the key signature. By default, it initializes with `C_MAJOR` (no sharps or flats) unless specified otherwise. When parsing begins, this global state is passed down to each Bar, ensuring consistent tonality across the piece.

```
class Score {
    constructor(bars, prefix, suffix, alterations) {}

    static parse(input, alterations = C_MAJOR) {
        let bars = new Array();
        let prefix = new Array();
        let suffix = new Array();
        while (Whitespace.hasFirst(input.peek())) {
            prefix.push(Whitespace.parse(input));
        }
        while (Bar.hasFirst(input.peek())) {
            if
(input.allRemainingAfterPeek(Whitespace.hasFirst)) {
                assert(Bar.hasFirst(input.next()));
                break;
            }
```

```
            bars.push(Bar.parse(input, alterations));
        }
        while (input.hasNext()) {
            assert(Whitespace.hasFirst(input.peek()));
            suffix.push(Whitespace.parse(input));
        }
        return new Score(bars, prefix, suffix, alterations);
    }

    generate() {}
    transpose(semitones) {}
    extend(score) {}
}
```

4.1.6 Transposition

Transposition is implemented as a mutable operation on the object graph. Rather than creating new instances, the transpose method modifies the internal state of existing objects in place. The system employs a recursive strategy for transposition, where container objects delegate the operation to their children. This ensures that a single call to `Score.transpose` propagates correctly down to every individual note.

The actual mathematical logic resides solely within `Note.transpose`. The note's pitch is first converted to an absolute integer value (`this.data.tone`) representing its semitone distance from a fixed origin. The desired interval is added to this integer. To convert the new integer back into a musical symbol, the system queries a lookup table called `TONE_EXPLORER`.

- This table maps every integer pitch class (0-11) to a set of valid enharmonic spellings (Pitch, Octave, Accidental).
- The algorithm iterates through these options, constructing temporary objects to see which spelling mathematically matches the new target tone.

```
class Note {
  transpose(semitones, alterations) {
        this.data.tone += semitones;
        let tone = this.data.tone % 12;
        let octave = Math.floor(this.data.tone / 12);
        let choices = TONE_EXPLORER[tone];
        for (let choice of choices) {
            let t_pitch = new Pitch(choice[0], 0);
            let t_octave = new Octave(octave + choice[2] -
5);
            let t_p_accidental = new
PartialAccidental(choice[1]);
            let t_accidental = new
Accidental(t_p_accidental, choice[0], alterations);
            if (t_octave.modifier == -1) {
                t_pitch.modifier = -1;
                t_octave.modifier = 0;
            }
            let t_note_tone = t_pitch.toTone() +
t_octave.toTone() + t_accidental.toTone();
            if (t_note_tone === this.data.tone) {
                this.pitch = t_pitch;
                this.octave = t_octave;
                this.accidental = t_accidental;
                break;
            }
        }
    }
}
```

4.1.7 Generation

The `generate()` method serves as the inverse of the parsing process, serializing the internal object graph back into the abc-like string format. Like transposition, this is achieved via polymorphism, where each class is responsible for serializing its own data.

```
class Score {
    generate() {
        let output = "";
        for (let el of this.prefix) {
            output += el.generate();
        }
        for (let bar of this.bars) {
            output += bar.generate();
        }
        output += "|";
        for (let el of this.suffix) {
            output += el.generate();
        }
        return output;
    }
}
```

4.1.8 Concatenation

The `extend()` method allows for the horizontal composition of music appending one score to the end of another. Before merging, the method enforces a strict consistency check. It verifies that the alterations of the incoming score match exactly with the current score's key signature. Once validated, the extension process is straightforward:

- The bars from the incoming score are pushed directly into the current score's bars array using the spread syntax (…).
- The current score's suffix is replaced by the incoming score's suffix, ensuring that the final formatting of the combined piece is consistent with the appended segment.

```
class Score {
    extend(score) {
      assert(score instanceof Score);
```

```
        for (let key in this.alterations) {
            assert(this.alterations[key] ===
score.alterations[key]);
        }
        this.bars.push(...score.bars);
        this.suffix = score.suffix;
    }
}
```

4.2 Frontend module

4.2.1 General ‘app’ folder structure

The application’s source code is stored in the `app` folder. This folder contains the configuration files and the `src` directory, where the actual development takes place.

- **package.json**: defines the project’s dependencies, metadata, and scripts for building and running the frontend application.
- **package-lock.json**: locks the versions of dependencies to ensure consistent installations across different environments.
- **vite.config.js**: configures the Vite build tool, including plugins, server settings, and path aliases.
- **.env.development**: contains environment variables for the development environment, such as the API base URL.
- **.env.production**: contains environment variables for the production environment, such as the production API endpoint.
- **Dockerfile**: defines the instructions for building the Docker image of the frontend.
- **index.html**: the main entry point for the web application, providing the basic HTML structure where the Vue application is mounted.
- **public**: the directory that contains static assets like the favicon and the logo.
- **src**: the directory containing all the source code of the application.

src directory structure

The `src` directory organization is illustrated below:

- **App.vue, main.js**: the root component and the entry point that initializes the Vue application, respectively.
- **router**: manages client-side navigation.
- **views**: contains the application's pages.
- **components**: hosts the building blocks of the user interface that are used within the views.
- **composables**: collects reactive and reusable state logic.
- **lib**: contains support libraries, constants and pure utility functions.
- **stores**: manages the global state of the application using Pinia, handling user authentication and session persistence.
- **assets**: stores static resources such as images, icons, and global CSS styles.

4.2.2 Entry point and application bootstrapping

The entry point, defined in `main.js`, is responsible for bootstrapping the application instance and injecting global dependencies required for the component tree to function. The application instance is created using Vue's `createApp` factory. Before mounting to the DOM, the architecture registers two critical plugins via the `.use()` method:

- Pinia: the `createPinia()` instance is registered to handle global state management.
- Vue Router: the router instance is registered to manage client-side navigation and URL resolution.

The `App.vue` file serves as the root node of the Vue component hierarchy.

```
<script setup>
import { RouterView } from 'vue-router'

</script>

<template>
```

```
  <RouterView />
</template>

<style scoped>
</style>
```

The component's template contains a single directive: `<RouterView />`. This component acts as a dynamic placeholder. It detects the active route from the URL and automatically renders the matched component.

4.2.3 Routing

The application utilizes `vue-router` to manage client-side navigation, mapping URL paths to specific view components. This setup enables the SPA to simulate a multi-page environment while maintaining state and avoiding full page reloads. The router instance is configured to adapt its history mode based on the deployment environment. The router must trick the browser into displaying different content for different URLs without actually triggering a page reload request to the server. The code provided uses a ternary operator to choose between two strategies depending on whether the app is in Production or Development mode. It utilizes `createWebHistory()`, which leverages the HTML5 History API for clean, standard URLs (e.g., `example.com/login`). It falls back to `createWebHashHistory()`, using the URL hash (e.g., `localhost/#/login`) to manage routing state without requiring server-side configuration.

The route table defines the application's navigational structure. It handles both static paths and dynamic segments that accept parameters:

- **Static Routes**: standard views such as Login, Register, and Create are mapped to their respective components.
- **Dynamic Routes**: the configuration uses colon syntax to capture variable segments of the URL. For example, `/exercise/:id` loads the ExercisesView, passing the exercise ID as a parameter.

Security is enforced at the routing level via a global `beforeEach` navigation guard. The guard checks the meta field of the target route. The logic dictates that unless a route is explicitly marked with `noAuth:true`, the user must be authenticated.

```
const router = createRouter({
  history: import.meta.env.PROD ? createWebHistory() :
createWebHashHistory(),
  routes: [
    {
      path: '/login',
      name: 'login',
      component: LoginView,
      meta: {
        noAuth: true,
      }
    },
    {
      path: '/exercise/:id',
      name: 'exercise',
      component: ExercisesView,
      meta: {
        noAuth: false,
      }
    }]
```

4.2.4 State management

The application leverages Pinia, the standard state management library for Vue 3, to handle global data that must be accessible across the component tree. A key requirement for the user experience is state persistence, ensuring that user sessions and interface preferences survive page reloads. The implementation addresses this by integrating `useLocalStorage` from the `@vueuse/core` library directly into the store definitions. When a user refreshes the page, the application immediately rehydrates the `token` and `darkMode` preferences.

The `useCredentials` store manages the security context of the user session. It is designed around the JSON Web Token (JWT) standard. The state is minimalist, storing only the raw JWT. The actions `login(token)` and `logout()` are simple mutators that update this state.

- The Token Parser performs client-side decoding of the JWT. It splits the token string, extracts the payload section, and decodes the Base64-encoded JSON (`atob(payload)`).
- The Session Validator determines if the user is currently logged in. It validates two conditions: the existence of a token, and the expiration status of the token.

```
import { defineStore } from "pinia";
import { useLocalStorage } from "@vueuse/core";

export const useCredentials = defineStore("credentials", {
    state: () => ({
        token: useLocalStorage("token", null),
    }),
    getters: {
        isTokenPresent: (state) => {
            return state.token !== null;
        },
        data: (state) => {
            if (state.isTokenPresent) {
                try {
                    const payload = state.token.split(".")
[1];

                    return JSON.parse(atob(payload));
                }
                catch (e) {
                    console.error("Error decoding token",
e);

                    return null;
                }
            }
```

```
        },
        isAuthenticated: (state) => {
            return state.isTokenPresent && state.data.exp >
Date.now() / 1000;
        },
    },
    actions: {
        login(token) {
            this.token = token;
        },
        logout() {
            this.token = null;
        }
    }
})
```

The `useTheme` store manages the visual presentation preference. The `toggleDarkMode` action flips the boolean state. Crucially, it immediately applies this change to the document body by setting a data attribute: `document.body.setAttribute('data-theme', ...)`.

4.2.5 Views

The application's views are the primary components that define the user interface and coordinate the interaction between the user, the global state, and the backend services. Each view is responsible for a specific functional area of the application.

SearchView

The home page and main discovery hub. It fetches the list of public exercises from the backend and provides a filtering interface. Users can search by title, author or tags (Figure 19).

UserView

This is the user profile view. It distinguishes between owner and non-owner modes. Owners can manage their own exercises and update their account

settings (Figure 20), while non-owners can view the profile owner's published exercises.

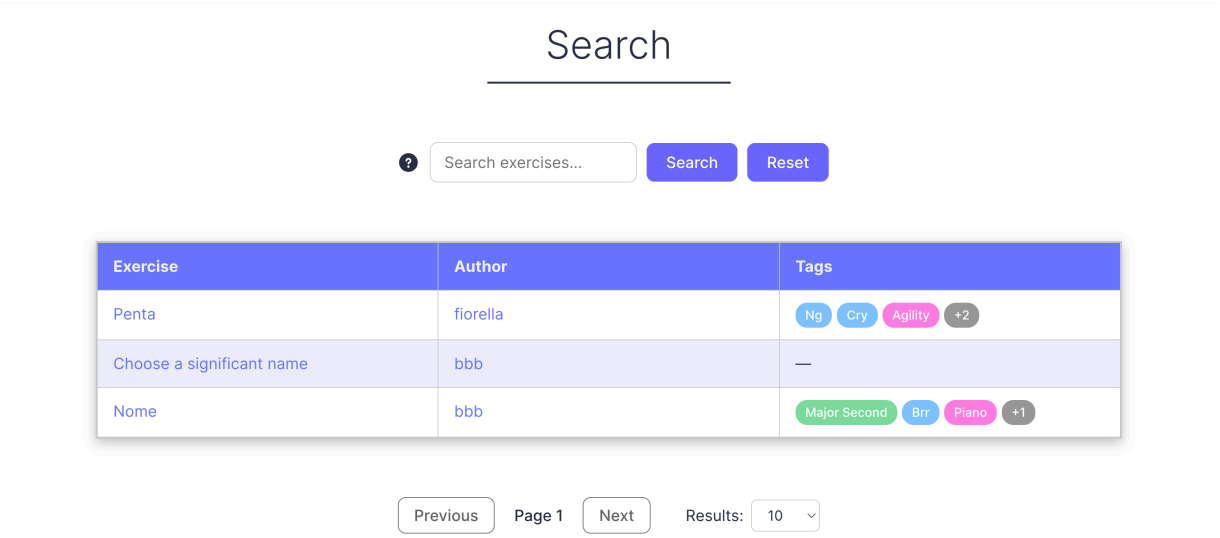


Figure 19: The search interface, showing the exercise list and filtering options

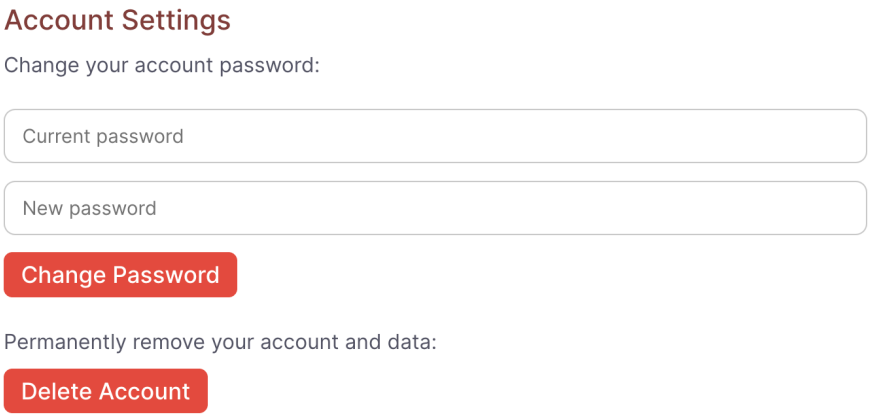


Figure 20: The account management options of the user profile

FavoriteView

Displays a list of exercises that the user has marked as favorites, allowing for quick access to preferred vocal routines.

TutorialView

Provides a static educational resource for users. It details the abc notation system used by the application and explains the core concepts of the transposition engine (Figure 21).

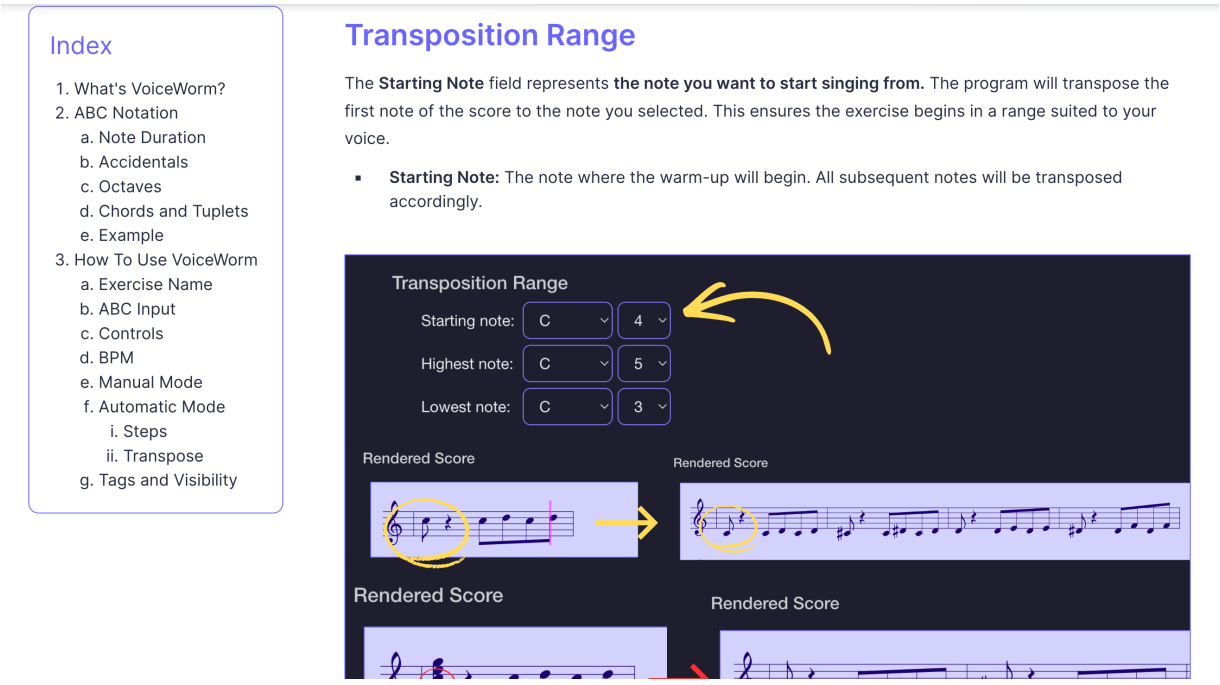


Figure 21: The tutorial page

ExerciseView

This is the view that handles the rendering and execution of a specific exercise. It distinguishes between owner and non-owner modes. It enables users to render the score, adjust the BPM, set vocal range limits and control playback. The owner can also modify the abc notation and metadata of the exercise or delete it.

CreateView

It provides a specialized editor for creating new exercises. It includes a live preview feature that renders the abc notation in real time as the user types. There is also a form for selecting metadata and tags, as well as the option to test the transposition and playback functions (Figure 22).

The transposition system has two modes: ‘Manual Mode’ and ‘Automatic Mode’. In Manual Mode, users input the transposition steps manually by clicking the ‘+’ or ‘-’ symbols to switch between the starting, ascending, and descending phases (Figure 23). In Automatic Mode, users select the transposition steps (in semitones) and the desired range for the exercise. Clicking ‘Generate’ creates a complete exercise with these parameters (Figure 24).

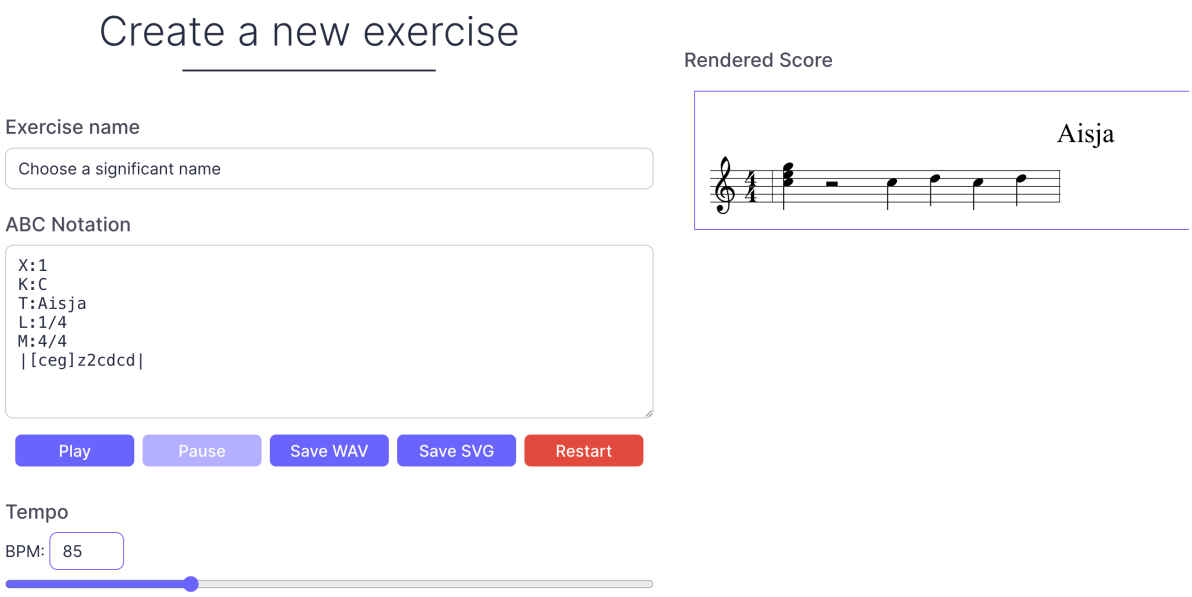


Figure 22: The create page and playback controls.

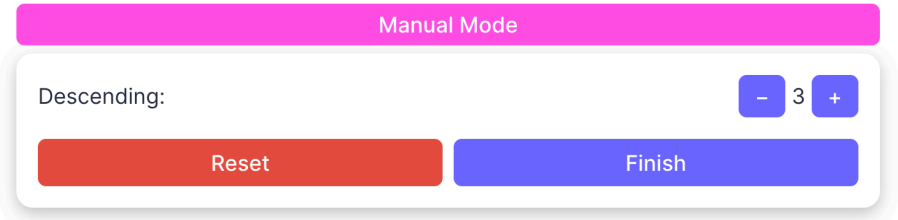


Figure 23: The create page manual transposition

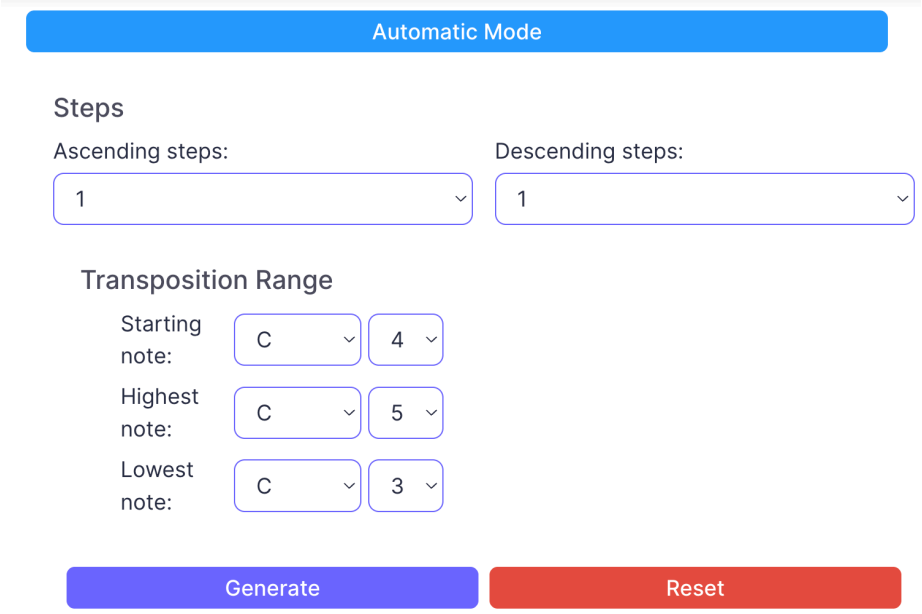


Figure 24: The create page automatic transposition

Login/Register View

Handle user authentication, managing the forms and communicating with the backend to establish a secure session.

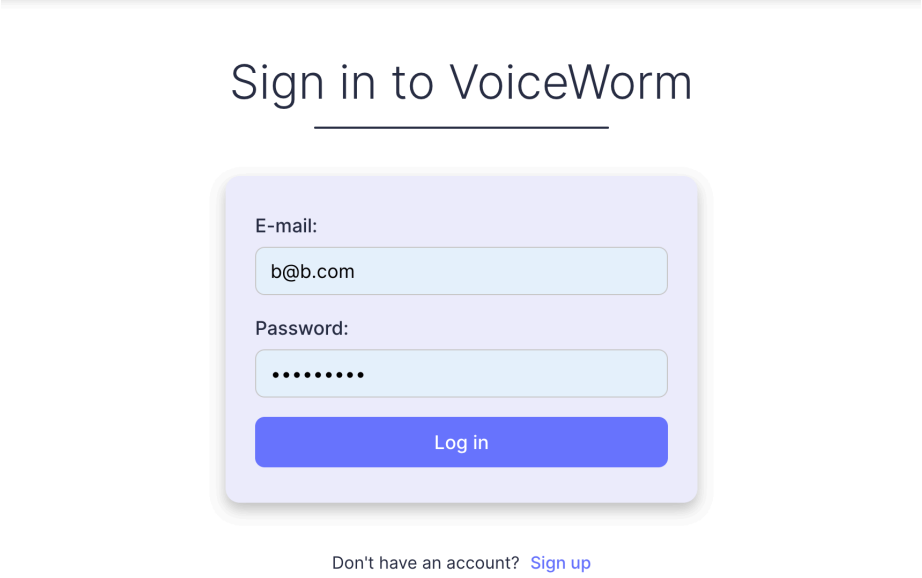


Figure 25: The login page

4.2.6 Components

The application's UI is built from modular, reusable components.

- **Header and Footer**: the `Header` component provides the primary navigation bar, which includes the logo, links to various views, the theme switcher, and the logout option. When not logged in, only the logo and the theme switcher will be displayed in the header. The `Footer` component contains static links and project information.
- **ExerciseTable**: a reusable data table component used to display lists of exercises. It supports sorting, pagination, and provides a consistent layout for exercise metadata (title, author, tags). It is used across the `Search`, `User`, and `Favorite` views to ensure a uniform data presentation.
- **ExerciseForm**: a comprehensive form component used for both creating and editing exercises. It manages the input for metadata and the abc no-

tation editor with a live preview. It also integrates the transposition logic, allowing users to define the boundaries for automatic transposition, or manually adjust the transposition steps.

4.2.7 Composables

To maintain a clean separation of concerns, the application relies heavily on Composables. These are custom hooks that encapsulate stateful logic, allowing API communication and music rendering to be imported into views without cluttering the UI code.

useApiClient

The `useApiClient` composable abstracts the configuration of the HTTP client. It uses the `axios` library to create a pre-configured instance. Before every request, the interceptor accesses the token stored in the credentials. If a token is found, it is automatically added to the Authorization header in the form of a Bearer token (`Bearer ${credentials.token}`). This pattern eliminates the need for individual components to handle authentication tokens manually. The composable also exports a utility function called `withMinDelay`. This enforces a minimum execution time to prevent UI 'flickering', which occurs when a loading element appears and disappears too quickly for the user to perceive it comfortably during page transitions.

useScore

The `useScore` composable is the engine of the application's frontend. The `abcjs.renderScore` function takes the current `userText` and renders it to a DOM element. The `play` function initializes the synthesizer with the current visual object and primes the audio buffer. It instantiates `abcjs.TimingCallbacks`. This callback system fires events during playback, providing the specific coordinate of the note currently being played. The composable updates the `scrollbarLeft` ref inside `nextTick`, ensuring the UI cursor moves in sync with the audio.

The `transposeAndRender` function demonstrates the practical application of the Score class (Chapter 4.1). It converts the raw `userText` into a Score

object using `Score.parse`. It calculates the semitone difference between the user's input notes and the target range limits. The function iterates through the range, cloning the score (`lodash.cloneDeep`), transposing it by the interval, and extending the accumulator score.

The composable provides methods to export the generated content. `downloadSvg` serializes the DOM nodes created by the renderer into a downloadable file. `downloadWav` manually constructs a RIFF WAVE file.

4.2.8 Styling

The application's visual layer is built using SCSS (Sass). A core requirement for the user interface is the support for light and dark themes. Rather than loading separate stylesheets, the application implements a variable-driven theming engine. The system defines a set of semantic CSS variables (e.g., `--bg-color, --text-color, --accent-color`) attached to the `<body>` element. The values of these variables change based on the data-theme attribute.

```
body[data-theme="light"] {
  --bg-color: #ffffff;
  --text-color: #2b2d42;
}

body[data-theme="dark"] {
  --bg-color: #1e1e2f;
  --text-color: #f0f0f0;
}
```

To improve perceived performance during data fetching, the application also implements a skeleton loading state. A global `@keyframes skeleton-loading` animation creates a pulsing effect that mimics the layout of text rows, reducing layout shifts and providing a smoother transition than a generic spinner.

4.3 Backend module

The backend of VoiceWorm is built using Hono, a fast, lightweight web framework designed for Cloudflare Workers. It handles API routing, authentication, and database interactions through Cloudflare D1.

4.3.1 General 'backend' folder structure

The `backend` directory contains the server-side logic and database management tools:

- **Dockerfile**: defines the instructions for building the Docker image of the backend, setting up the Node.js environment to run the Hono server.
- **package.json**: lists the backend dependencies, such as Hono and Wrangler, and defines scripts for database migrations and deployment.
- **wrangler.jsonc**: the configuration file for Wrangler, Cloudflare's CLI, which defines the environment settings, database bindings for D1, and deployment parameters.
- **db**: contains the SQL migration files used to define the database schema and seed initial data.
- **.dev.vars**: stores sensitive environment variables for the local development environment, such as the JWT secret key for authentication.
- **src**: the directory containing the backend source code, including the API routes, database controllers, and authentication middleware.

The `src` directory contains the following files:

- **lib/pbkdf2.js**: implements the PBKDF2 (Password-Based Key Derivation Function 2) algorithm to securely hash and verify user passwords before they are stored in the database.
- **index.js**: the main entry point for the backend application. It initializes the Hono instance, configures CORS (Cross-Origin Resource Sharing) to allow requests from the frontend, and defines the API routes for every feature.

4.3.2 Database management

The database schema is defined in the file `db/voiceworm.sql`. This file uses DDL to define the structure, creating tables and relationships. The main entities are ‘user’, ‘exercise’ and ‘tag’, while the relationship tables (many-to-many) are ‘favorite’ and ‘exercise_tag’. Each of the main entities is easily accessible through its unique identifier (ID), while the relationship tables are linked to the primary keys of the entities they connect.

Foreign keys ensure referential integrity, meaning that an exercise cannot be associated with a non-existent entity. The ‘ON DELETE CASCADE’ clause also ensures that, if a user or exercise is deleted, all related entries in other tables are automatically removed. Data that is essential is defined as ‘NOT NULL’, while some data has a default value to ensure consistency even when the user does not provide specific inputs.

The following SQL code snippet illustrates the structure of the `exercise` and `favorite` tables:

```
CREATE TABLE exercise (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    abc TEXT NOT NULL,
    userID INTEGER NOT NULL,
    is_public INTEGER NOT NULL DEFAULT 0,
    bpm INTEGER,
    a_steps INTEGER,
    d_steps INTEGER,
    s_note INTEGER,
    h_note INTEGER,
    l_note INTEGER,
    FOREIGN KEY (userID) REFERENCES user(ID) ON DELETE
CASCADE
);

CREATE TABLE favorite (
    userID INTEGER NOT NULL,
```

```
    exerciseID INTEGER NOT NULL,

    PRIMARY KEY (userID, exerciseID),
    FOREIGN KEY (userID) REFERENCES user(ID) ON DELETE
CASCADE,
    FOREIGN KEY (exerciseID) REFERENCES exercise(ID) ON
DELETE CASCADE
);
```

The `tags.sql` file uses DML to manage the initial population. It inserts data into existing tables. In this case, it was important to have a script to manage the insertion of tags because they were numerous and subject to frequent changes, which would otherwise require amending the entire database.

```
-- Objective
INSERT INTO Tag (category, label) VALUES
('Objective', 'Resonance'),
('Objective', 'Stability'),
// . . .
('Objective', 'Melisma'),
('Objective', 'Riffs and Runs');
```

As previously mentioned, the database operational management is handled through Wrangler. During development, the `wrangler d1 execute` command is used to apply these SQL scripts to the local or production database, ensuring the schema remains synchronized with the application logic.

4.3.3 Password hashing and authentication

Password management is never performed in plain text. The system implements the PBKDF2 (Password-Based Key Derivation Function 2) algorithm, an industry standard designed specifically to make brute force attacks computationally expensive [61].

A unique 16-byte salt is generated for each new password using a cryptographically secure pseudo-random number generator. The user's password is then combined with the salt and subjected to 1,000 iterations of the SHA-256

hashing algorithm. This transforms the password into a fixed-length (256-bit) derived key. The final result, which is stored in the database, is a composite string in the format ‘saltHex:hashHex’. This concatenation is essential for subsequent verification, as it enables the system to retrieve the salt specific to that user.

```
export async function hashPassword(password,providedSalt) {
  const encoder = new TextEncoder();
  const salt = providedSalt || crypto.getRandomValues(new
Uint8Array(16));
  const keyMaterial = await crypto.subtle.importKey(
    "raw",
    encoder.encode(password),
    { name: "PBKDF2" },
    false,
    ["deriveBits", "deriveKey"]
  );
  const key = await crypto.subtle.deriveKey(
    {
      name: "PBKDF2",
      salt: salt,
      iterations: 1000,
      hash: "SHA-256",
    },
    keyMaterial,
    { name: "AES-GCM", length: 256 },
    true,
    ["encrypt", "decrypt"]
  );
  const exportedKey = (await crypto.subtle.exportKey(
    "raw",
    key
  ));
  const hashBuffer = new Uint8Array(exportedKey);
  const hashArray = Array.from(hashBuffer);
  const hashHex = hashArray
```

```
    .map((b) => b.toString(16).padStart(2, "0"))
    .join("");
  const saltHex = Array.from(salt)
    .map((b) => b.toString(16).padStart(2, "0"))
    .join("");
  return `${saltHex}:${hashHex}`;
}
```

In the `index.js` file, during the registration process, the endpoint first checks a `REGISTRATION_SECRET_TOKEN`. This prevents the general public from creating accounts unless they have this specific app secret. It calls `hashPassword(password)` to convert the user's input into the safe salt:hash format, and to store it hashed.

The authentication process is managed through JSON Web Tokens using the jose library [62]. When a user successfully logs in, the server generates a token signed with a secret key. This token is then sent back to the client and stored in the browser. For every subsequent request to a protected route, the client includes this token in the Authorization header. The server uses a middleware to verify the token's validity before granting access to the requested resource.

```
import { hashPassword, verifyPassword } from "./lib/pbkdf2";
import * as jose from "jose";
import { bearerAuth } from 'hono/bearer-auth'

// Middleware to verify JWT and extract user information
const auth = bearerAuth({
  verifyToken: async (token, c) => {
    try {
      const secret = new
TextEncoder().encode(c.env.JWT_SECRET);
      const { payload } = await jose.jwtVerify(token,
secret);
```

```
      c.set("user", payload);
      return true;
    } catch (err) {
      return false;
    }
  }
});
```

Once the user has the JWT, they are authenticated for subsequent requests via the auth middleware. The app uses `bearerAuth` middleware from Hono. This expects the client to send the token in the header: `Authorization: Bearer <token>`. The middleware uses `jose.jwtVerify` to check that the token was signed by your server and hasn't expired. If valid, the user's data (the payload) is attached to the request context via `c.set("user", payload)`. This is why routes like `/exercises` can access `c.get("user").id` to filter data specifically for that user.

```
// To delete the user account, the user must be
authenticated.
app.delete("/user/me", auth, async (c) => {})
```

4.3.4 API Routes

The backend logic is organized into several RESTful endpoints that handle the communication between the frontend and the database. These routes are categorized by their functional domain:

- **Authentication Routes**:
  - `POST /register`: validates the registration secret and user credentials, hashes the password, and creates a new user record.
  - `POST /login`: verifies the provided email and password against the stored hash. If successful, it generates and returns a JWT.
- **User Routes**:
  - `GET /user/:id`: returns public profile information and the public exercises of a specific user.

- `DELETE /user/me`: A protected route that allows a user to permanently delete their account and all associated data.
- `PUT /user/me/password`: a protected route that allows an authenticated user to update their account password after verifying the current one.

- **Exercise Routes**:
  - `POST /exercises`: a protected route that allows authenticated users to save a new exercise.
  - `GET /exercises`: retrieves a list of all public exercises.
  - `GET /search/exercises`: allows users to filter the public library based on titles, authors or tags.
  - `GET /exercises/:id`: fetches the full details of a specific exercise by its ID.
  - `GET /exercises/:userid`: retrieves all exercises created by a specific user.
  - `PUT /exercises/:id`: allows the owner to update an existing exercise's metadata or musical content.
  - `DELETE /exercises/:exerciseid`: allows the owner to remove an exercise from the system.
- **Tag and Favorite Routes**:
  - `GET /tags`: returns all available tags.
  - `POST /tags`: creates a new tag (used in the beta version before adding the tags.sql script).
  - `GET /favorites`: retrieves the list of exercises bookmarked by the current user.
  - `GET /favorites/:exerciseid`: checks if a specific exercise is in the user's favorites.
  - `POST /favorites/:exerciseid`: adds an exercise to the user's favorites.
  - `DELETE /favorites/:exerciseid`: removes an exercise from the user's favorites.

5 Requirements validation

Developing an app dedicated to vocal training and warm-ups can present challenges in terms of user experience. Singers, speakers and voice professionals often work in a variety of contexts and have developed study habits that should be supported by the software rather than disrupted. One of the greatest risks in a project's life cycle is misalignment between the features envisaged by the developer and the actual needs of the end user. This chapter describes a requirements validation study conducted after the implementation phase to assess the impact of this issue.

The following sections detail the survey's specific objectives, the participants' profiles, the data collection methodology and key observations on the study's validity.

5.1 Objectives

Unlike traditional usability tests, which focus on carrying out specific tasks on an interface, this study aimed to validate expectations. A questionnaire was given to a selected group of potential users, asking about their ideal requirements, current habits and perception of the app's core features. The main objective was to shift the focus from 'how to use the developed app' to 'what users naturally look for in an app of this type'. Analyzing these expectations at this stage enabled design choices to be validated and confirmed whether critical features implemented in the software corresponded to user priorities. Any discrepancies between the application's current offering and user desires were identified to inform the roadmap for future software iterations. Specifically, the study aimed to answer the following questions:

- How much do you value vocal warm-ups in a vocal teaching context?
- What are the most common contexts in which users perform vocal warm-ups?

- What are the main difficulties in online vocal training and warm-up routines?
- Which features are considered ‘essential’ versus ‘optional’ for a vocal warm-up application?

5.2 Participants

A total of 52 participants took part in the study. Most respondents were recruited from the professional network of the Voice Evolution Institute, a specialised center for vocal research and training[24].

The participant pool can be categorized into four main categories:

- **Voice teachers**: professionals with in-depth pedagogical knowledge. Their feedback provided insights into whether the app supports correct vocal hygiene and educational standards.
- **Students and trainees**: active learners at various levels of proficiency. This group represents the primary end users, providing data on their practical needs and study habits.
- **Both**: individuals who both teach and study, offering a dual perspective on the application’s utility.
- **Other**: individuals who do not fall into the previous categories but use their voice professionally (e.g., actors, speakers, or speech therapists).

5.3 Methodology

The study employed a quantitative survey methodology with targeted qualitative components to gather comprehensive data on user habits and feature preferences. The questionnaire, titled ‘Abitudini di Riscaldamento Vocale e Supporto Tecnologico (Vocal Warm-up Habits and Technological Support)’, was distributed digitally via Google Forms.

The survey design followed a structured logic flow divided into four key areas:

[24]For more information, see: https://www.voiceevolutioninstitute.it/institute

- **User profiling and demographics**: initial questions categorized participants by their professional role and years of experience.
- **Current habits and pain points analysis**: this section examined the current state of vocal training. Participants were asked about:
  - Frequency: how often participants engage in vocal exercises.
  - Tools: the use of traditional instruments versus digital aids (e.g. YouTube, apps and audio files).
  - Vocal health: a specific inquiry into past vocal injuries or chronic fatigue was included to assess the need for safe, guided protocols.
- **Feature prioritization (Likert scale)**: to validate the application's functional roadmap, users were presented with a list of potential features and asked to rate their importance on a 5-point Likert scale (1 = not important, 5 = essential). This quantitative method provided objective data with which to distinguish 'must-have' features from secondary ones.
- **Technical and pedagogical context**: some questions addressed technical feasibility and workflow integration. Participants were asked about the contexts in which they would use such an app, the technical barriers they face during remote teaching (specifically regarding latency), and their current methods for sharing and organizing pedagogical materials.

5.4 Tools and setup

The survey was designed and hosted on Google Forms to ensure cross-platform accessibility without requiring user registration. Data collection was managed through the Google Workspace ecosystem, with responses automatically aggregated into Google Sheets for statistical processing. Charts and frequency distributions were generated using the Python library Matplotlib for the quantitative analysis.

5.5 Survey results

The following section presents the quantitative data collected from the 52 respondents.

5.5.1 User profiling and demographics

What is your primary role in using your voice?

The survey revealed a fairly homogeneous distribution of professional backgrounds among the participants. As shown in the figure below (Figure 26), the majority of respondents identified as singers and voice teachers, with a few exceptions. Those working in voice-intensive professions, such as speech therapists and other teachers, also perform voice warm-ups and participated in the survey.

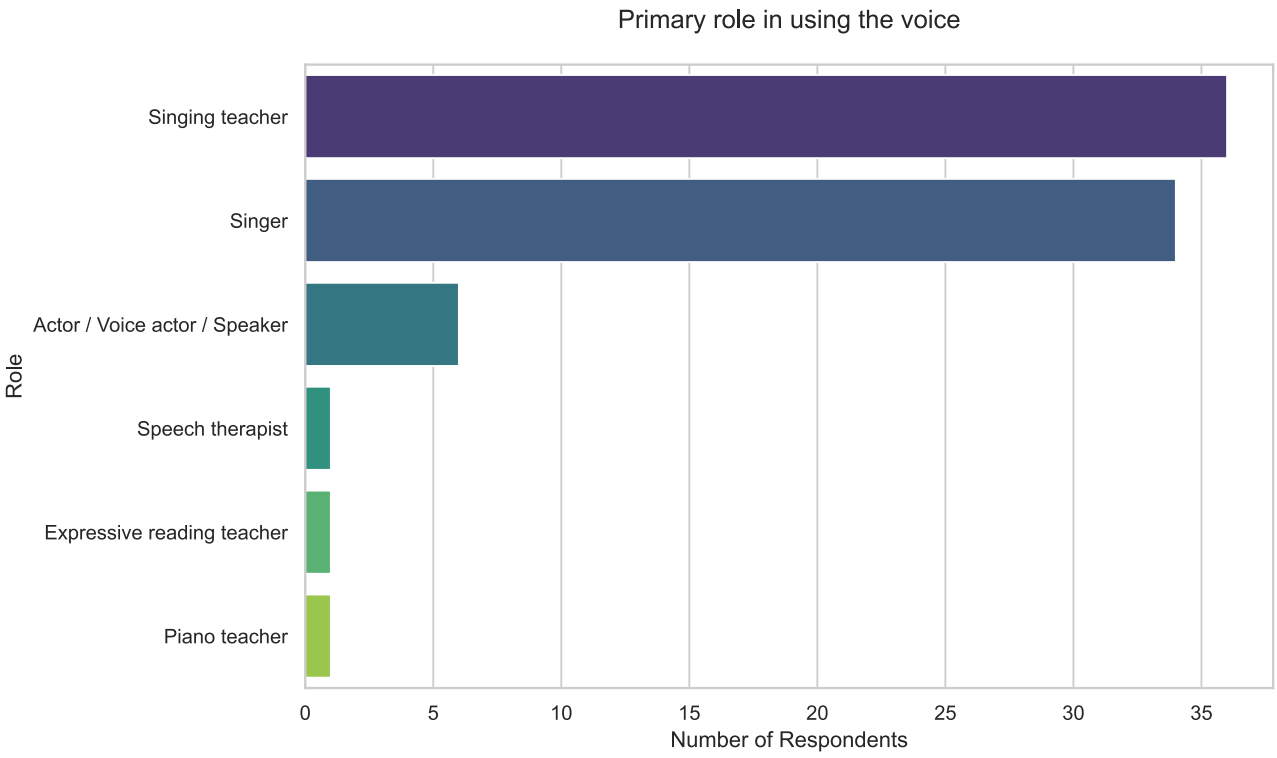


Figure 26: Distribution of participants' primary roles in voice usage among the 52 respondents.

How many years have you been studying or working with the voice?

The data regarding years of experience (Figure 27) indicates a high level of expertise among the respondents. Over 70% of participants have more than

10 years' experience, while a further 17.3% have between 5 and 10 years' experience. This suggests that the feedback was provided by individuals with a deep understanding of vocal training requirements and long-term practice habits.

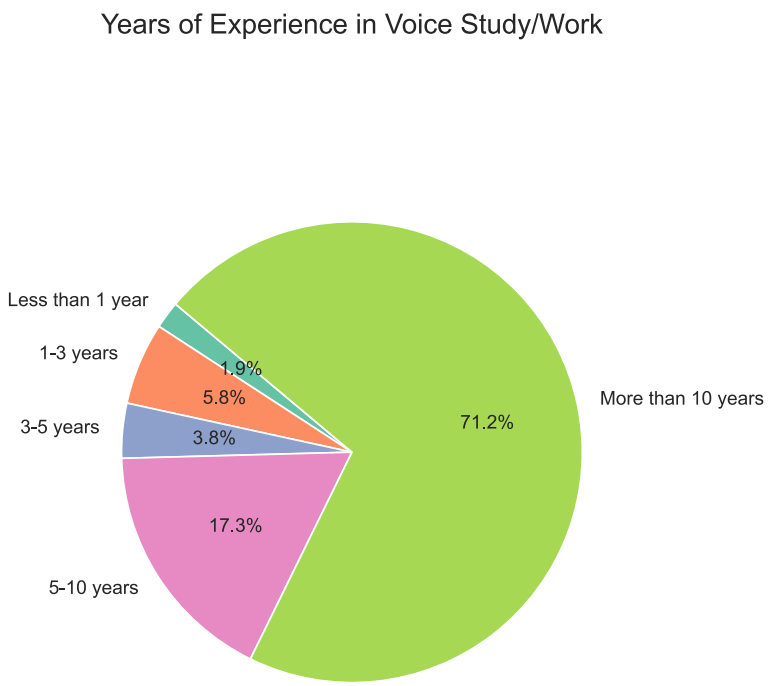


Figure 27: Distribution of participants' years of experience in vocal study or professional work.

5.5.2 Practice habits and pain points analysis

How often do you perform your vocal warm-up?

The data on warm-up frequency (Figure 28) shows that vocal practice is a consistent habit for the majority of respondents. Around 52% of participants warm up more than three times a week. This highlights the potential usefulness of a digital tool that can support such a frequent routine.

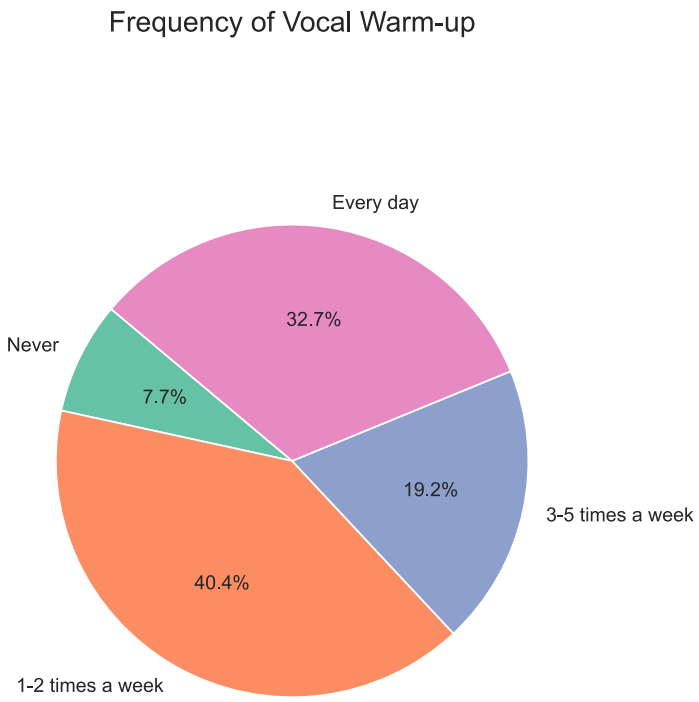


Figure 28: Distribution of the frequency of vocal warm-ups among study participants.

What tools do you use to warm up?

The data regarding the tools used for vocal warm-ups (Figure 29) highlights a significant reliance on digital and pre-recorded resources. While the piano and guitar remain fundamental tools, a large proportion of the sample uses videos, audio files, and voice memos. This confirms the trend towards the use of portable media.

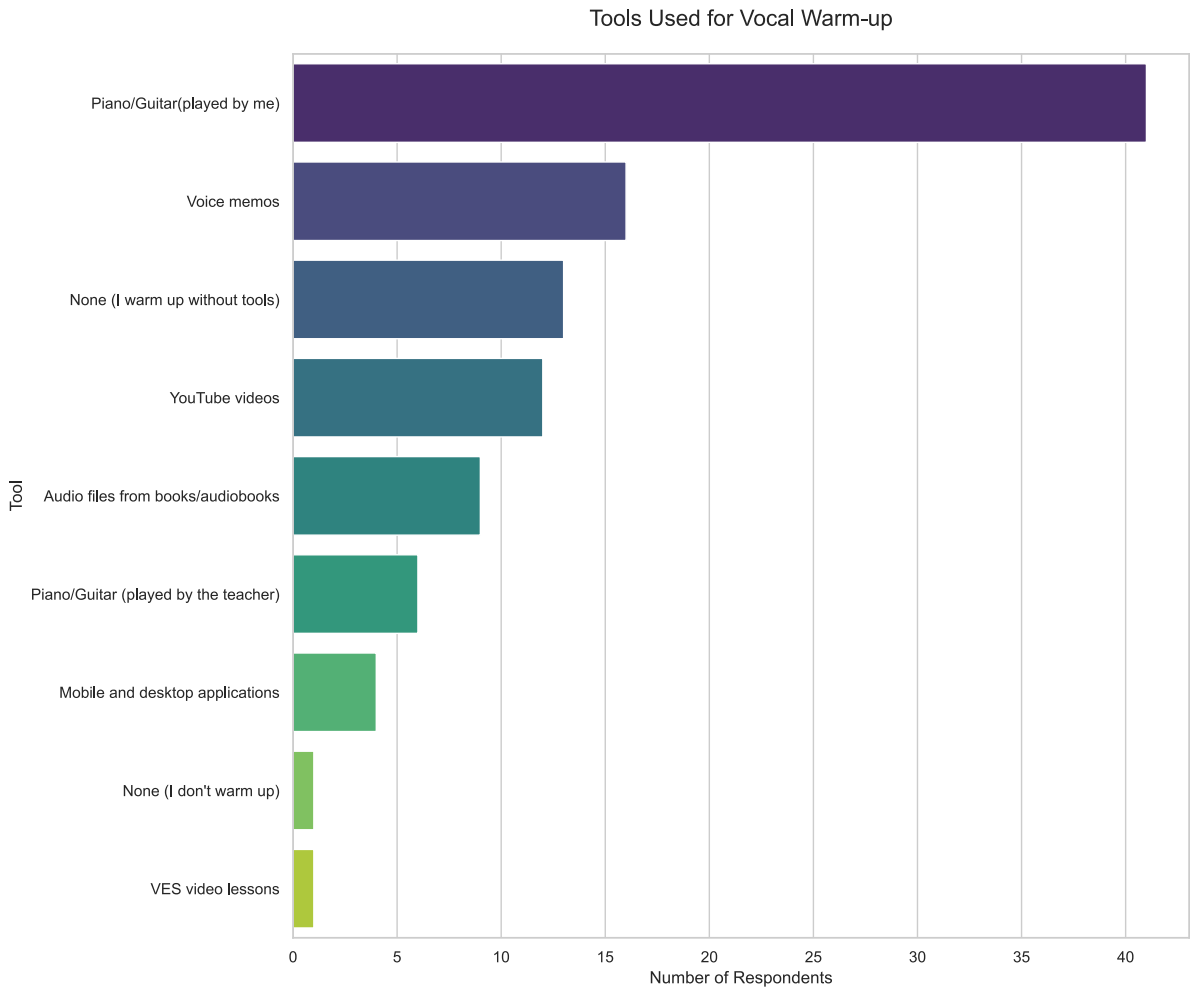


Figure 29: Distribution of the various digital and traditional tools used by participants for vocal warm-ups.

Have you ever suffered from vocal injuries or chronic fatigue?

The data regarding vocal health (Figure 30) shows that a significant portion of the sample (48%) has experienced vocal injuries or chronic fatigue.

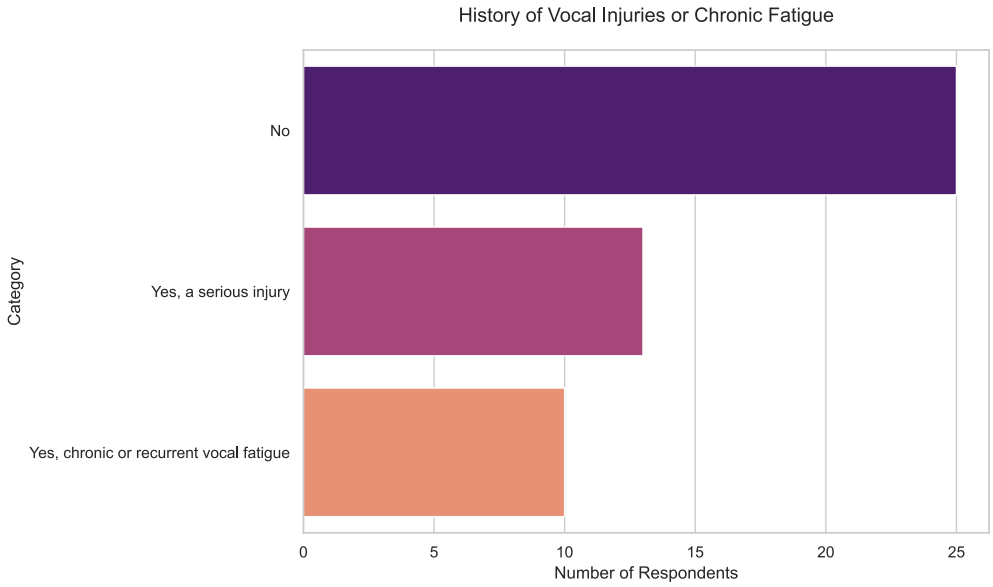


Figure 30: Distribution of participants who have experienced vocal injuries or chronic fatigue.

If applicable, what did you do during your rehabilitation process?

Many respondents who had experienced vocal issues reported that they had followed a structured rehabilitation program. The most common actions included:

- consulting a speech and language therapist or a phoniatrician;
- taking vocal rest and staying hydrated;
- performing targeted cool-downs or rehabilitation exercises, such as vocal stretching, SOVT and Lax Vox[25]
- adjusting their daily vocal load and improving vocal hygiene.

This shows that warm-ups are not only a tool for performance preparation but also a critical component of vocal health management and recovery, being often integrated into therapeutic protocols.

[25] LaxVox is a method of vocal rehabilitation and training based on the use of a medical silicone tube partially immersed in water.

5.5.3 Feature prioritization (Likert scale)

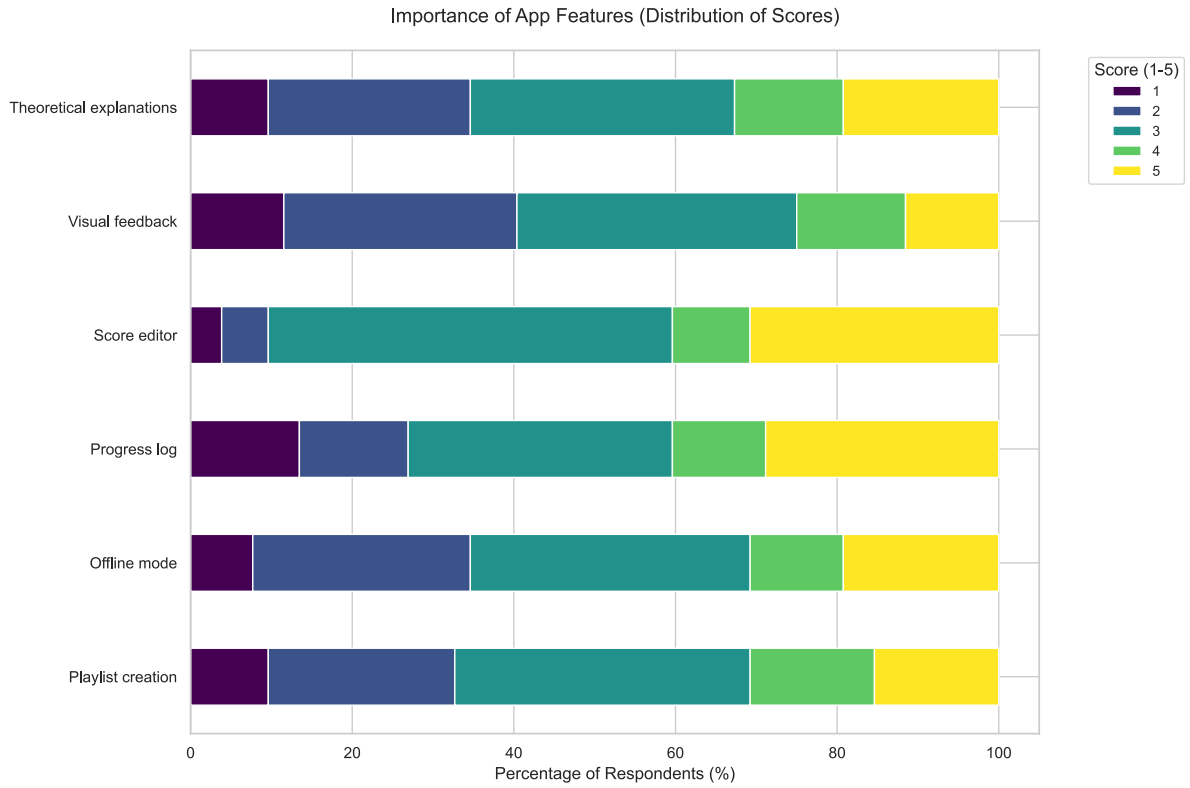


Figure 31: Distribution of the importance assigned to different app features on a 5-point Likert scale.

Based on the poll results (Figure 31), the most requested feature is **Playlist Creation** ($M = 3.58$), suggesting that singers value the ability to sequence their own exercises to suit their specific needs. This is followed by **Offline Mode** ($M = 3.29$), indicating a strong desire for an app that remains functional without a constant internet connection.

Features focused on tracking and technical assistance, such as the **Progress Log** ($M = 3.08$), **Score Editor** ($M = 3.08$), and **Visual Feedback** ($M = 3.04$), all received moderate scores, showing they are considered useful but secondary to the core functionality of the app. Interestingly, **Theoretical Explanations** ($M = 2.85$) was ranked as the least important feature. This

suggests that the target audience likely already possesses the necessary technical knowledge and is primarily looking for a practical tool to support their daily routines rather than an educational platform.

5.5.4 Technical and pedagogical context

In what contexts would you use a vocal warm-up app?

The data regarding the contexts of use (Figure 32) reveals that the primary scenario for using a vocal warm-up app is to assign exercises to students for home study, followed by personal warm-up routines. A significant proportion of respondents also indicated that they would use the app during online lessons. Interestingly, a consistent proportion would also use it during in-person lessons.

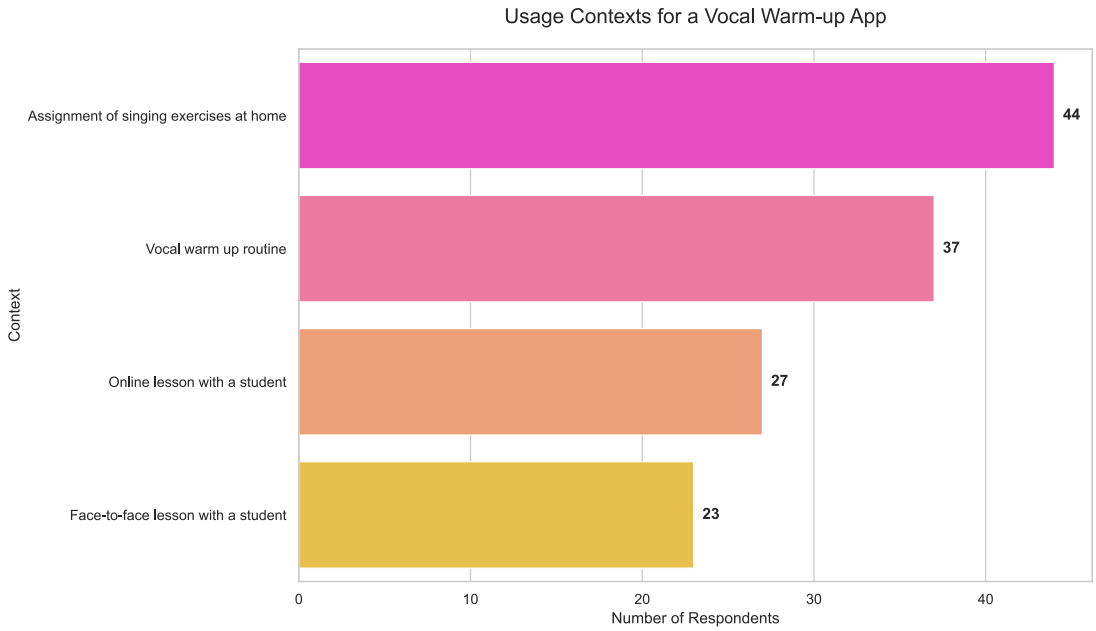


Figure 32: Distribution of the contexts in which participants would use a vocal warm-up app.

What is the main technical difficulty you encounter in online singing lessons?

The data regarding technical difficulties (Figure 33) highlights that audio latency is the most significant barrier in online vocal training. This confirms that the primary challenge for remote teaching is the lack of real-time synchronization between the teacher's accompaniment and the student's voice.

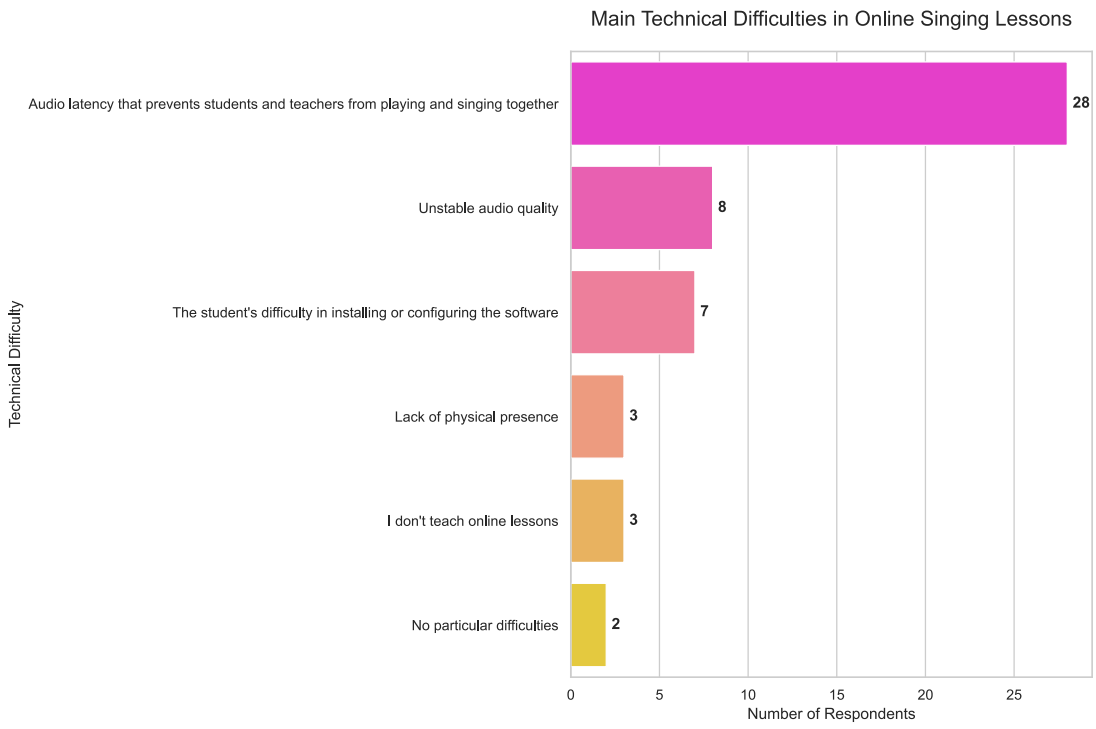


Figure 33: Distribution of the main technical difficulties encountered in online singing lessons.

Have you ever tried low-latency software for musicians (eg, Jamulus, Syncroom)?

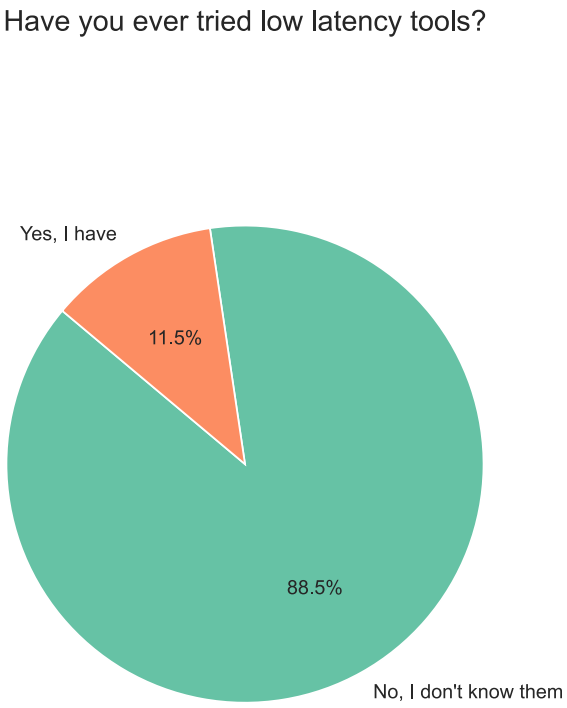


Figure 34: Distribution of participants' previous experience with specialized low-latency tools like Jamulus or Syncroom.

The data regarding low-latency software (Figure 34) shows that the vast majority of participants (88.5%) have never used such tools. This suggests that while latency is recognized as a major problem, specialized technical solutions are not yet part of the standard workflow for most vocal professionals, likely due to their complexity or hardware requirements.

How do you currently share warm-up exercises with your students for studying at home?

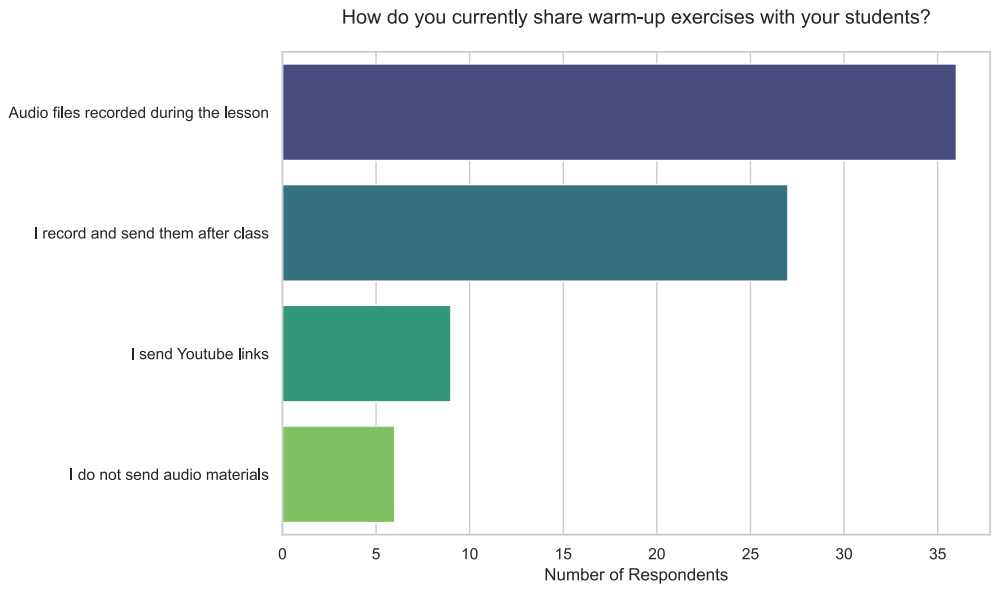


Figure 35: Distribution of the current methods used by participants to share warm-up exercises with students.

Data regarding exercise sharing (Figure 35) shows that the most common method is to record audio files and send them to students. This is followed by sharing links to YouTube videos. These results highlight the fragmented nature of the current workflow, in which teachers must manually manage and distribute files across different platforms. This further justifies the need for a centralized system like VoiceWorm to streamline exercise management.

How do you currently organize your repertoire of vocal exercises?

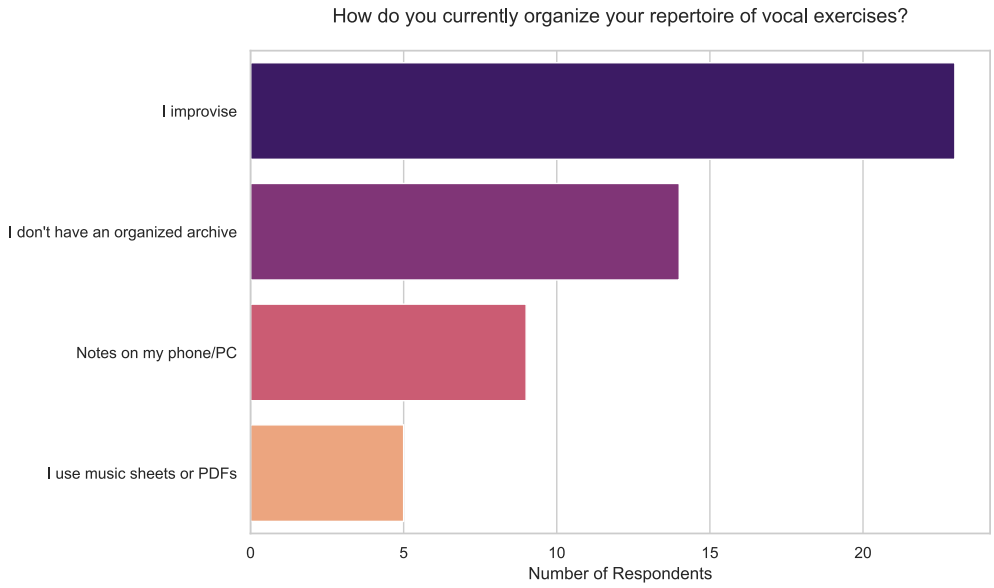


Figure 36: Distribution of the methods used by participants to organize their repertoire of vocal exercises.

Data regarding the organisation of vocal exercises (Figure 36) shows that the majority of respondents do not use a structured system, instead relying on memory. Only a small percentage use physical and digital tools such as music sheets or notes on their devices. This lack of organization highlights the potential value of a digital repository that enables the systematic categorization and retrieval of exercises.

How useful would a digital database be to you for cataloging exercises by educational objectives (eg, “Agility,” “Resonance,” “Control”)?

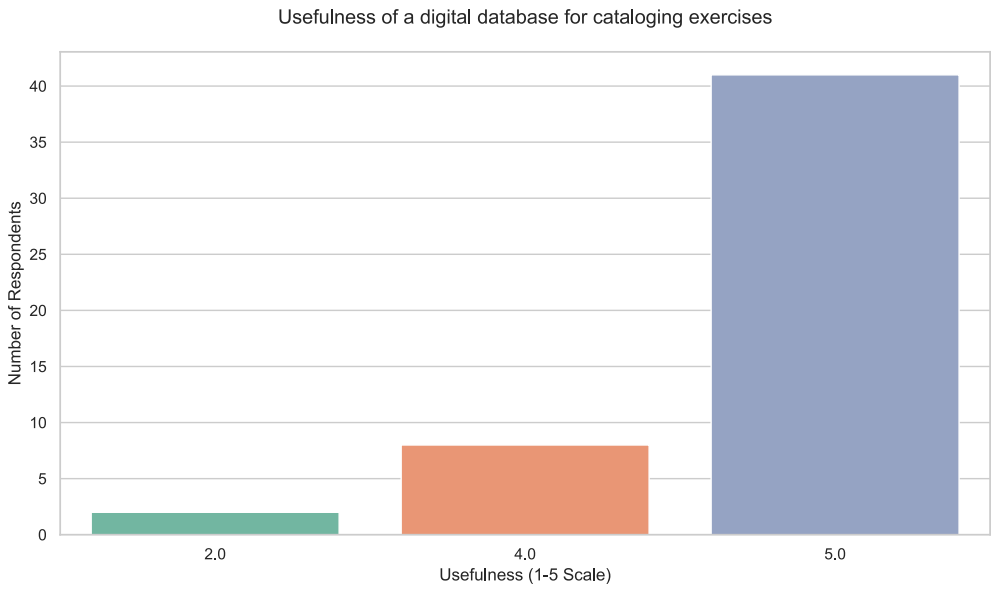


Figure 37: Distribution of the perceived usefulness of a digital database for cataloging exercises by educational objectives.

The data regarding the perceived usefulness of a digital database (Figure 37) shows an overwhelmingly positive response. Over 40 participants rated the utility of such a feature as a 5 on a 5-point scale. This strong consensus validates the core concept of VoiceWorm as a tool for systematic pedagogical organization.

Do you have any additional comments or suggestions on the topic of vocal warm-ups, or on VoiceWorm?

We can finally summarize the qualitative feedback received from the open-ended section of the survey.

- **Customization and categorization**: some participants emphasized the importance of being able to categorize exercises by their pedagogical objective.

- **Importance of warm-ups**: many respondents reiterated that a structured warm-up is essential in a singer's routine.
- **Teacher-student interaction**: one participant expressed a desire for a 'bridge' between teacher and student, whereby the teacher could monitor the student's progress remotely and assign specific exercises privately.
- **Rhythmic accuracy**: one participant noted that maintaining rhythmic structures and grooves is crucial for ear training and stylistic development, and suggested that the app should provide pre-recorded tracks rather than just a synthesized piano sound.

6 Conclusions

This thesis documents the design and development of VoiceWorm, a web-based tool optimized for remote voice teaching. The project emerged from the need to overcome the limitations imposed by audio latency on general-purpose video conferencing platforms, which prevent teachers and students from performing vocal warm-up exercises together. By providing a specialized environment for executing and storing vocal exercises, VoiceWorm aims to enhance the study experience in synchronous and asynchronous contexts.

The proposed solution addressed the latency problem by shifting the synthesis of musical accompaniment directly to the user's device via abc notation and the abcjs library. The application's integration of a collective exercises database, categorized by pedagogical objectives, made the platform a versatile tool for both guided lessons and autonomous practice. A modern architecture based on Vue.js and Cloudflare Workers supported the entire application, ensuring scalability through a serverless approach.

One of the primary challenges addressed was transposing musical exercises to different keys. In asynchronous settings, teachers often rely on pre-recorded tracks that cannot be adjusted to suit the student's specific objectives. VoiceWorm solves this problem by enabling users to create patterns that can be performed in any key and range. The transposition engine was developed by integrating symbolic music representation with a custom-built parser. This required an in-depth understanding of the syntax of musical notation in order to create a robust LL(1) parser capable of handling complex scenarios. This grammar-first approach enabled the creation of logic that respects the rules of music theory during transposition, thereby avoiding errors commonly encountered in naive text manipulation.

The validation of requirements, which was conducted via an online survey of 52 voice professionals, confirmed the effectiveness of the design choices.

The results revealed that latency remains the primary technical challenge, while also demonstrating significant interest in a structured system for categorizing exercises according to pedagogical objectives. Although the current system provides a foundation for remote vocal training, the survey identified several ways in which VoiceWorm could evolve to become a comprehensive vocal development and health monitoring platform, moving beyond its current role as a warm-up tool.

Based on the survey results and the theoretical framework established in previous chapters, the next stage of VoiceWorm's development will focus on advanced acoustic analysis to help teachers deliver more precise guidance. The application will provide objective visual feedback, including a real-time pitch visualizer to monitor intonation accuracy, as well as spectral analysis tools to display formant structures and resonance strategies. By analyzing acoustic parameters, the system could potentially assist with the early detection of vocal irregularities, providing an initial defense against vocal health issues and dysphonia.

Another primary objective is the implementation of an exercise sequencing system, identified as a high-priority feature by users. This functionality would allow teachers to organize individual patterns into cohesive playlists, mirroring the natural progression of a vocal session from warm-ups to athletics without interruption. To further enhance the practice experience, the platform aims to evolve beyond purely synthesized accompaniment by adopting a hybrid audio engine. This would involve the integration of pre-recorded tracks to improve rhythmic dynamics and groove, while simultaneously introducing a client-side recording interface.

In conclusion, VoiceWorm represents a first step in the digitalization of vocal pedagogy. By combining the flexibility of symbolic music notation with the accessibility of modern web technologies, the platform provides an answer to the long-standing issue of latency in online music education. Rather than replacing human guidance, VoiceWorm is designed to serve as a solid technological foundation to support the educational relationship. This paves

the way for an ideal ecosystem in which voice care and digital innovation can coexist to improve the learning experience for singers and teachers alike.

Bibliography

[1] W. M. Hartmann, *Principles of Musical Acoustics*. Springer, 2013, p. 2–4; 227–230.

[2] T. J. Clark, "Harmonizing Voices: Vocal Pedagogy in 21st Century Music Education," DME Thesis, Lynchburg, VA, 2024.

[3] R. Singh and B. Raj, "Human Voice is Unique." [Online]. Available: https://arxiv.org/abs/2506.18182

[4] N. Elliot, J. Sundberg, and P. Gramming, "What happens during vocal warm-up?," *Journal of Voice*, vol. 9, no. 1, 1995, doi: 10.1016/S0892-1997(05)80221-8.

[5] I. R. Titze, "Voice Training and Therapy with a Semi-Occluded Vocal Tract," *Journal of Singing*, vol. 62, no. 4, 2006.

[6] V. V. Ribeiro, L. F. Frigo, G. R. Bastilha, and C. A. Cielo, "Vocal warm-up and cool-down: systematic review," *Revista CEFAC*, vol. 18, no. 6, 2016, doi: 10.1590/1982-0216201618617215.

[7] R. O. Gottliebson, "Efficacy of Cool-Down Exercises In the Practice Regimen of Elite Singers," PhD Dissertation, Cincinnati, OH, 2011.

[8] Y. Lee, M. Oya, T. Kaburagi, S. Hidaka, and T. Nakagawa, "Differences Among Mixed, Chest, and Falsetto Registers: A Multiparametric Study," *Journal of Voice*, 2021, doi: 10.1016/j.jvoice.2020.12.028.

[9] K. Steinhauer, M. M. Klimek, and J. Estill, *The Estill Voice Model: Theory and Translation*. Pittsburgh, PA: Estill Voice International, 2017.

[10] W. Apel, Ed., *Harvard Dictionary of Music*, 2nd ed. Cambridge, MA: Belknap Press of Harvard University Press, 1974.

[11] Wikipedia contributors, “Musical note — Wikipedia, The Free Encyclopedia.” [Online]. Available: https://en.wikipedia.org/wiki/Musical_note

[12] Wikipedia contributors, “Piano key frequencies — Wikipedia, The Free Encyclopedia.” [Online]. Available: https://en.wikipedia.org/wiki/Piano_key_frequencies

[13] “Staff, Notes, and Pitches.” [Online]. Available: https://musictheoryexamreview.weebly.com/staff-notes-and-pitches.html

[14] Wikipedia contributors, “Interval (music).” [Online]. Available: https://en.wikipedia.org/wiki/Interval_(music)

[15] Musicnotes, “How to Read Sheet Music.” [Online]. Available: https://www.musicnotes.com/blog/how-to-read-sheet-music/

[16] D. Ostuni, “GLL parsing - Lecture slides.”

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 1st ed. Addison-Wesley, 1979, p. Chapter 4: Context–Free Grammars, pp. 77–106; Chapter 6: Properties of Context–Free Languages, pp. 125–137.

[18] D. J. Rosenkrantz and R. E. Stearns, “Properties of Deterministic Top-Down Grammars,” *Information and Control*, vol. 17, pp. 226–256, 1970.

[19] Y. Chen and A. R. Safian, “A Review of Online Learning Pedagogy and Vocal Music Education,” *International Journal of Academic Research in Business and Social Sciences*, vol. 14, no. 11, 2024, doi: 10.6007/IJARBSS/v14-i11/23561.

[20] J. Nix, D. Meyer, R. Scherer, and D. Michael, “Practical Science in the Studio, Part 2: “Low-Tech” Strategies,” *Journal of Singing*, vol. 77, no. 4, 2021.

[21] Fiveable, “Latency and Buffer Settings | Music Production and Recording Class Notes.” [Online]. Available: https://fiveable.me/

music-production-and-recording/unit-5/latency-buffer-settings/study-guide/hcGJcxVrBVP0Vr9G

[22] K. Tsioutas, G. Xylomenos, and I. Doumanis, “Impact of Audio Delay and Quality in Network Music Performance,” *Future Internet*, vol. 17, no. 8, 2025, doi: 10.3390/fi17080337.

[23] K. Murdaugh, J. Bainac Hausknecht, and C. T. Herbst, “In-Person or Virtual? – Assessing the Impact of COVID-19 on the Teaching Habits of Voice Pedagogues,” *Journal of Voice*, vol. 36, no. 5, 2022, doi: 10.1016/j.jvoice.2020.08.027.

[24] Zoom Community, “Original sound 'On' function not working.” [Online]. Available: https://community.zoom.com/meetings-2/original-sound-on-function-not-working-21386?postid=24430#post24430

[25] J. C. Wu, “Empowering Musicians: Innovating Virtual Ensemble Concert Music with Networked Audio Technology,” *Virtual Worlds*, vol. 4, no. 1, 2025, doi: 10.3390/virtualworlds4010009.

[26] Yamaha Corporation, “SYNCROOM Manual (Desktop Version) Ver.2.0.” [Online]. Available: https://syncroom.yamaha.com/global/v2/play/manual/index_pc.html

[27] P. Pšenák and M. Tibenský, “The usage of Vue JS framework for web application creation,” *Mesterséges Intelligencia*, vol. 2, no. 2, pp. 61–72, 2020, doi: 10.35406/MI.2020.2.61.

[28] World Wide Web Consortium, “World Wide Web Consortium (W3C).” Accessed: May 20, 2024. [Online]. Available: https://www.w3.org/

[29] W3Schools, “JavaScript HTML DOM.” Accessed: Feb. 11, 2026. [Online]. Available: https://www.w3schools.com/js/js_htmldom.asp

[30] Mozilla Developer Network, “HTML: HyperText Markup Language.” Accessed: Feb. 11, 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML

[31] Wikipedia contributors, “HTML — Wikipedia, The Free Encyclopedia.” Accessed: Feb. 11, 2026. [Online]. Available: https://en.wikipedia.org/wiki/HTML

[32] W3Schools, “CSS Tutorial.” Accessed: Feb. 12, 2026. [Online]. Available: https://www.w3schools.com/css/

[33] Wikipedia contributors, “CSS — Wikipedia, The Free Encyclopedia.” Accessed: Feb. 12, 2026. [Online]. Available: https://en.wikipedia.org/wiki/CSS

[34] Wikipedia contributors, “JavaScript — Wikipedia, The Free Encyclopedia.” Accessed: Feb. 12, 2026. [Online]. Available: https://en.wikipedia.org/wiki/JavaScript

[35] Google Developers, “Introduction to JavaScript.” Accessed: Feb. 12, 2026. [Online]. Available: https://web.dev/learn/javascript/introduction

[36] Ecma International, “The JSON Data Interchange Syntax,” Standard ECMA-404, Dec. 2017. [Online]. Available: https://ecma-international.org/publications-and-standards/standards/ecma-404/

[37] Mozilla Developer Network, “Working with JSON.” Accessed: Feb. 12, 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/JSON

[38] B. L. Sturm and O. Ben-Tal, “folk-rnn: Machine Learning Music Generation.” [Online]. Available: https://folkrnn.org/

[39] C. Walshaw, “About abc notation.” [Online]. Available: https://abcnotation.com/about

[40] P. Rosen and G. Dyke, “abcjs: javascript for rendering abc music notation.” [Online]. Available: https://www.abcjs.net/#why

[41] P. Rosen and G. Dyke, “Purpose — abcjs Documentation.” [Online]. Available: https://docs.abcjs.net/overview/purpose

[42] J. Juviler, “The Bulma CSS Framework: What It Is and How To Get Started.” [Online]. Available: https://blog.hubspot.com/website/bulma-css

[43] J. Thomas, “Bulma: Free, open source, and modern CSS framework based on Flexbox.” [Online]. Available: https://bulma.io/documentation/

[44] M. Chapple, “SQL Fundamentals.” [Online]. Available: https://web.archive.org/web/20090125120638/http://databases.about.com/od/sql/a/sqlfundamentals.htm

[45] SQLite Development Team, “About SQLite.” [Online]. Available: https://www.sqlite.org/about.html

[46] Cloudflare, “How Workers works.” Accessed: Feb. 08, 2026. [Online]. Available: https://developers.cloudflare.com/workers/reference/how-workers-works/

[47] The V8 Project, “V8 Documentation.” Accessed: Feb. 08, 2026. [Online]. Available: https://v8.dev/docs

[48] Cloudflare, “Cloudflare Workers Documentation.” Accessed: Feb. 08, 2026. [Online]. Available: https://developers.cloudflare.com/workers/

[49] A. Kshirsagar, “Routing in Web Development.” [Online]. Available: https://medium.com/@abhikshirsagar1999/routing-in-web-development-f3e5c75c49c5

[50] Y. Wada and Hono Contributors, “Hono Documentation.” Accessed: Feb. 08, 2026. [Online]. Available: https://hono.dev/docs/

[51] Arpeggio Music, “Warm Me Up.” Accessed: Feb. 10, 2026. [Online]. Available: https://www.warm-me-up.com/

[52] Virtually Vocal, “Online Vocal Exercises.” Accessed: Feb. 10, 2026. [Online]. Available: https://www.virtuallyvocal.com/Home/Online-vocal-exercises

[53] ToneGym, “Vocal Warmup Tool.” Accessed: Feb. 10, 2026. [Online]. Available: https://www.tonegym.co/tool/item?id=vocal-warmup

[54] Muse Group, “MuseScore: The world's most popular notation app.” Accessed: Feb. 11, 2026. [Online]. Available: https://musescore.org/en

[55] PhonicScore, “OpenSheetMusicDisplay (OSMD).” Accessed: Feb. 11, 2026. [Online]. Available: https://opensheetmusicdisplay.org/

[56] RISM Digital Center, “Verovio: Music Notation Engraving Library.” Accessed: Feb. 11, 2026. [Online]. Available: https://www.verovio.org/index.xhtml

[57] L. Pugin, R. Zitellini, and P. Roland, “Verovio: A Library for Engraving MEI Music Notation into SVG,” in *Proceedings of the 15th International Society for Music Information Retrieval Conference*, 2014, pp. 107–112. [Online]. Available: https://archives.ismir.net/ismir2014/paper/000221.pdf

[58] Flat, “Flat: Create sheet music online.” Accessed: Feb. 11, 2026. [Online]. Available: https://flat.io/

[59] scorio GmbH, “scorio: Write and play music online.” Accessed: Feb. 11, 2026. [Online]. Available: https://www.scorio.com/

[60] J. Dvorak, “abc-notation-transposition.” Accessed: Feb. 11, 2026. [Online]. Available: https://github.com/dvorakjt/abc-notation-transposition

[61] Wikipedia contributors, “PBKDF2 — Wikipedia, The Free Encyclopedia.” [Online]. Available: https://en.wikipedia.org/wiki/PBKDF2

[62] F. Skokan, “jose: JSON Object Signing and Encryption.” [Online]. Available: https://www.npmjs.com/package/jose

Acknowledgements

I would like to express my sincere gratitude to everyone who contributed to the realization of this thesis and the development of the VoiceWorm project.

In particular, I would like to thank my advisor, Dr. Giovanni Delnevo, and my co-advisors, Dr. Kelvin Oluwada Milare Obuneme Olaiya and Dr. Manuel Andruccioli, for their guidance and for enabling me to pursue this research topic. This opportunity has allowed me to combine my passion for music with my interest in web development, finally bridging the gap between my artistic inclinations and my aspiration to work in a scientific environment.

I would also like to extend my heartfelt thanks to VES - Voice Evolution Institute for being my family during my ten years of vocal training and teaching, and for their unwavering support throughout this project.

Finally, my deepest thanks go to my loved ones for their constant encouragement. They have always been there for me, supporting every path I have decided to take.