Music Education and Entertainment Simulation System: A Digital Guitar Instrument Controller with an Integrated Web Application

ICON College of Technology and Management

Department of Information Technology

Bachelor's Degree Dissertation Project | BSc Hons

Accredited by Falmouth University

Tivadar Debnar

2022-2023

Playing Alone on the Road *(EclecticMusicAtlanta, 2022)*

"I dream for instruments obedient to my thought and which, with their contribution of a whole new world of unsuspected sounds, will lend themselves to the exigencies of my inner rhythm."

(Edgard, 1917)

Table of Contents

1. INTRODUCTION

Earthly creatures are very fortunate. Extremely few places in the vast vacuum-filled Universe have suitable mediums that support audio signals to travel. However, here on Earth, sound vibrations can move through the atmosphere, providing information about our environment. As a result, mammalian evolution adapted to transform soundwaves into electrical signals, genetically engineering us to detect sounds. Hearing sounds increases our survival chances by identifying danger outside our visual zone and extending our communication channels.

Even though humans are not the only species communicating by creating sounds, we discovered a way of self-expression that conveyed a broader spectrum of emotional range beyond mere spoken words: music. From as early as 40000 years ago, music has played an essential part in our everyday life. Our innate musicality drove us to experiment with new sound ranges, inventing the primary types of instruments. Ideophones (clapping, bells), membranophones (drums), aerophones (flute), and most importantly, chordophones (harp, guitar).

Although the exact origin of the modern guitar is debated, the instrument is already mentioned in the Bible, and it can be traced back to the Greek *κιθάρα* (kithara) and Arabic قيثارة (qitharah) words. By the 17th century, it became popular among amateurs. With the advent of the jazz age, the electric guitar's success elevated its status to become the instrument of virtuosos and rock stars. However, the evolution of the guitar continues as digital technology integrates with musical skills. *This project's goal is to bring digital technology, musical entertainment, and education under the same roof as an intelligent guitar console system*.

1.1 INSPIRATION

Holding Gabriel's Guitar Hero in my hand, I was surprised that my years of actual guitar practice couldn't beat a skilled gamer like him. It became evident that his virtual guitar expertise had refined his abilities. Naturally, I couldn't resist returning the favour and handing over a real guitar to him. I quickly taught him a simple melody from his favourites, and to my amazement, he mastered it decently in our short session. So I asked him:

"Why do you waste your time practising an imaginary instrument? You'd become a great guitarist by now."

"You'll see me playing when they invent real guitars for game consoles." He answered with a smirk.

Then, I pondered why talented people invest time playing on *unauthentic* consoles. If I could create a lightweight device resembling a real guitar, I could develop a freely accessible, vendor-independent and educational online application. I am confident it would be at least as attractive an entertainment option as playing Guitar Hero. Well, the time has come to wipe off the sneer from Gabriel's face; he will be the first to play it.

1.2. AIM AND OBJECTIVES

Aim

To offer a comprehensive and unique simulated music experience, providing a hardware device with a naturalistic guitar layout and software to learn and play the instrument. The controller device should have a minimalistic design to be affordable to a broader range of players. As the application will be online, the users are not required to own or buy any software licence. Users connect the device to any computer via a USB cable and play on the online platform. The software also helps in learning the instrument from the fundamentals; hence, users should be able to create an account to track their progress. Because we focus on playfully learning guitar riffs, chords and songs, it will be baptised *RiffMaster*.

Objective-1

Design and develop a digital guitar controller with a layout that accurately simulates the instrument's mechanism. The controller's look, dimensions, and operability must be of a guitar, while the materials used may differ and be similar to a mock guitar. However, the guitar controller's neck and the frets' distances must translate to a real guitar's exact proportions to enhance the players' precision in muscle memory and help them gain an easily transferable skill.

Objective-2

Six strings or strum bars may activate notes, and as guitars are polyphonic instruments, one or more strings may be played simultaneously. On activity, the device should communicate the positions of the active frets. The activated notes must be transmitted using a predefined, custom protocol through a USB port. The hardware must be safe to use and in accordance with safety regulations.

Objective-3

Design and develop an application that accepts, detects and listens to user inputs from the device through a USB port without keyboard interruption. Capture device inputs with event listeners and structure state to make note information easily accessible. Optimise the application to accept and process parallel string actions.

Objective-4

Build a user-friendly web application featuring signup and login options. Allow the users to play the device and listen to the generated music. Editing functionalities should also be available, such as music creation, chords, composing, and saving. Lastly, users can play a real-time music game with score feedback on a restricted number of sample songs to test the prototype and the application.

Objective-5

Develop and deploy a backend application that reflects a simple startup's real-world business model for prototyping the web application. Our business model must be constrained, focusing on hardware and software prototyping rather than business implementation. The backend should communicate to a database and store essential information, such as user profiles and tablatures.

1.3. REQUIREMENTS AND SPECIFICATIONS

Requirements are as crucial for large-scale projects as for smaller or individual ones because they concisely and unambiguously capture the project's parameters. "*Understanding user requirements is an integral part of information systems design and is critical to the success of interactive systems*" (Maguire and Bevan, 2002). User and system requirements detail how the user interacts with the system and can be used to test the project.

1. Prerequisites
   - 1.1. Users must have access to a console,
   - 1.2. Users must have a desktop or laptop with a USB-A socket,
   - 1.3. USB-A/B cable connects the host to the console,
   - 1.4. Users must have internet access,
   - 1.5. The application will be publicly accessible.

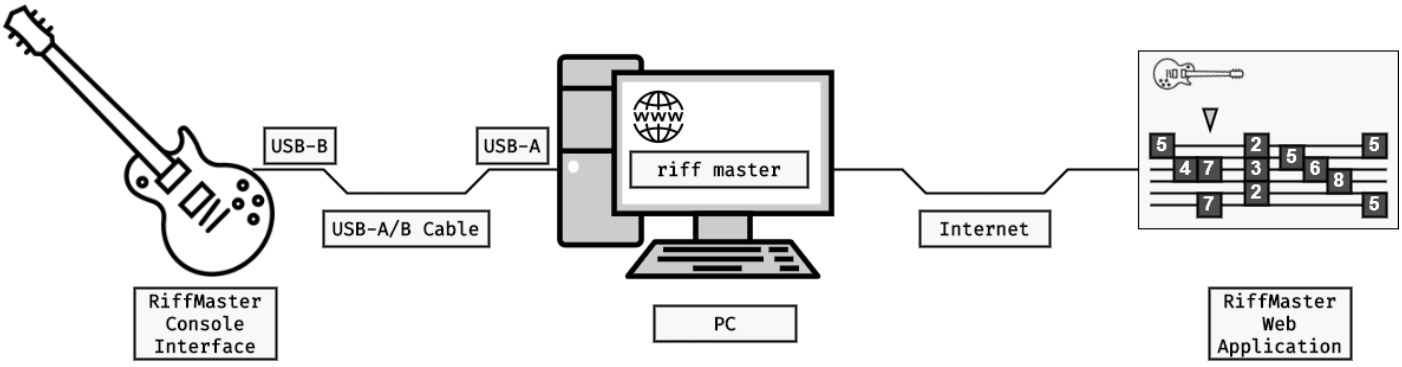


Figure-1 Concept (Appendix/Concept.drawio)

2. Hardware
   - 2.1. Guitar consoles have 20 fret buttons in six rows, strum bar switches, and a power toggle,
   - 2.2. The users interact with the website with traditional mouse and keyboard inputs,
   - 2.3. The users interact with game functionalities with the guitar console,
   - 2.4. A 5V USB connection powers the console,
   - 2.5. Input transactions can be switched off.

3. Software
   - 3.1. The application is a web-based GUI,
   - 3.2. The application is compatible with the guitar console,
   - 3.3. The software runs on major browser vendors (Edge, Firefox, Chrome, Safari, Explorer),
   - 3.4. The application uses dark themes and bright buttons for accessibility,
   - 3.5. The landing page redirects to signup and login.

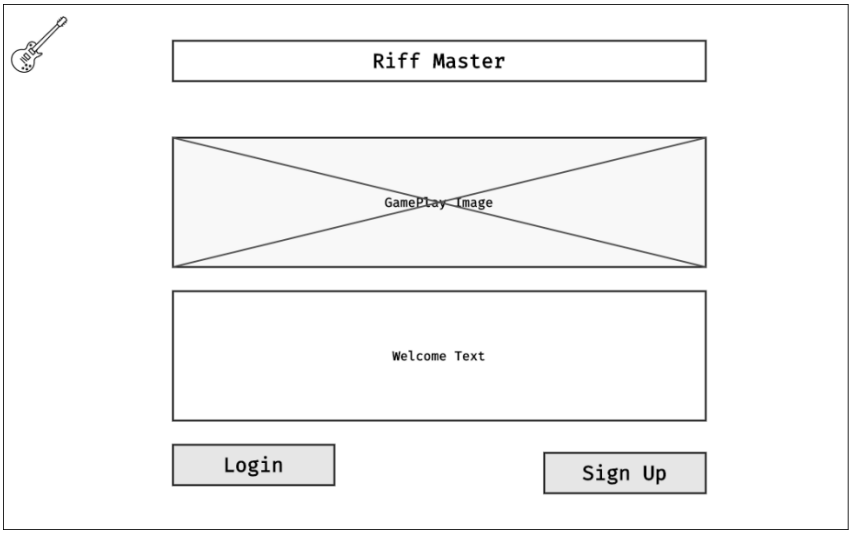


Figure-2 Redirect (Appendix/Wireframes-Landing_Page.drawio)

4. Authentication
   - 4.1. Users sign in to access the application.
   - 4.2. Users may create a personal account:
     - 4.2.1. The signup button redirects the user to a registration form,
     - 4.2.2. The registration form includes personal details, avatar selection and password confirmation,
     - 4.2.3. Submit buttons are disabled until all fields are filled in,
     - 4.2.4. User submissions are validated,
     - 4.2.5. Highlighted invalid fields provide information on the invalidity,
     - 4.2.6. Input character restrictions align with the field type,
     - 4.2.7. Provide email address validation and filtering,
     - 4.2.8. Disallow weak password construction,
     - 4.2.9. Confirm password fields must match the password inputs,
     - 4.2.10. The terms and conditions check box must be checked to allow submission,
     - 4.2.11. Check box labels have links to the terms and conditions.

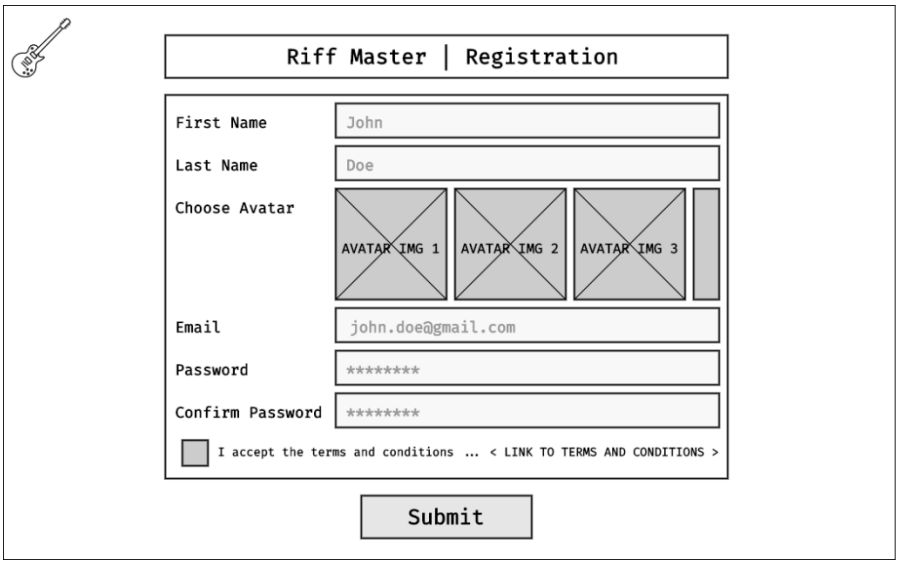


Figure-3 Wireframes Registration (Appendix/Wireframes_Registration.drawio)

4.3. Prevent submitting multiple forms by disabling the submit button after pressing,
4.4. Users receive notifications that an activation email will be sent to the email address,
4.5. Upon clicking the email link, it activates their account and redirects to the login,
4.6. Users with accounts must sign in:
  4.6.1. The login form has email and password fields and a submit button,
  4.6.2. Submit is disabled until both the email and password feature texts,
  4.6.3. Upon submission, the server will match the emails and passwords, sending a response to the website,
  4.6.4. Unmatching emails and passwords deny application access,
  4.6.5. The form locks down after the fifth unsuccessful login attempt,
  4.6.6. Matching passwords allow access to our home page.

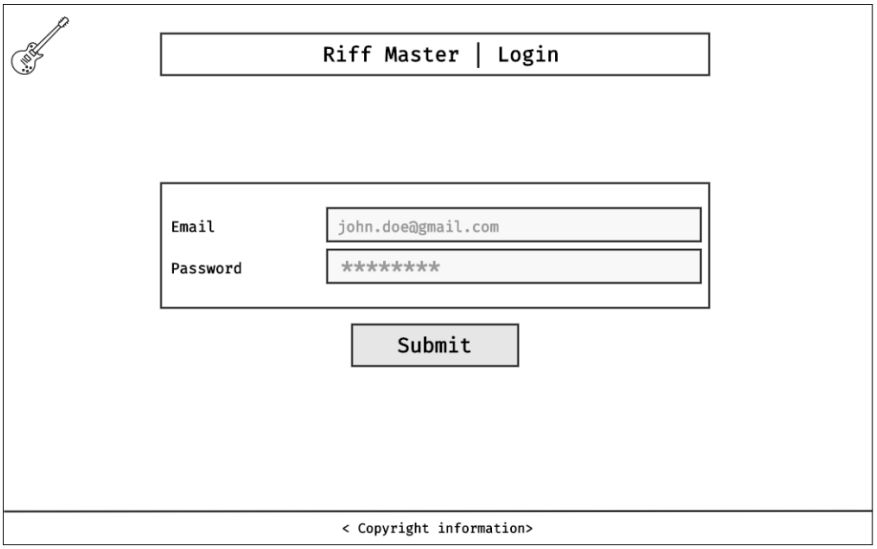


Figure-4 Login Wireframe (Appendix/Wireframe_Login.drawio)

5. Home
5.1. The home page displays profiles, playlists, and a menu: Practice, Jam, Play, Compose, Chords and Sign out,
5.2. Profile Information
  5.2.1. Profiles include the users' avatars and names,
  5.2.2. Profiles display scores, achievements and rankings,
  5.2.3. Profiles display lastly played songs in chronological order,
  5.2.4. Song information includes authors, title and the scores achieved.

5.3. Playlist
  5.3.1. Playlists consist of Album cards,
  5.3.2. Album cards display the album cover, author and title,
  5.3.3. Users are directed to play the chosen song by clicking Album covers.

5.4. Main Menu
  5.4.1. App and options have common colour sequences: yellow, orange, pink, purple, azure, and aquamarine.
5.5. The footer displays copyright information with the current year.

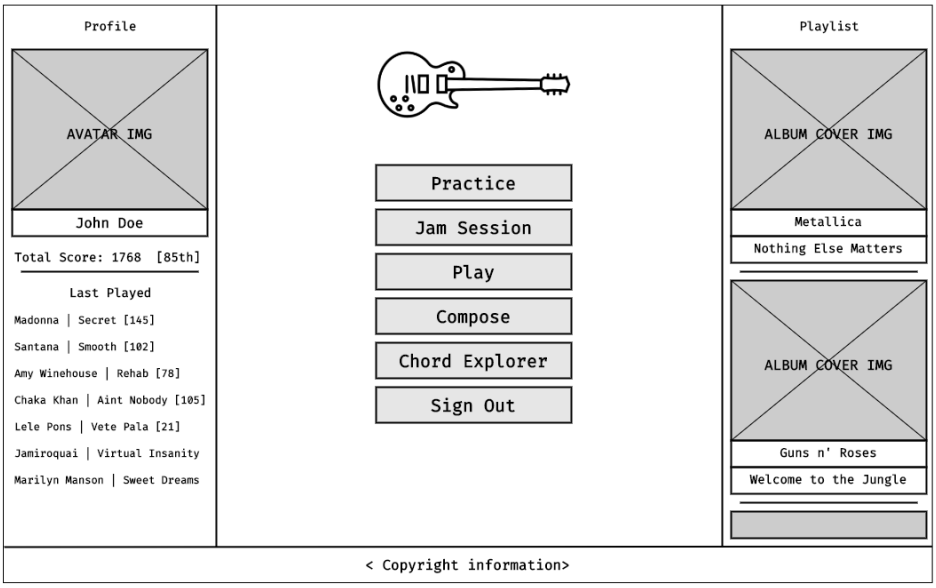


Figure-5 Home Wireframe *(Appendix/Wireframes_Home.drawio)*

6. Play
   - 6.1. The play option displays the playfield,
   - 6.2. Players hit start, and a countdown starts,
   - 6.3. The countdown is followed by music, and game start,
   - 6.4. Players can pause, resume and stop the game,
   - 6.5. Author and title information is displayed in the header.
   - 6.6. Tablature
   - 6.7. One or two tablature lines represent the music, and they have the following elements:
     - 6.7.1. Tablature: Representing the music that is playing with rhythm notation,
     - 6.7.2. Track Highway: Lines represent strings, and the numbers represent frets,
     - 6.7.3. Guitar Board: Virtual representation of the guitar.
   - 6.8. Virtual Guitar Board
     - 6.8.1. Shows frets, finger positions and strum bars,
     - 6.8.2. Active finger positions and strums are highlighted,
     - 6.8.3. Note misses and inaccurate playing are highlighted.

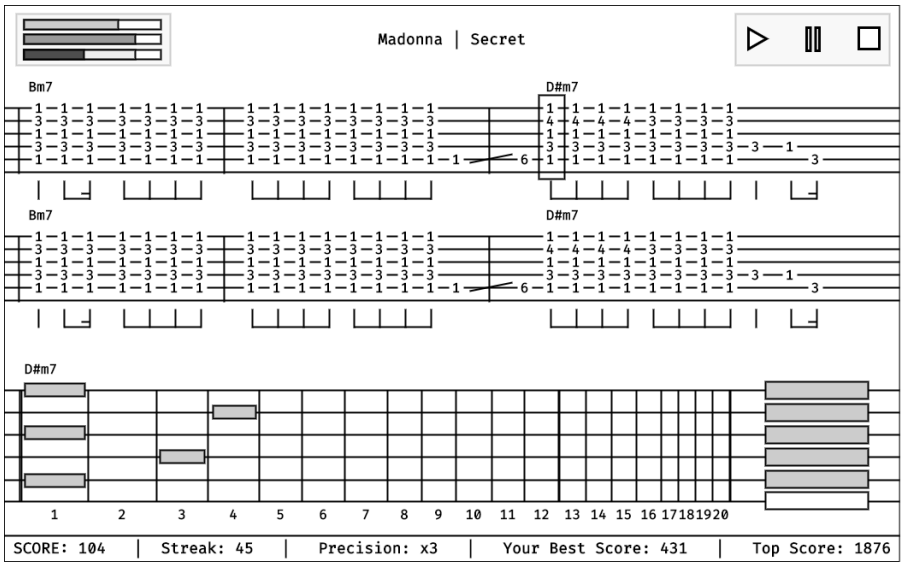


Figure-6 Play Wireframe *(Appendix/Wireframes_Play.drawio)*

7. Scoring
   7.1. The user can see the current score while playing,
   7.2. The application compares actual and expected user actions,
   7.3. Scores calculate accuracy and time precision values.

Note Accuracy	
Regular Note	5
Hammer / Pull Off	7
Diad (2 Notes)	10
Chord (3 Notes)	12
Chord (4 Notes)	15
Chord (5 Notes)	17
Chord (6 Notes)	20

Precision Multipliers	
t > 100ms	x0 (miss)
t ⩽ 100 & t > 75ms	x1
t ⩽ 75 & t > 60ms	x1.2
t ⩽ 60 & t > 55ms	x1.5
t ⩽ 55ms	x2

Streak Multipliers	
10 Notes Streak	x1.5
20 Notes Streak	x2
30 Notes Streak	x3
40 Notes Streak	x4
50 Notes Streak	x5

Figure-7 Scoring (Appendix/Pointing.drawio)

   7.4. Consecutive correct notes count towards streak points,
   7.5. Missed notes are set if outside an interval,
   7.6. Multiple strum strokes reset the stroke timers,
   7.7. Users can see the community's high score.
8. Options
   8.1. The practice option lets the player in the game without scoring,
   8.2. Jam Session is a virtual guitar that responds to strums and keynote presses,
   8.3. Strums produce guitar sounds,
   8.4. Compose Menu option records a replayable console input,
   8.5. Notes are displayed on the tablature when the player presses the record button.
   8.6. Notes are cleared when pressing the delete button.
9. Chords
   9.1. Display an extended list of chords,
   9.2. Chord cards show finger positioning for chords,
   9.3. Chord card frets are relative to position,
   9.4. Standard barre chords notation,
   9.5. Users filter chords with root notes, chord types and bases,
   9.6. Chord cards play audio.

1.3.1 CONSTRAINTS

Scope

"*The scope of the study refers to the parameters under which the study will be operating*" (Simon and Goes, 2013). This project's scope is to ***conceptualise and prototype hardware and software*** components; therefore, the research study extensively focuses on technical exploration, design and delivery of requirements. Consequently, some exciting topics, such as finding the target audience, possible business models, brand design, marketability, mass manufacturing, industrial design, patent collision and litigations, will be investigated briefly but will lack in-depth research.

Limitations

"*Limitations are matters and occurrences that arise in a study which are out of the researcher's control*" (Simon and Goes, 2013). This project is a ***simplified version of a complete video game system*** with a strict timeframe; therefore, several features will be limited or omitted.

- The console will have a coarse functional design lacking the sophistication of an industrial end product,
- The console uses wired communication, as developing Wi-Fi solutions would introduce additional complexity,
- The application will be designed for desktop screen sizes,
- Some user stories are intentionally omitted, such as:
    - Forgotten password retrieval,
    - Deletion of user accounts,
    - Social features: multiplayer mode, messaging, following,
    - Performance statistics and data visualisation,
    - Highly-animated gameplay graphics.

Our hardware and software prototypes aim to present a proof of concept.

1.3.2 MARKETING

Cost

The project's cost details refer to the prototype model building rather than the merchantable end product. Unlike the hardware components, the software solution requires no additional costs (such as acquiring new software licences).

Item	Cost	Item	Cost	Item	Cost
Arduino Uno / Leonardo Kit	42	Guitar Hero Console (optional)	35	Cables (USB, Converter...)	6
Tactile Push Button Switch (120)	24	Toggles and LEDs	12	Components (LEDs, Toggles)	20
Wood for Template	20	Paints (3)	24	Wires and Wiring Tools	40
PCB Printing (optional)	20	CT1	12	Diodes (120) and Resistors	70
Plastic Sheets (2mm and 3mm)	50	Gorilla Glue (3-4)	34	Jumper and Wiring Cables	15
Colored Wiring	15	Carbon Fiber D Tubes (optional)	15	Button Caps	23
Soldering Iron	18	USB B to B with Fixture	5	Strings	16
Tactile Push Button Cap (120)	23	USB A to B	5	Universal PCBs (20)	30

Figure-8 Budgeting *(Appendix/Cost.xlsx)*

The total estimated cost of the ***prototype*** will be around ***£350-500***; however, this price includes tools. The ***retail*** price would be significantly lower in mass production. Depending on the materials' quality, it would cost between ***£70-100***, mainly because of the cheaper equipment and component costs in large-scale productions.

Target Users

A commercialised product's potential target audience would be the following:

- Guitar Hero players who want to extend the difficulty and authenticity of the current games,
- Guitar Hero players who would like to take on instrument skills while being entertained,
- Novice guitarist in need of guided instructions,
- Students and game-oriented talents wanting to explore music while having fun,
- Guitarists that desire silent instruments for practising at night,
- Beginner composers or upcoming bands with a lack of music theory,
- People who want to improve their fine motor skills, rhythm-sense, and musical aptitude.

Marketability

Consequently, the target audience may be a good indicator of the device's success in the market. For instance, MI Digital Guitar raised over £360.000 in funds (*14.10.2022*). This success is promising because our target audience intersects with MI's community. Another relevant market intersection is music gaming. Even though comparing the project to giants like Guitar Hero or Rock Band would seem overly ambitious, it is worth understanding that this ***industry branch has been declining for a decade***.

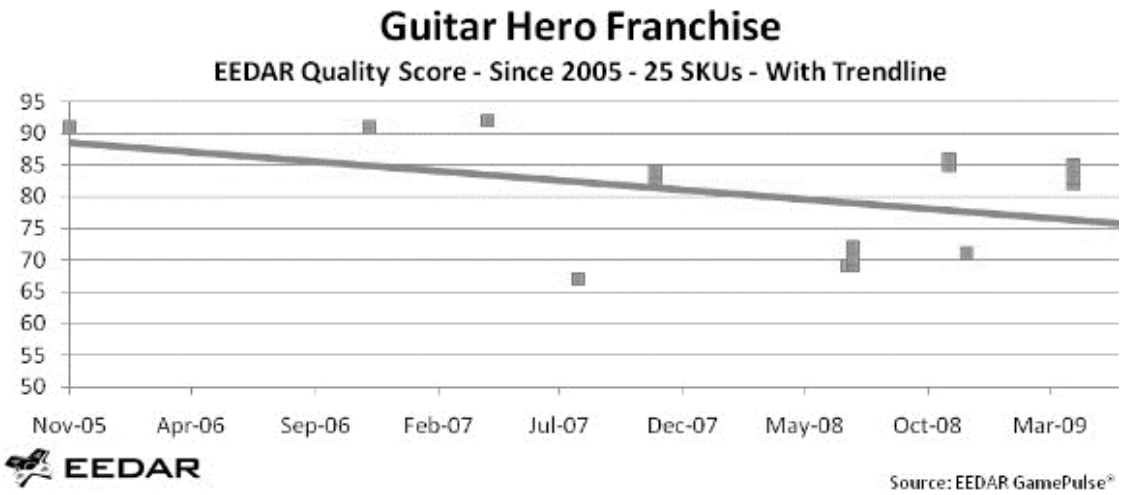


Figure-9 Guitar Hero Trendline (Endgadet, 2022)

The ***oversaturation of the simplified simulation game*** instruments on the market caused a severe decline in the industry. Unfortunately, this means that relying solely on gaming features would leave a narrow margin of opportunity for success. Therefore, the system must emphasise ***authentic instrument digitalisation and build a social community*** rather than jump on the train of the existing gaming schemas.

1.3.3 FEASIBILITY

"*A feasibility study evaluates the project's potential for success; therefore, perceived objectivity is an important factor in the credibility of the study for potential investors and lending institutions*" (Ibrahim R, 2022). And sticking to the gaming industry would navigate the project towards dangerously thin ice. A more promising implementation of RiffMaster would be to build it around a community of users who appreciate the games' educational and social impact.

As digital interactions and education become ever more convenient and natural for our modern society, it opens unchartered territories. Such territory is the online fandom of music enthusiasts, composers and amateur players. Therefore, a feature-rich social web platform that integrates the product would increase the feasibility of the entire project and be a novelty.

Players could organise online competitions, play simultaneously, publish compositions, like others' compositions, follow each other, group in bands, collaborate, teach, or post music instructions. ***Riffmaster has the potential to be impactful only as a comprehensive social and educational gaming platform***.

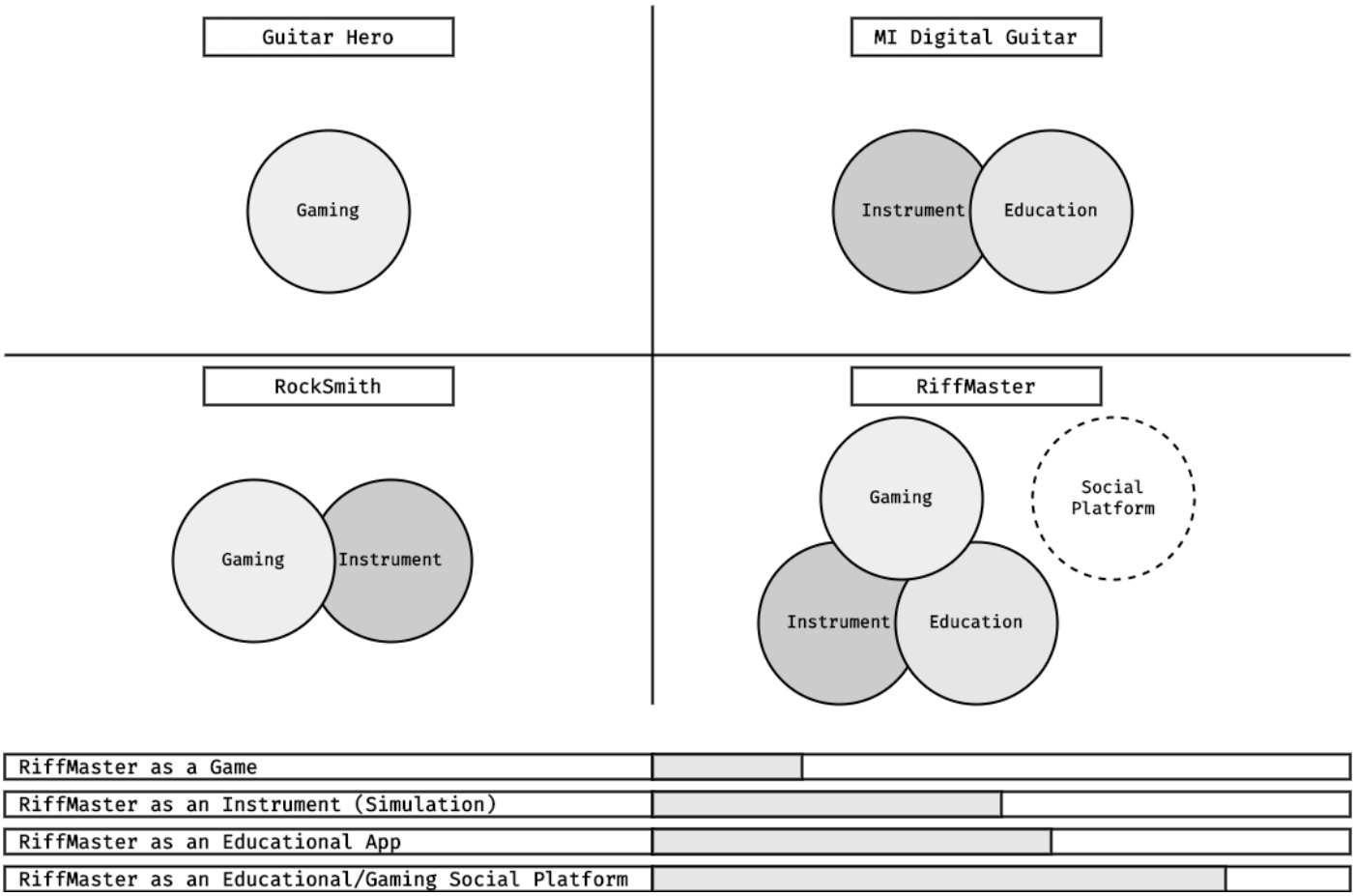


Figure-10 Feasibility *(Appendix/Feasibility-Diagram.drawio)*

1.3.4 RISK MANAGEMENT

Every hardware or software project is inherently risky, and those risks must be identified and addressed. "*To identify risks, the following techniques can be used: brainstorming, work breakdown analysis, risk breakdown structure, checklists, among others*" (Menezes Jr and Gusmão, 2013). Thus, we will collect, identify and propose actions for the project in the following table:

#	Risk	Description	Probability	Severity	Action
1	Scope Variation	Addition of unforeseen features.	Occasional	Marginal	Only add additional features if every requires met.
2	Optimistic Schedule	Bad estimation of task executions.	Probable	Critical	Estimate optimistic and pessimistic schedule
3	Facilities	Hardware parts are delayed or unavailable.	Remote	Critical	Order hardware part in advance of the assembly schedule.
4	Finances	Higher hardware or software cost as expected.	Remote	Moderate	Create a hardware cost table in advance.
5	Data Loss	Loss of project data or source code or documentation.	Occasional	Marginal	Frequent Data Backup (Flash Drive / Git Account)
6	Legal	Legislation issues or patent collision.	Improbable	Critical	Check existing patents and technologies.
7	QA	Failure of Quality Assurance, bugs.	Remote	Critical	TDD, code reviews, automated testing.
8	Analysis	Incomplete or missing requirement details.	Remote	Moderate	Review Requirements
9	Value	Inadaquate value analysis to measure progress.	Remote	Marginal	The project is not intended for commertial use in this phase.
10	Safety	Health and Safety requirements	Remote	Catastrophic	Review and adhere electrical safety practices.
11	Development	Developing inaccurate user interface.	Occasional	Moderate	Prototyping, peer review and user acceptance testing
12	Difficulty	Development technically too difficult.	Remote	Moderate	Technical analysis before project approval.
13	Licence	Software Licence management and expiry.	Remote	Marginal	Licenses recently reviewed

Figure-11 Risk Assessment *(Appendix/Risk-Assessment.xlsx)*

For this project, ***scheduling*** is the risk with the highest probability and severity; therefore, the design, experimentation and development phase will ***start as early as possible*** to avoid delays in delivering the hardware or any software features.

2. PROJECT MANAGEMENT

We can approach the workflow of our project tasks in several ways. We can select the best solution for our needs by examining the relevant software development methodologies. After identifying the chosen method, we can create a detailed timeline of the project development steps.

2.1. METHODOLOGIES

Methodologies can be divided into linear and iterative categories, and both approaches have distinct benefits and limitations, which can affect the project's outcome. When selecting our development method, we must remember that this project consists of hardware and software components, and we may use a mixed methodology.

2.1.1. LINEAR METHODOLOGIES

When engineers discuss sequential (linear) software development, they essentially refer to the Waterfall methodology, which dates back to the 1960s manufacturing. "*Waterfall Model predominately emphasises on the freezing of requirement specifications or the high-level design very early in the development life-cycle, prior to engaging in more thorough design and implementation work*" (Van, 2017). This inflexibility is often problematic because Waterfall is unidirectional, and the projects cannot be modified after the specification and design.

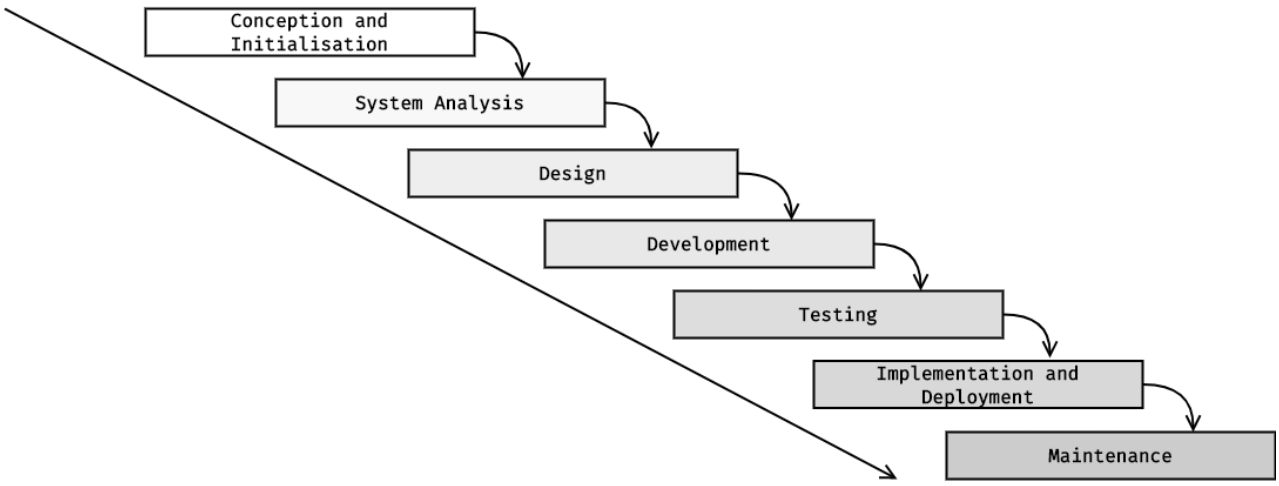


Figure-12 Waterfall *(Appendix/Waterfall.drawio)*

Waterfall assumes that project requirements are accurate, permanent and complete before the development starts, and software development often requires flexibility; however, Waterfall's approach is ***suitable for our console device***.

2.1.2. ITERATIVE METHODOLOGIES

2.1.2.1. AGILE

Agile is not a methodology, as it does not stipulate the exact steps of the development phases. It, instead, can be considered a ***development philosophy***. The agile manifesto is mere 68 words, yet its principles completely revolutionised traditional software development. For example, one principle states that "*our highest priority is to satisfy the customer through early and continuous delivery of valuable software*" (Fowler and Highsmith, 2001). Continuous delivery cannot be achieved through sequential approaches. Moreover, Agile lacks linear development's extensive documentation style.

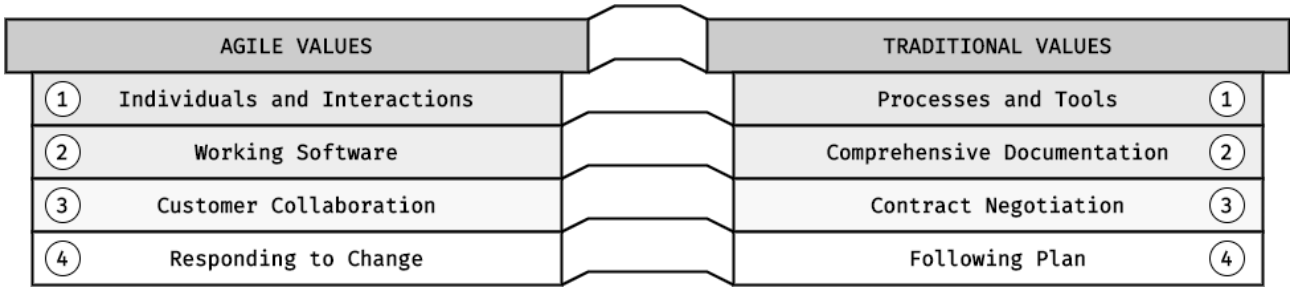


Figure-13 Agile-Manifesto *(Appendix/Agile.drawio)*

Agile solidified in the developer community's collective awareness; methodologies soon were based on its principles, such as Scrum, Extreme Programming, Lean, Feature-Driven Development, or DevOps. Some of these methods may be tailored or trivialised to suit a one-person project, but they essentially revolve around managing processes for ***teams and departments***. For instance, Scrum requires Scrum Master, Product Owner and Developers, or XP increases quality by pair programming. Therefore, we must discover methodologies more aligned with a ***single-developer project scenario***.

2.1.2.2. TEST-DRIVEN DEVELOPMENT

Waterfall methodology tests code and business functionalities retroactively, which carries the potential risk of failing to mitigate problems early. TDD, an agile practice, "*requires writing automated tests prior to developing functional code in small, rapid iterations*" (Janzen and Saiedian, 2005). The functions must be granulated into individual testable units to achieve such automation.

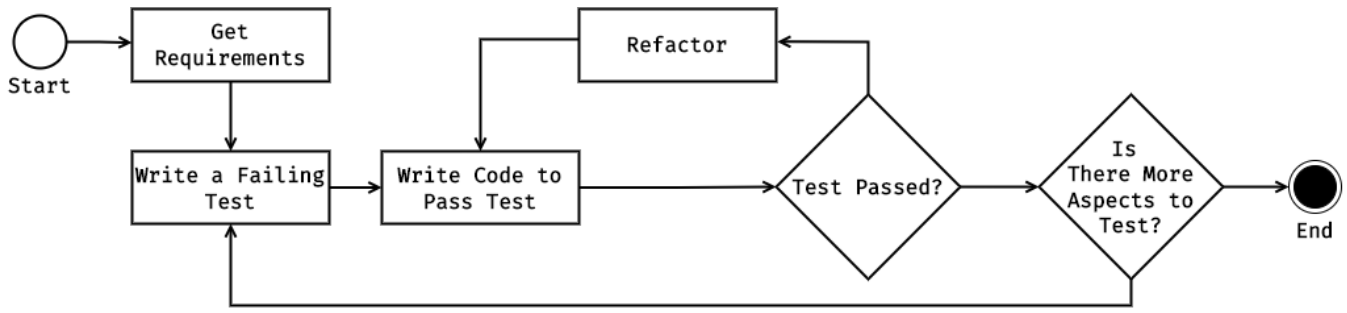


Figure-14 TDD Algorithm *(Appendix/TDD.drawio)*

As we can see, TDD is a test-first approach. Units and functions are tested deterministically, avoiding any side effects.

2.1.2.3. KANBAN

The Japanese company Toyota introduced the Kanban system in the 1950s to visualise development work. "*Kanban means card, literally, a visible record used as a means of communication, of conveying ideas and information*" (Esparrago, 1988). Software development soon adopted this methodology for improved efficiency to divide its workflow into requested, in-progress and done categories in a board system. One fundamental characteristic of Kanban is the limit of in-progress cards to ensure manageability and reduce multitasking.

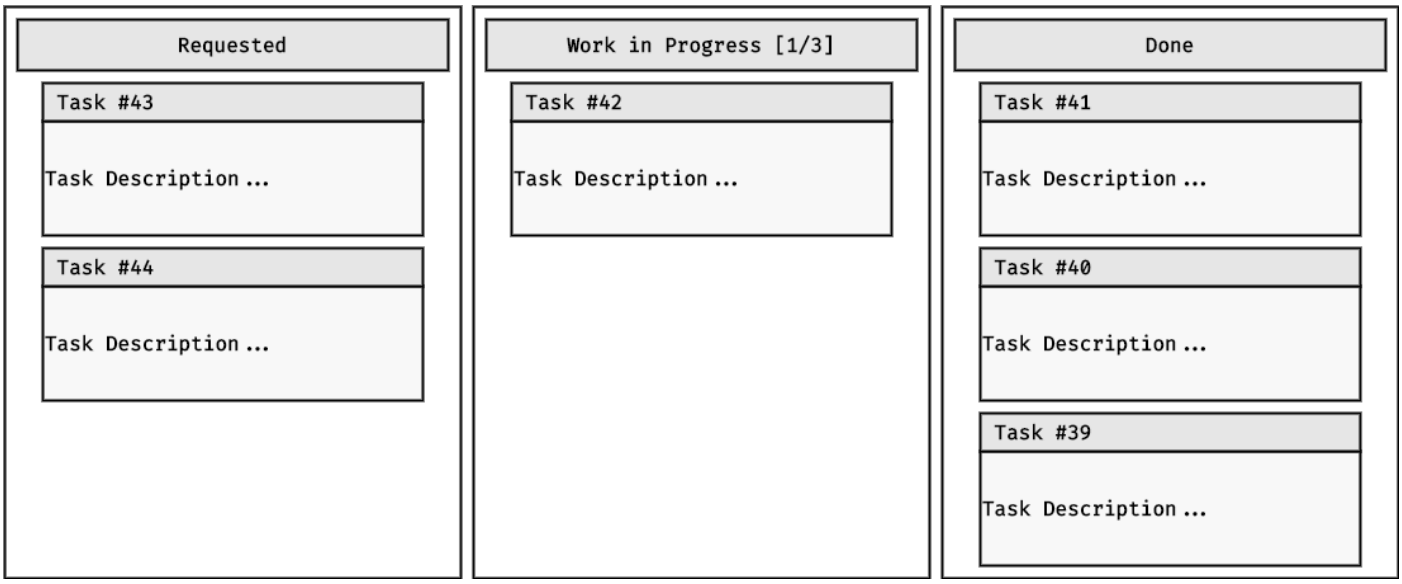


Figure-15 Kanban *(Appendix/Kanban.drawio)*

2.1.3. APPLIED METHODOLOGIES

"*When an organisation states that it uses a particular methodology, it often applies a combination of smaller, finer-grained methodologies on a project scale instead*" (Janzen and Saiedian, 2005). Our project design and development will be split into hardware and software and handled separately. The hardware development will follow a linear progression, like ***Waterfall***. Because of our fixed requirements, no changes are expected in manufacturing.

Software development will start when the fully-functional console is tested. The software engineering will follow ***TDD*** principles to ensure quality, and tests will be automated with ***Jest***. The software features will be divided and developed iteratively. Additionally, we will apply ***Kanban*** because its simple and practical approach is well-suited for a single-developer project supporting the level of organisation required in small-scale environments. The in-progress cards will be limited to two to mitigate the accumulation of unfinished tasks.

2.2. WORK BREAKDOWN STRUCTURE

The Work Breakdown Structure is split into hardware and application development phases. The console prototyping follows a linear, step-by-step development style where each task is started when the last one finishes emphasising early finalisation of hardware development because failing to engineer a working prototype would jeopardise the entire project.

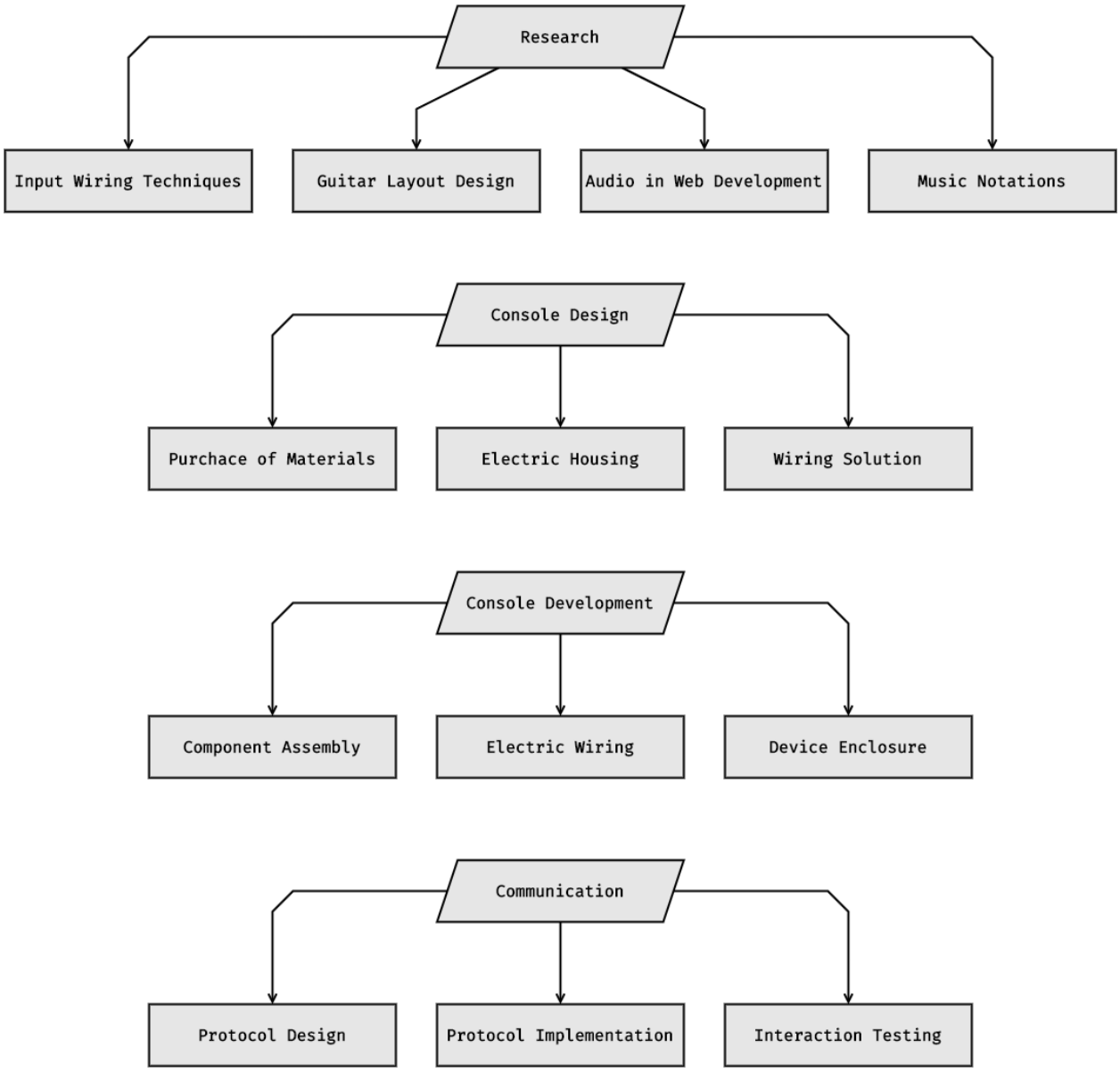


Figure-16 WBS-Console

The application can be built concurrently, allowing ***independent development of many features***. The play and practice options or chord tutorial features may be started after creating our landing page. The order of the tasks, on the other hand not arbitrary. Functionalities like tablature translation or audio may require extensive development time with reusable code. These significant components precede miscellaneous, less vital functionalities.

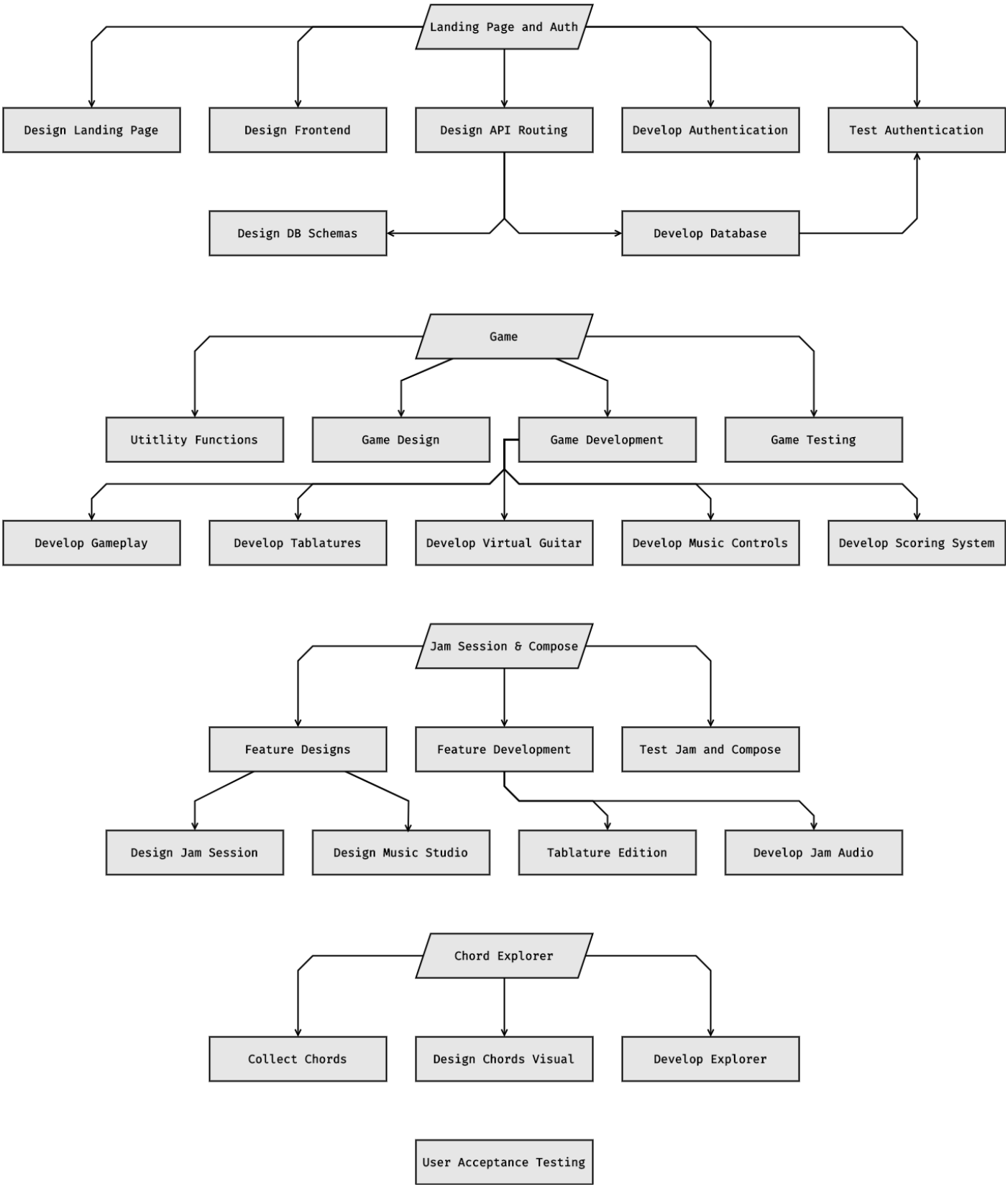


Figure-17 WBS- Application

2.3. SCHEDULING

The project console development phase follows a sequential order, and the software development is iterative with subtask dependency. Our schedule is based on a pessimistic scenario, allocating ample time to develop features, and the final user acceptance tests are scheduled for early April.

	Name	Duration	Start	Finish	Predeces
1	**Research**	16 days?	**03/10/22 08:00**	**24/10/22 17:00**	
2	Input Wiring Technics	10 days	03/10/22 08:00	14/10/22 17:00	
3	Guitar Layout Design	4 days?	14/10/22 08:00	19/10/22 17:00	
4	Audio in Web Development	2 days?	19/10/22 17:00	21/10/22 17:00	
5	Music Notations	2 days?	21/10/22 08:00	24/10/22 17:00	
6	**Design Console**	9 days?	**25/10/22 08:00**	**04/11/22 17:00**	**1**
7	Wiring	5 days?	25/10/22 08:00	31/10/22 17:00	
8	Housing	3 days?	01/11/22 08:00	03/11/22 17:00	7
9	Purchase of Materials	1 day?	04/11/22 08:00	04/11/22 17:00	8
10	**Develop Console**	18 days?	**07/11/22 08:00**	**30/11/22 17:00**	**6**
11	Unhoused Wiring	5 days?	07/11/22 08:00	11/11/22 17:00	
12	Mock Housing	3 days?	14/11/22 08:00	16/11/22 17:00	11
13	Final Housing	10 days?	17/11/22 08:00	30/11/22 17:00	12
14	**Test Console**	5 days?	**01/12/22 08:00**	**07/12/22 17:00**	**10**
15	Design Protocol	1 day?	01/12/22 08:00	01/12/22 17:00	
16	Build Protocol Software	3 days?	02/12/22 08:00	06/12/22 17:00	15
17	Test Console Interactions	1 day?	07/12/22 08:00	07/12/22 17:00	16
18	**Landing Page and Authentication**	21 days?	**08/12/22 08:00**	**05/01/23 17:00**	**14**
19	Design Landing Page and Authenticatio	2 days?	08/12/22 08:00	09/12/22 17:00	
20	Design Frontend	3 days?	10/12/22 08:00	14/12/22 17:00	
21	Design DB Model	2 days?	14/12/22 08:00	15/12/22 17:00	
22	Design API Calls	2 days?	15/12/22 08:00	16/12/22 17:00	
23	Develop Frontend Authetication	3 days?	19/12/22 08:00	21/12/22 17:00	22
24	Develop Backend and DB	10 days?	22/12/22 08:00	04/01/23 17:00	23
25	Feature Test Authentication	1 day?	05/01/23 08:00	05/01/23 17:00	24
26	**Play Option**	31 days?	**06/01/23 08:00**	**17/02/23 17:00**	**18**
27	Design Play Option	1 day?	06/01/23 08:00	06/01/23 17:00	
28	**Develop Play Option**	29 days?	**09/01/23 08:00**	**16/02/23 17:00**	**27**
29	Develop Guitar Console Simulation	6 days?	09/01/23 08:00	16/01/23 17:00	27
30	Develop Tabulature Visualisation	8 days?	17/01/23 08:00	26/01/23 17:00	29
31	Develop Music Control	4 days?	27/01/23 08:00	01/02/23 17:00	30
32	Develop Gameplay	7 days?	02/02/23 08:00	10/02/23 17:00	31
33	Develop Scoring System	4 days?	13/02/23 08:00	16/02/23 17:00	32
34	Feature Test Play Option	1 day?	17/02/23 08:00	17/02/23 17:00	33
35	**Practice, Jam and Compose**	18 days?	**20/02/23 08:00**	**15/03/23 17:00**	**26**
36	**Design Practice, Jam and Compos**	17 days?	**20/02/23 08:00**	**14/03/23 17:00**	**26**
37	Develop Manual Tab Editing	7 days?	20/02/23 08:00	28/02/23 17:00	26
38	Develop Practice, Jam and Compose	10 days?	01/03/23 08:00	14/03/23 17:00	37
39	Feature Test Practice, Jam and Compos	1 day?	15/03/23 08:00	15/03/23 17:00	38
40	**Chord Explorer**	9 days?	**16/03/23 08:00**	**28/03/23 17:00**	**35**
41	Collect Chords JSON	1 day?	16/03/23 08:00	16/03/23 17:00	
42	Design and Develop Chord Translation	5 days?	17/03/23 08:00	23/03/23 17:00	41
43	Develop Chord Visualisation	3 days?	24/03/23 08:00	28/03/23 17:00	42
44	User Acceptance Testing	5 days?	24/03/23 08:00	30/03/23 17:00	42

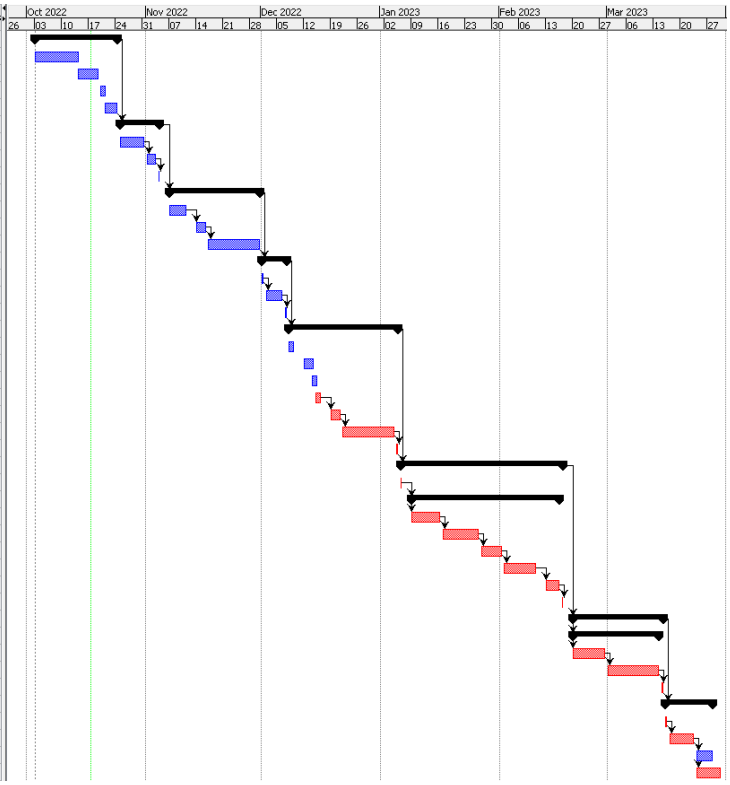


Figure-18 Gantt Chart *(Appendix/RiffMaster.pod)*

3. LITERATURE REVIEW

3.1. TERMINOLOGY

Guitar Parts: We will refer to the digital guitar components, such as frets, bridge, and neck, as they were parts of real instruments *(Figure-19)*.

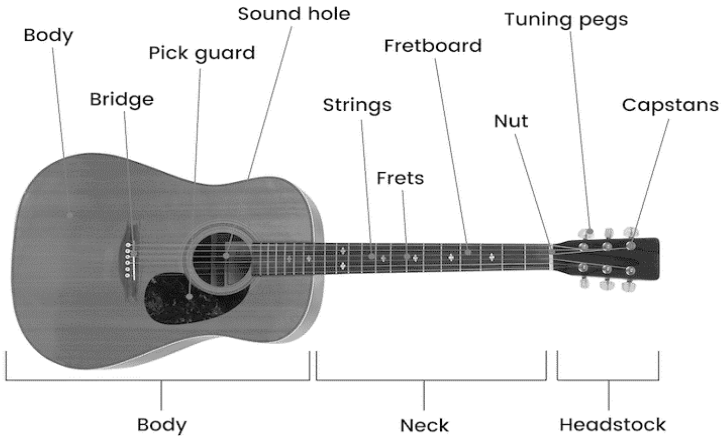


Figure-19 Guitar Parts (HelloMusic, 2022)

Sound: is mechanical wave energy (vibration) that propagates through a medium, causing variation in pressure.

Audio: refers to the digital (electronic) representation of sound vibrations. The difference between sound and audio is their energy form.

Chord: is "*three or more notes sounding simultaneously. It can be played on one instrument, like a guitar, or by many instruments at once, like a brass quartet or a choir*" (Eriksson, 2016)

Riff: The riff's etymology is unclear; however, it was first used in the 1930-s to shorten refrains. The term stayed in pop music; now, it refers to short, repeated melody patterns.

Monophony and Polyphony: We address monophony because some keyboards are wired so that only one keypress can be recognised at a given time. This limitation would result in a monophonic instrument. As guitars are polyphonic instruments, strings can be strummed simultaneously. ***Our goal is to make the input device polyphonic.***

Hammer-On Pull-Off (HOPO): This guitar technic allows the player to produce distinct sounds without strumming by hammering/pulling off fingers from a given string, a desired feature of guitar games.

3.2. NOTATIONS

Guitar literature uses standard music and tablature notation for different circumstances. ***Standard notation*** is primarily used in classical music, while tablatures are more common for novice players. Although the standard notation is music's lingua-franca, it has no boundaries for the instrument it interprets. Its difficulty might be a hurdle for gameplayers because of its steep learning curve.

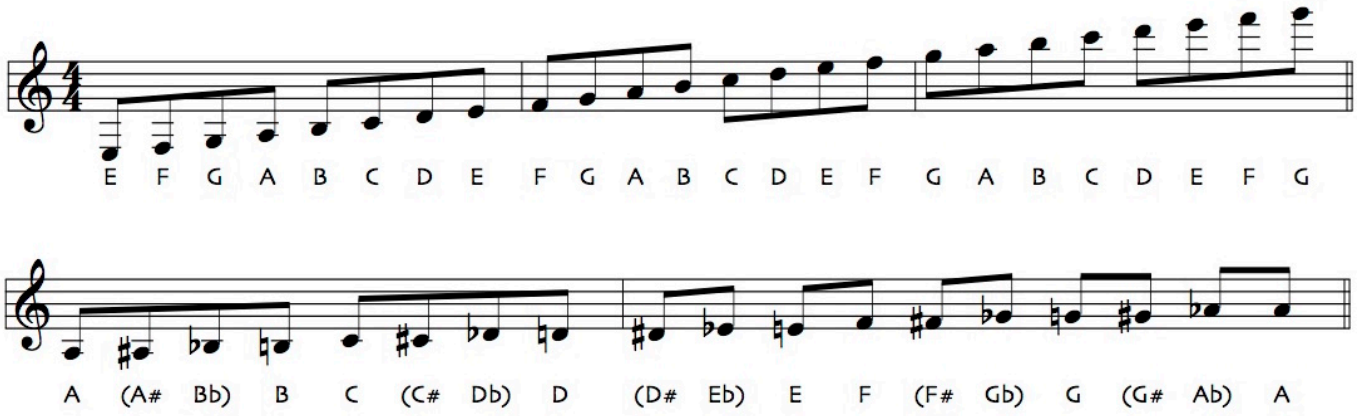


Figure-20 Standard Music Notation (PegHead, 2022)

"*Because it is meant to be used by any instrument, common notation is not written to tell how to play notes on a specific instrument. Instead, common notation simply tells you what each note should sound like … **Tablature** focuses on telling how to play the notes on the instrument. A note is presented in terms of which string to play, and where to hold the string down on the fretboard*" (Schmidt-Jones, 2016).

The application will use tabular notation in the gameplay for simplicity. The only drawback is that tablatures lack rhythmic syntax that must be appended artificially. We restrict our tablatures to 4/4 common times, standard tune settings, and well-known rhythm signs as per convention.

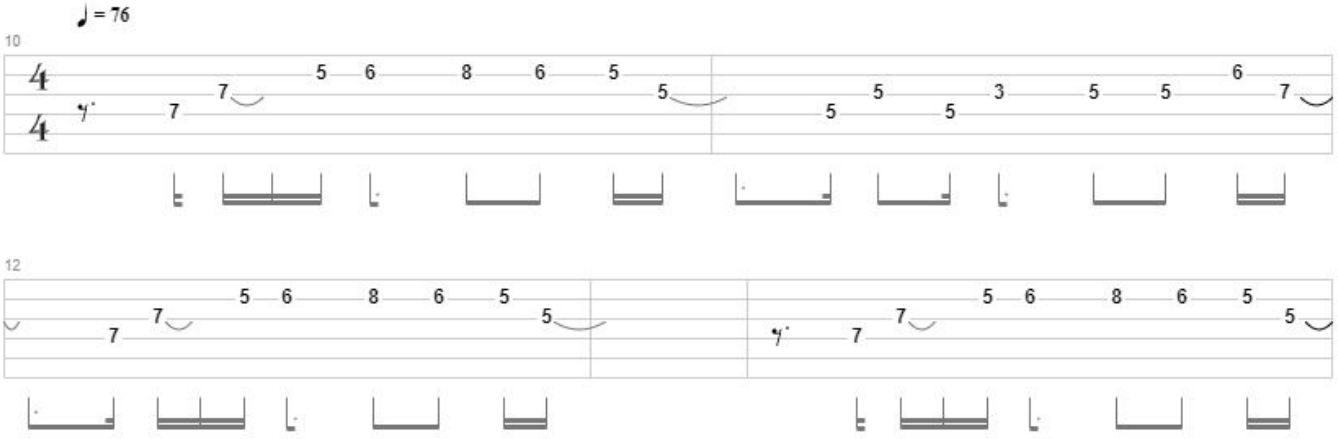


Figure-21 Rhythmic Tab (UltimateGuitar, 2022)

3.3. TOPIC RESEARCH

To create a unique and not yet invented product, we must set apart from the crowd and clarify the aim and objectives of the project in reflection of the existing market. Hence, before proceeding to any project design, we must explore the relevant gaming and musical entertainment devices. This precautionary research will prevent us from reinventing the wheel and will refine the project's outline more accurately.

3.3.1. EXISTING TECHNOLOGIES

Guitar Hero

This pop culture phenomenon was the initial inspiration behind the project's idea. Harmonix Music System (former owner of Guitar Hero) defined it in its patent as "*a simulated musical instrument that may be used to alter the audio of a video game*" (Chrzanowski, 2015). It was first released in 2005 and has seen several iterations since then. The device features five fret buttons, a strum bar, a whammy bar, and control buttons for the relevant console interface. The main limitations of this console are the restricted number of fret buttons and the single strum bar, which prevents it from being used as an authentic polyphonic instrument.

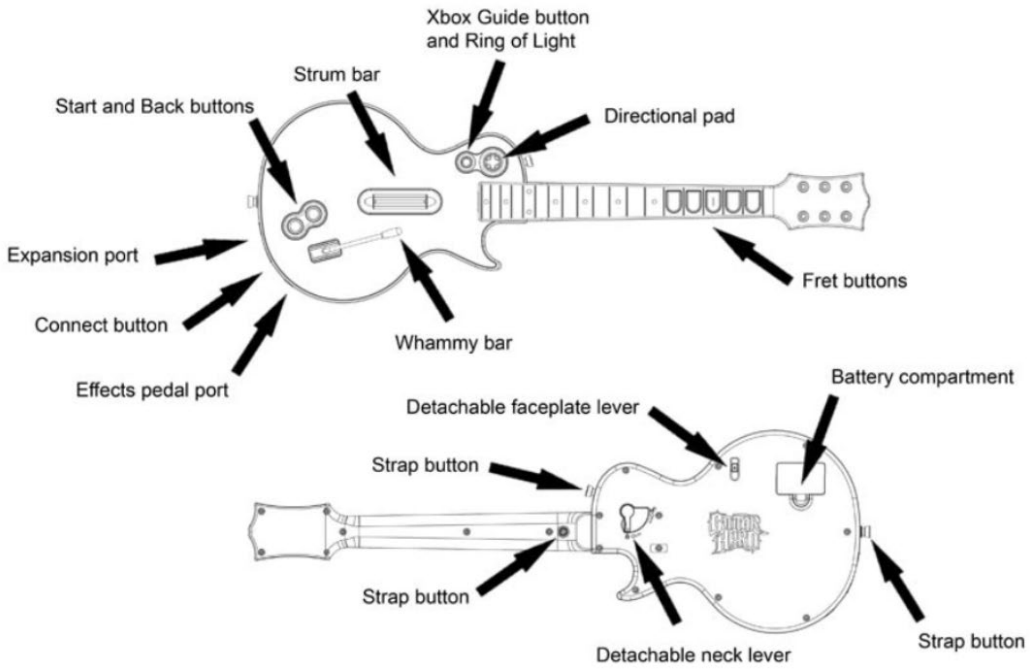


Figure-22 Guitar Hero Layout (Fccid, 2022)

MI Digital Guitar

One of the most promising technologically-enhanced instruments is the MI Digital Guitars from Magic Instruments, currently in the prototyping phase. MI will have a smooth, modern design and be a stand-alone instrument rather than an entertainment system. Among the digital devices on the market, the MI's layout resembles the most to an actual guitar because of its digital strings. Unfortunately, the device is only meant to teach fundamental chord progressions, lacks finger-style features, and is expected to be a premium range device.

Figure-23 MI Digital Guitars (DigitalMusicNews, 2022)

RockSmith

This Ubisoft-developed video game brought music education to the next level by teaching acoustic, electric or bass guitar with adjusting difficulty levels and real-time play feedback. It is one of the leading software technologies focusing on musical autodidactic training. RockSmith is exclusively a software product, and players must own a guitar that can be connected to the game. Consequently, RockSmith can be considered a specialised training software focusing on guitar skills.

3.3.2. TECHNOLOGY GAP

While these examples represent a fraction of the available market, they cater to diverse needs for specific audiences. Based on our topic research, we may sum up our findings:

	Release	Vendor / Owner	Type	System / Platform
Guitar Hero	2005	Harmonix / Activision	Game / App	PC, PS, XBox, Nintendo
MI Digital Guitar	TBA	Magic Instruments	Instrument / App	iOS / Android
RockSmith	2011	Ubisoft	Application	PS / XBox / Win / Mac
RiffMaster	2023 June	Personal Project	Instr. / App / Game	PC

	Instrument Layout	Weight	Dimension	Cost (App + Device)
Guitar Hero	5 Fret / 1 Strum	1.01 - 1.3kg	0.43 x 0.34 x 0.08m	£20 - £300+
MI Digital Guitar	14 Fret / 6 Strings	~2.26kg	TBA	~£299
RockSmith	Actual Guitar / Varies	N/A	Varies	£12.99 / month
RiffMaster	18 Fret / 6 Strum	TBA	TBA	TBA

Figure-24 Comparison (Appendix/Comparision.drawio)

Our summary discovered a range of functionalities that novice guitarists would benefit from. But unfortunately, they are sparsely isolated throughout several products with distinct merits and limitations. Ideally, these features should be ***consolidated into one comprehensive product***. While Guitar Hero has outstanding gameplay, its unrealistic controller lacks authenticity. On the other hand, MI's Digital Guitar has an excellent artistic design with a natural layout but inaccurate fret distances. Unfortunately, its synthetic music system cannot play notes, just chords. RockSmith offers an exceptionally realistic application that teaches fundamental music skills, but users must buy a decent-quality instrument to play.

Finally, all these systems are vendor-specific, proprietary, and licenced. As a software developer, I desire to create applications around a digital guitar instrument ***free from licences or the concern of litigations***. Additionally, our device should have specifications, protocols and documentation available for software developers in the community to encourage competition.

3.4. DESIGN RESEARCH

3.4.1. FRET DISTANCES

Current devices like the MI Digital Guitar also ignore the distances between frets, damaging their authenticity. Conventionally, we calibrate the guitar neck to natural proportions where the 12th fret is halfway to the scale length between the nut and the saddle. The fret positions may be calculated by dividing the remainder of a fret-saddle distance by 17.817 recursively.

Fret 1	0.406m	Fret 2	0.383m	Fret 3	0.362m	Fret 4	0.341m
Fret 5	0.322m	Fret 6	0.304m	Fret 7	0.287m	Fret 8	0.271m
Fret 9	0.256m	Fret 10	0.241m	Fret 11	0.228m	Fret 12	0.215m
Fret 13	0.203m	Fret 14	0.192m	Fret 15	0.181m	Fret 16	0.171m
Fret 17	0.161m	Fret 18	0.152m	Fret 19	0.143m	Fret 20	0.135m

Figure-25 Fret Distances *(Appendix/Fret-Distance-Table.drawio)*

These distance values can be used to create our trial template for the physical layout. However, this layout doesn't yet consider wiggle rooms between the buttons.

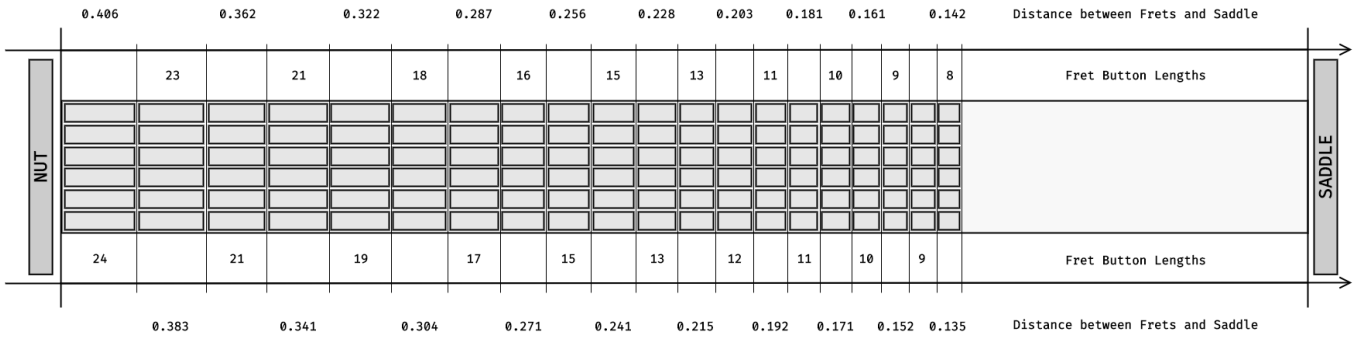


Figure-26 Guitar Neck Distances *(Appendix/Fret-Distance-Neck-Diagram.drawio)*

3.4.2. KEY-SWITCH INTERFACES

To build a successful guitar system based on time precision, we need to detect inputs from the user interface accurately and efficiently. From an electrical engineer's perspective, our console can be abstracted as an intricate keyboard input consisting of an array of key switches, similar to a conventional computer keyboard. "*Depending on how individual switches are connected, mechanical keypads are commonly available in two forms – matrix and common bus*" (Dave and Rajiv, n.d.).

3.4.2.1. BUSES

In our scenario, we need 120 fret buttons and six strum switches. Unfortunately, connecting switches directly to pins would be inelegant even if we could afford to use a microcontroller with 126 digital pins. The fundamental sequential wiring design is to use a common bus.

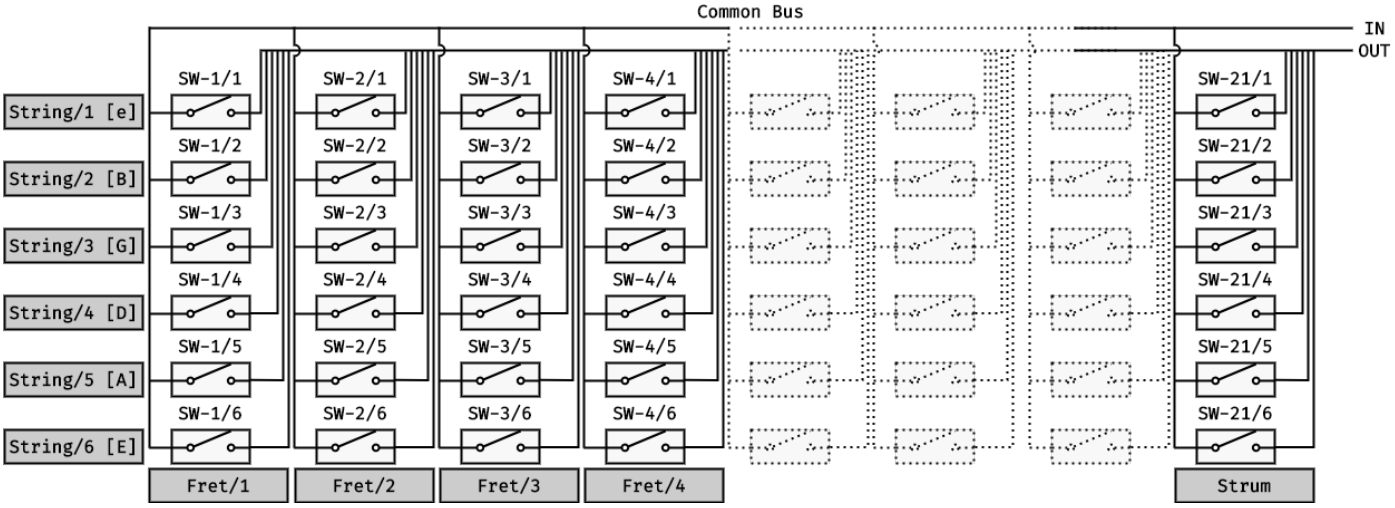


Figure-27 Common Bus Wiring Schema (Appendix/CommonBus.draw.io)

This solution in the above form is not ideal for RiffMaster, and other linear solutions will be discussed in the following chapters.

3.4.2.2. MATRICES

Keyboards and keypads often use matrices to consolidate greater numbers of switches to microcontroller pins. "*When a key is pressed, a column wire makes contact with a row wire and completes a circuit. The keyboard controller detects this closed circuit and registers it as a key press*" (Dribin, 2000). For example, PC keyboards usually range from 63-105 keys arranged in a matrix. Meshing the switch wires would result in a drastically reduced digital pin requirement:

$$bus = rows \times columns = 6 \times 20\ fret + 1\ strum = \mathbf{126}\ pins$$

$$matrix = rows + colums = \mathbf{26}\ pins$$

Matrix keyboards use scanning algorithms to detect button presses, where rows and columns are individually read. After reengineering different matrix examples, we boiled down the algorithms to this universal diagram:

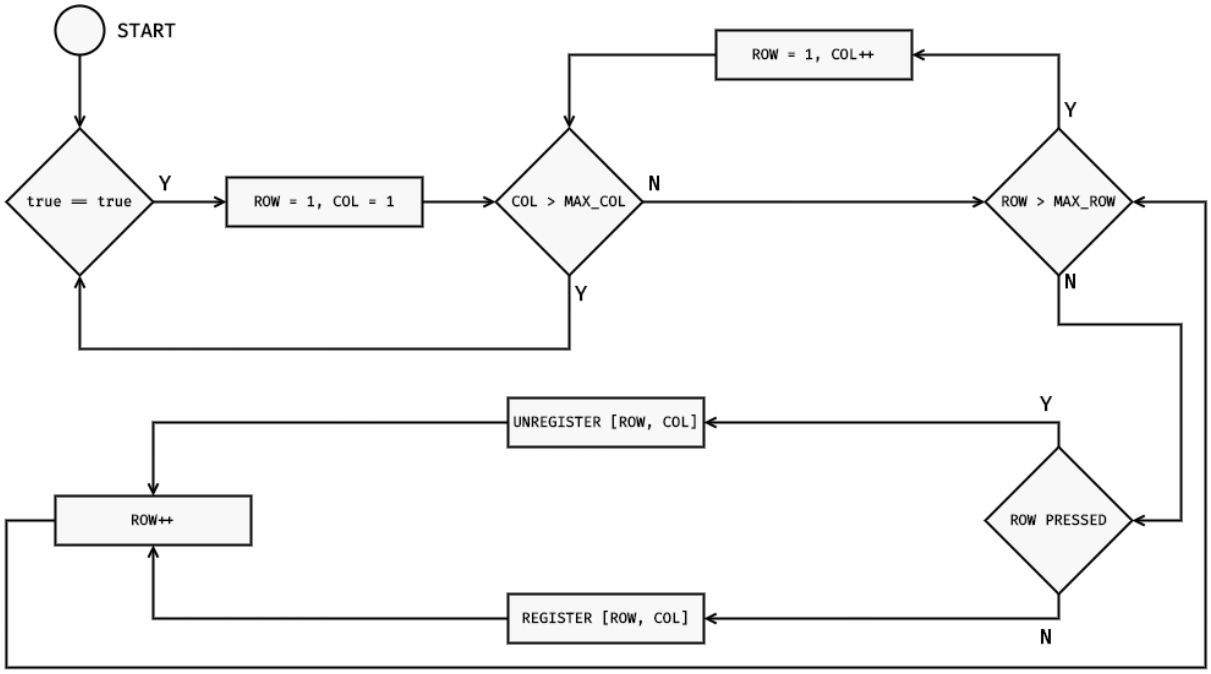


Figure-28 Keyboard Scanning (Appendix/Keyboard-Scanning.drawio)

Unfortunately, keyboard matrices introduce problems with simultaneous key presses, called keyboard ***ghosting*** (unrecognised strokes) and ***masking*** (unpressed strokes mistakenly registered). These problems are well-known in the gaming community, and anti-ghosting keyboards are sold for professional gamers.

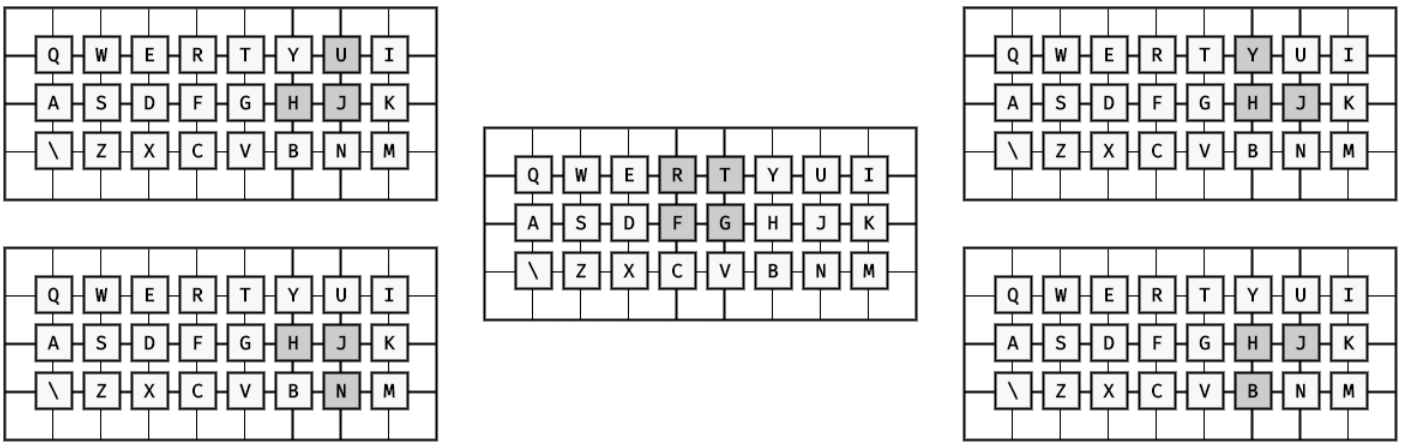


Figure-29 Masking (https://www.microsoft.com/applied-sciences/projects/anti-ghosting, 2022)

Ghosting and masking may cause problems when the player simultaneously interacts with the controller's fret buttons, playing chords. ***These glitches may be corrected by using diodes.***

3.4.3. LADDERS

Traditionally buttons are registered through digital pins because of the inputs' binary nature. However, with an ingenious trick, we may capture keypresses on analog pins, similar to how potentiometers work or how sensor values are read. "*To get the value from the sensor, call analogRead() that takes one argument: what pin it should take a voltage reading on. The value, which is between 0 and 1023, is the representation of the voltage on the pin*" (Fitzgerald and Shiloh, 2012).

Analog ladders could also eliminate the ghosting and masking problems because only the topmost button press on each row is registered, similar to the real-world instrument. Ladders could reduce the pin requirements to six (strings). One way of creating analog voltage dividers is to use sequential switch wiring and identical resistors, often called ***logarithmic resistor ladders***.

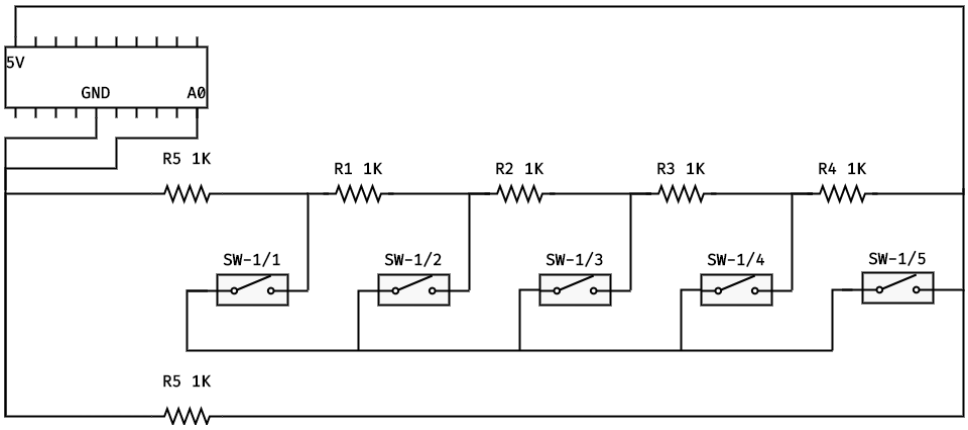


Figure-30 Resistor Ladder *(Appendix/Resistance-Ladder.drawio)*

Although this solution is straightforward, it works for projects with fewer switches. Even if we assume perfect resistors, we must interpret our voltage divider values by hard-coding a logarithmic range. Our readings would be inaccurate since each string has twenty values on a logarithmic curve. We can equalise the distances with ***linear resistor ladders*** by calculating the resistance for each button switch.

$$R_{button} = \frac{R_1}{1 - B_{index}/B_{total}} - R_{total}$$

Finding twenty resistor values require more than a handful of calculations, but there are algorithms to help us. The real drawback of linear ladders is that we must have a supply of various precise resistors in large numbers.

3.4.4. DEBOUNCE MECHANISM

"*The physical action of pushing a button might require a half-second or so, so we tend to think in those terms. On the other hand, a digital circuit can react to a million of events in the same time frame*" (Warren, 2015). This mechanism may cause multiple input activations in relatively short intervals. For game consoles, it would result in disastrous UXs. Electrical solutions, such as flip-flops and Schmitt-triggers, have been used to solve this problem. A "S*chmitt trigger circuit relies on changing the voltage or current threshold levels by means of positive feedback in the analogue loop*" (Kader and Rashid, 2012), improving the immunity to analogue disturbances.

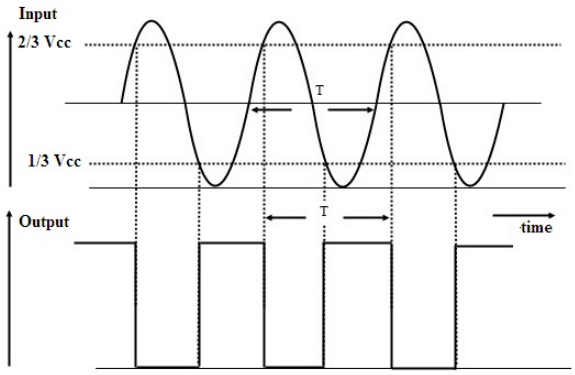


Figure-31 Schmitt-Trigger (WatElectronics, 2022)

Alternatively, ***debouncing can be addressed through software engineering techniques***. For example, Arduino microcontrollers have a millis function that "*returns the number of milliseconds passed since the Arduino board began running the current program. This number will overflow (go back to zero) after approximately 50 days*" (Arduino, 2022). So we can mitigate debouncing by measuring switch state change intervals. The time of changes should be recorded, and a debounce delay may be applied to compensate for noise. Input actions should be ignored on state changes with unreasonably short intervals.

3.5. HUMAN INTERFACE DEVICE

We need a protocol that works through the USB connection to establish efficient communication between the console and the computer. The Human Interface Device Protocol "(*HID) is designed for common PC interface devices such as keyboard and mouse, but can be adapted for many custom applications*" (Murphy, 2017). Arduino uses HID communication through the Serial class, a built-in library for simple ASCII I/O actions. However, even though a USB cable powers most Arduinos, ***bidirectional USB communication*** is only possible on few Arduino MCUs.

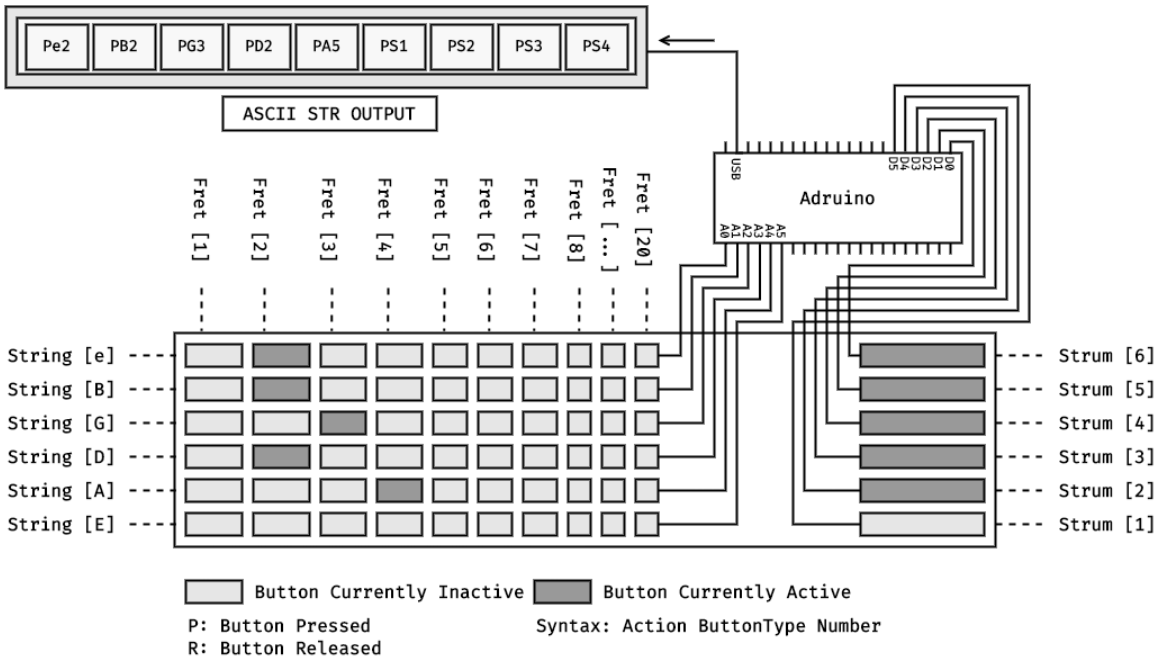


Figure-32 Protocol Example *(Appendix/Serial-Protocol.drawio)*

ASCII keyboard communication is the most straightforward solution because characters can easily be mapped to corresponding user actions. However, a significant drawback is that using our console device as a keyboard might interfere with the user's original keyboard input, confusing our users.

3.5 AUDIO CACHING

The application aims to engage the player through music and seamless game interaction. "*Well-designed games create engagement by promoting a low state, a total absorption that makes the player gratifyingly oblivious to anything else. Good musical experiences also involve low states, and music classes are most effective when they foster flow*". (Csíkszentmihályi, 2009) However, to create the right conditions for flow, the players need audio feedback of several kinds.

Fortunately, web browsers support audio interactions, and multiple audio files can play simultaneously. However, as media files (MP3/MPEG-4) are typically larger than text-based files, the application may need to preload music data to the cache. "*Web caching is the temporary storage of Web objects (such as HTML documents) for later retrieval. There are three significant advantages to Web caching: reduced bandwidth consumption (fewer requests and responses that need to go over the network), reduced server load (fewer requests for a server to handle), and reduced latency*" (Sulaiman and Siti, 2008). Unfortunately, browsers don't handle caching uniformly, which might cause ***vendor compatibility*** problems, and users may also have browser settings that prevent caching.

4. DESIGN

Our entertainment system has multiple well-separable sections, such as the game controller's physical design, communication with the user interface, and application design. While each goes through separate design phases, they must be consolidated into one compatible working interface.

4.1 CONTROLLER DESIGN

Our controller is divided into three fundamental units, the neck, where the user interacts with the fret inputs; the strum, and the body. The main focus of our design is to arrange our elements in a simple, feasible, and usable manner.

4.1.1. PHYSICAL DESIGN

When designing the game controller, we must know what physical elements our user interface communication may involve. For instance, multiple types of inputs: 120 tactile switches, strum switch solutions or ON-OFF toggle switches. Furthermore, the hardware must accommodate a microcontroller, extensive component wiring, outputs, and USB conversion. The comprehensive list of components suggests that using toy-sized measurements, such as the Guitar Hero console, may prove overly restrictive. As the elements must be assembled into a hand-made device, life-sized instrument blueprints would benefit physical real estate constrictions and authenticity. ***Semi-acoustic*** designs are ideal because of their distinctively elegant open-cavity body structure.

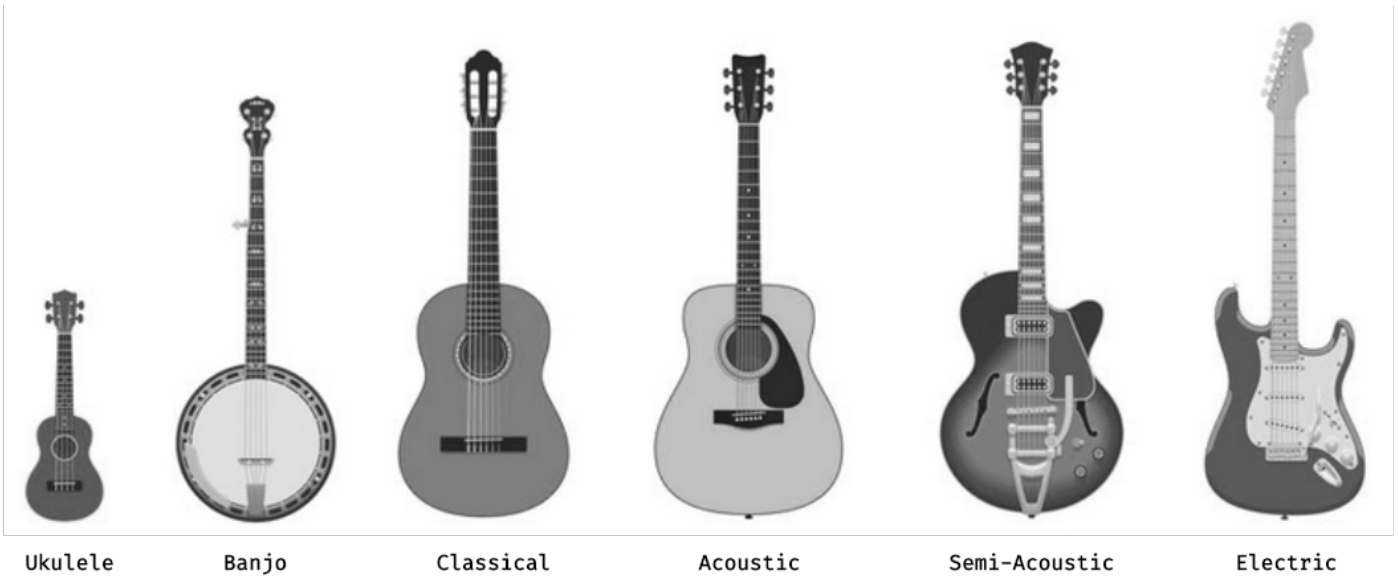


Figure-33 Guitar Types (PianoDreamers, 2023)

One of the most popular and well-recognisable designs is ***Gibson's Les Paul***. We may achieve better accuracy for our overall design by using open-source blueprints for some parameters, such as dimensions and curves.

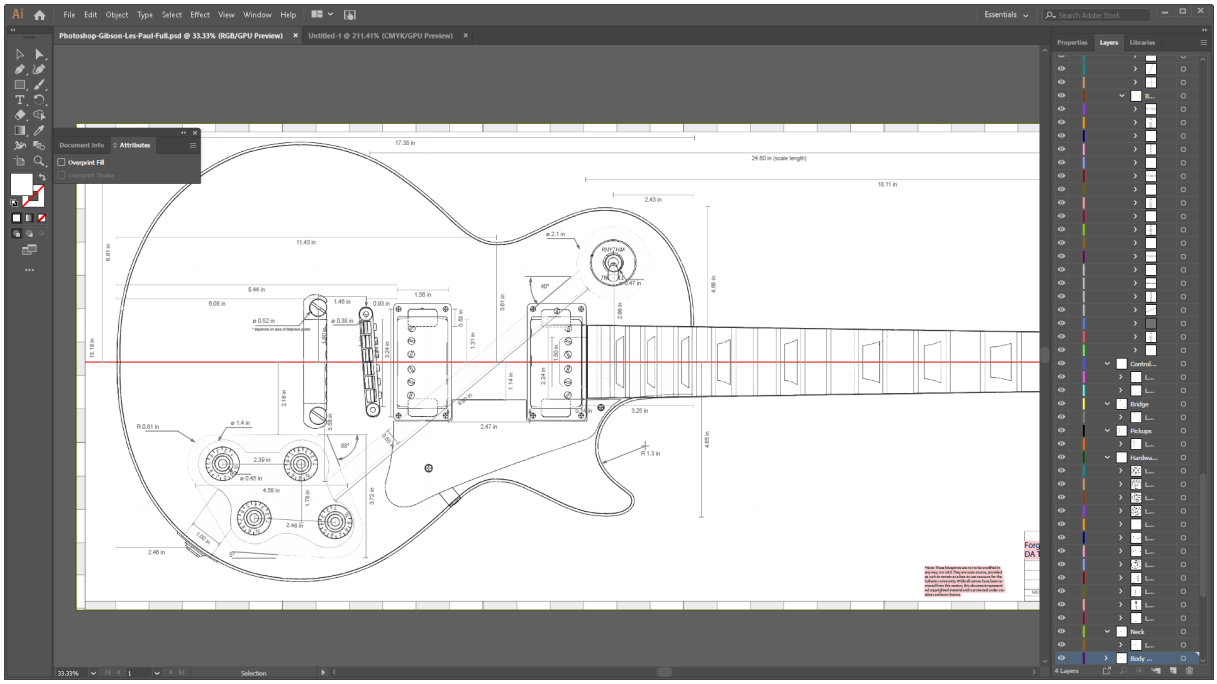


Figure-34 Les Paul Gibson in Adobe Illustrator

4.1.2. LEFT-HAND CONTROL

The user will interact with a range of buttons in a ***mesh arrangement*** *(Section-3.4.2.2.)* to communicate which string has active finger press positions associated. Hence, the neck width and depth must be small enough that all six buttons on a fret may be pressed simultaneously with an index finger for barre chords. However, the neck must be wide enough to leave sufficient space for the electronics (42-57mm for Les Paul). Our console's neck will have a constant width to simplify the hardware development, and 50mm is a compromise that satisfies usability requirements while sufficient for allocating all electrical components. The length of the neck is the standard 460mm, and the fret distances are calculated with the golden ratio discussed in previous chapters *(Section-3.4.1)*.

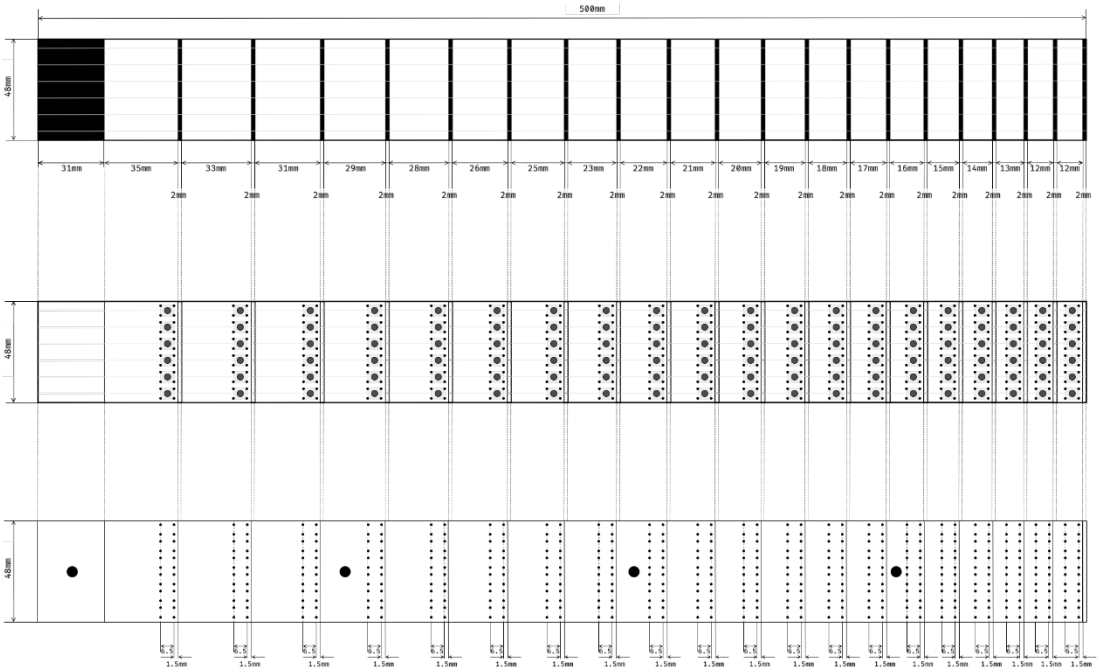


Figure-24 Drilling Template *(Appendix/Drilling_Template.drawio)*

The drill holes will be evenly allocated alongside the fret's axis, distributing the buttons comfortably with 0.7mm wide switch legs in rectangular arrangements.

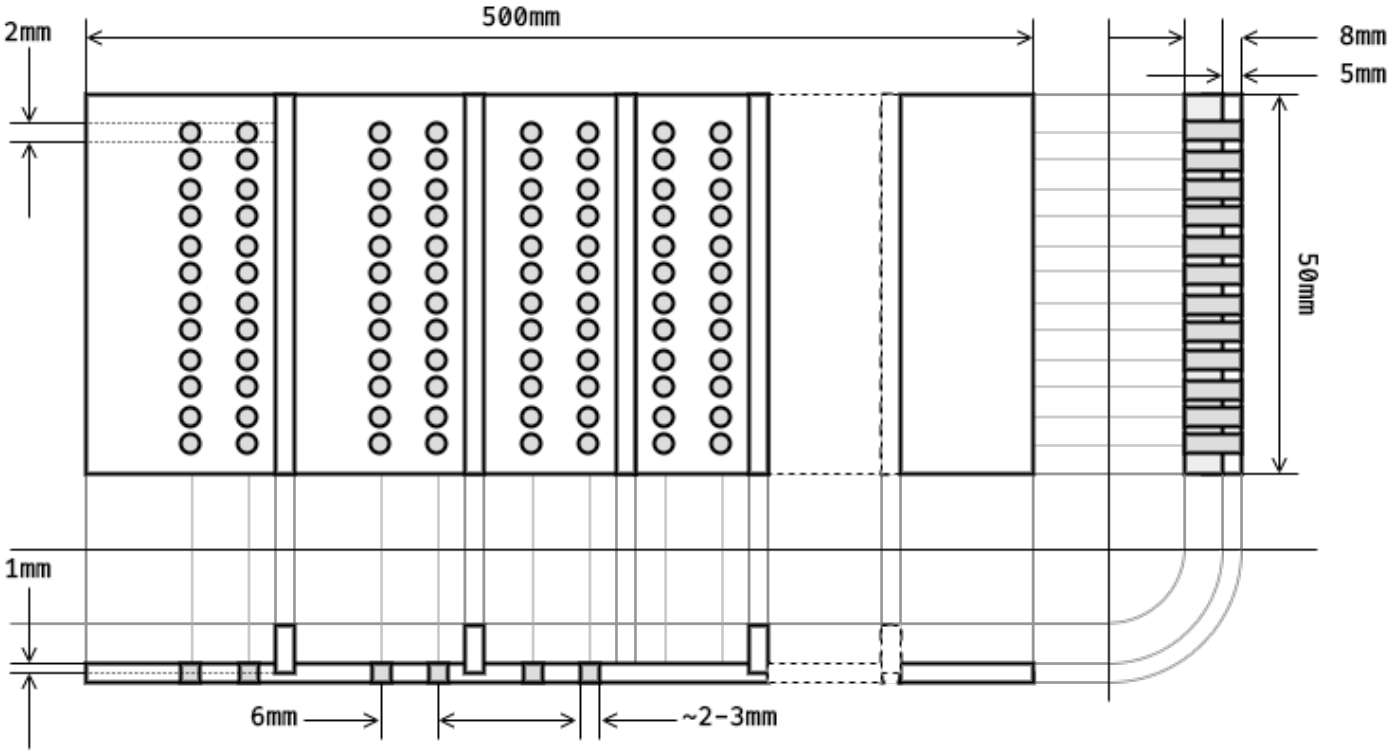


Figure-36 Neck Design (Appendix/Neck_Design.drawio)

Frets receive string supports with grooves, which offer protection for tactile switches, provide guides to the users' hands, and can keep actual strings in place for an optional stringed solution.

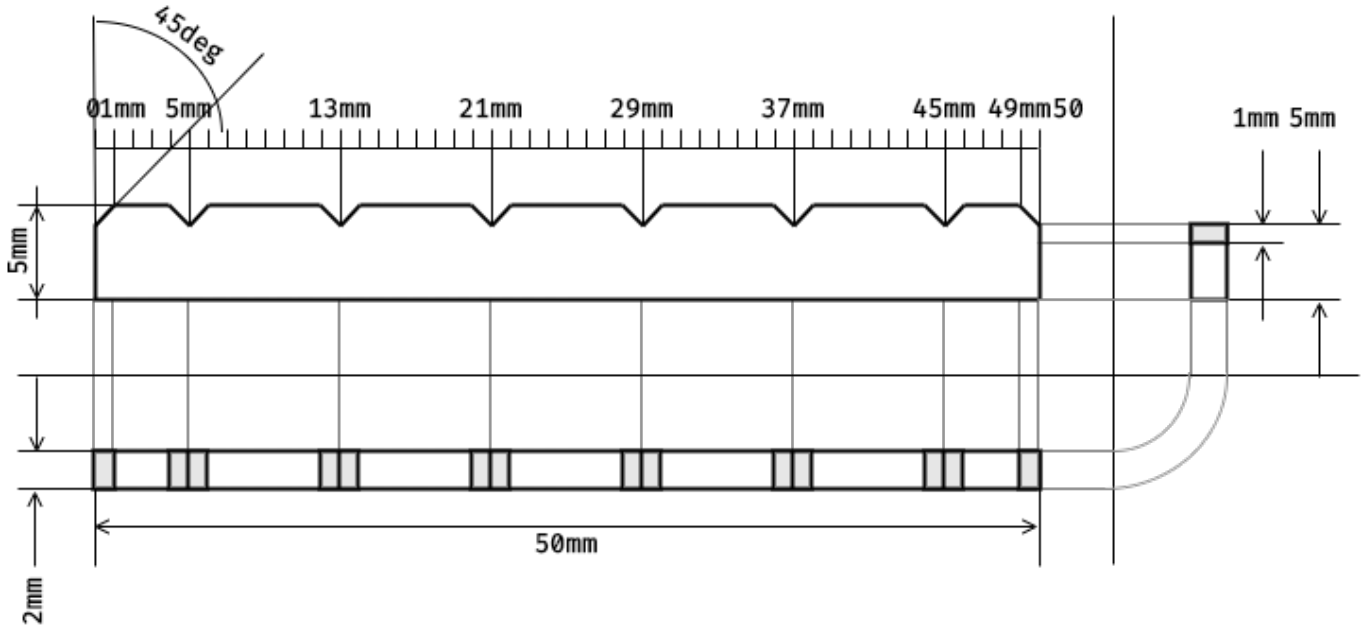


Figure-37 Fret Design (Appendix/Fret_Design.drawio)

The buttons will face towards the user's fingers; therefore, no electronic components may allocate the otherwise compressed interface. The switch legs must lead through the drill holes by wire extensions to reach the backside circuit. The electronics must be protected from physical impacts; therefore, a back cover and side protection are required. Moreover, the neck length must be extended to hold the neck in the instrument body securely.

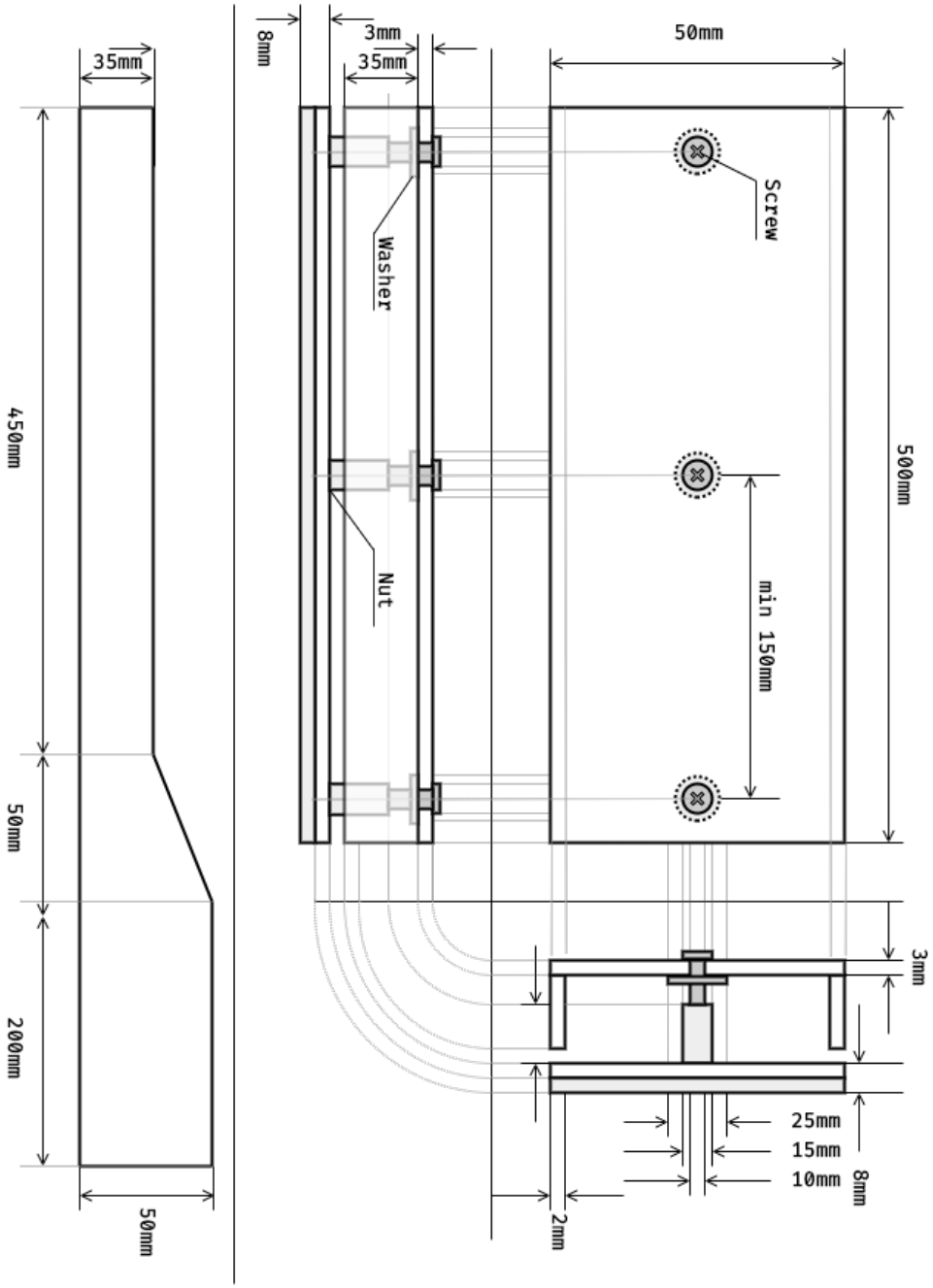


Figure-38 Neck Design (Appendix/Neck_Cover.drawio)

4.1.3. CIRCUITRY

The electrical layout is one of the most challenging parts of the controllers' design. The below diagram includes only the components required for the left-hand side interactions.

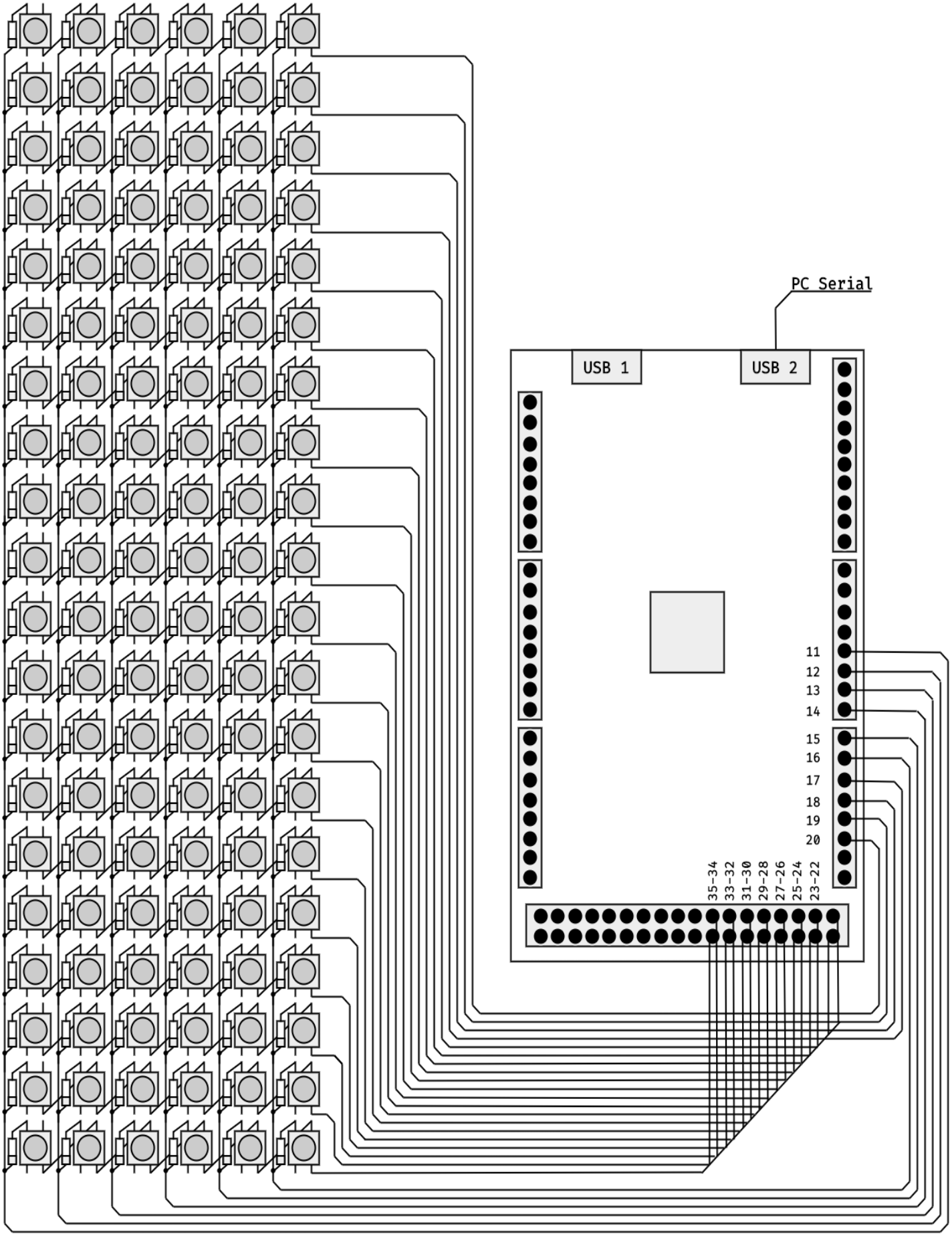


Figure-39 Electric Design (Electric_Design.draw.io)

The interface design research chapter discovered several possible electric solutions, such as analogue voltage dividers like the linear and logarithmic resistor ladders *(Section-3.4.2)*. Nevertheless, the chosen solution fell on the ***matrix design with diodes*** to eliminate ghosting issues. The following reasons directed me towards this design decision:

Linear ladders: building any solutions mentioned so far will take weeks, and linear ladders may not read 120 switch positions accurately, an unnecessary risk that would compromise the entire project.

Logarithmic ladders: would be accurate; however, obtaining the necessary components with exact resistances may take a long time.

Keyboard Matrices: are the most feasible options:

- Requires fewer types of components, switches and diodes; any N1 range diode is suitable for our scenario (N14007 will be used),
- Accurate voltages reading,
- Straightforward, considering that 120 switches are involved,
- Require similar build time than other options,

Disadvantages:

- A 6/20 matrix is a concise layout and may require stripped wires for the row buses (*Figure-40*),
- Components' proximity risk short circuits.

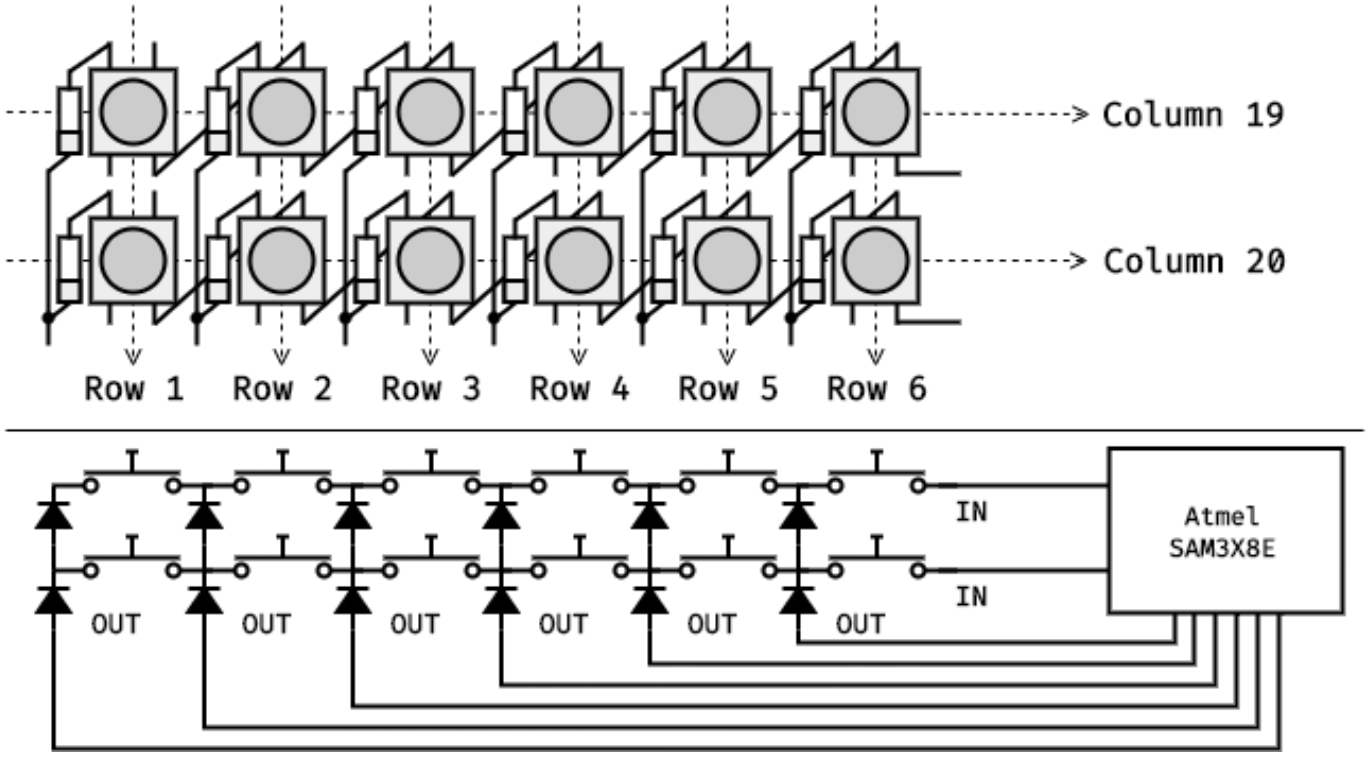


Figure-41 2x6 Matrix *(Appendix/Fret_Electric.drawio)*

4.1.4. RIGHT-HAND CONTROL

The main right-hand component is the strum unit, which triggers events that produce sound effects or gameplay actions. But beyond the strum, the instrument's body must allocate a standard-size on/off toggle, LEDs, USB connection, and support elements to attach the neck, cover, and body. Hand manufacturing constraints the range of executable solutions, like bending perfect curves from plastic or creating parts that fit surgically together, and our design must consider an acceptable tolerance of ±1mm.

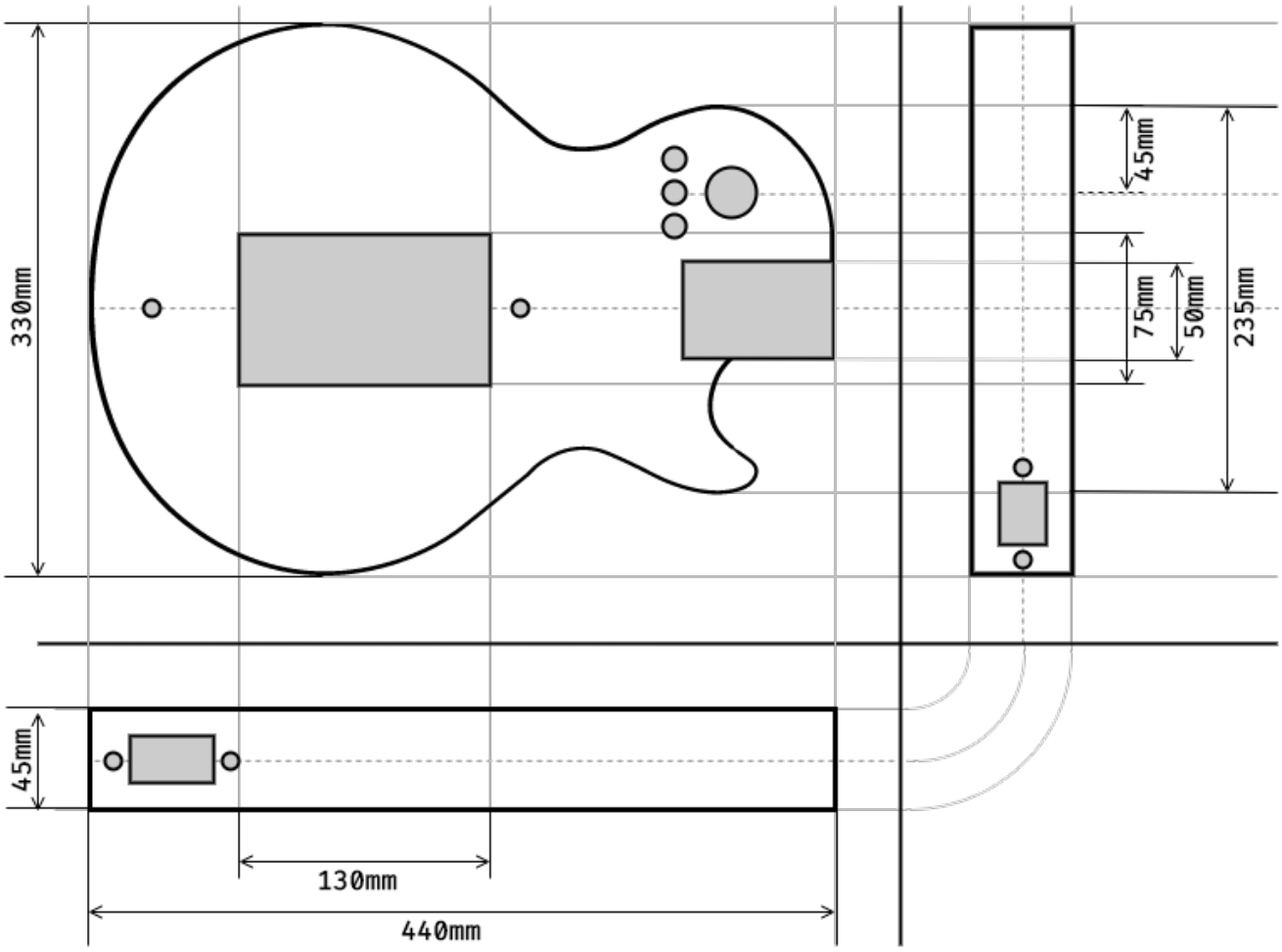


Figure-42 Body Design *(Appendix/Body_Design.drawio)*

4.1.4.1. BODY

The body must be assembled in a way to allow disassembly for servicing. Support units with embedded nuts and screws will hold the body and cover together. These connectors reinforce the device's structural integrity, as the body may be under more stress than other electrical handheld devices. The body must withstand the pressure of the user's right arm's weight without damaging the open-cavity structure. One typical guitar design failure is the weak neck-body connection, resulting in bent or damaged guitar necks. Substantial pulling or pushing forces generate leverage that may easily deform or break the controller apart, and additional components will reinforce the neck to endure reasonable use.

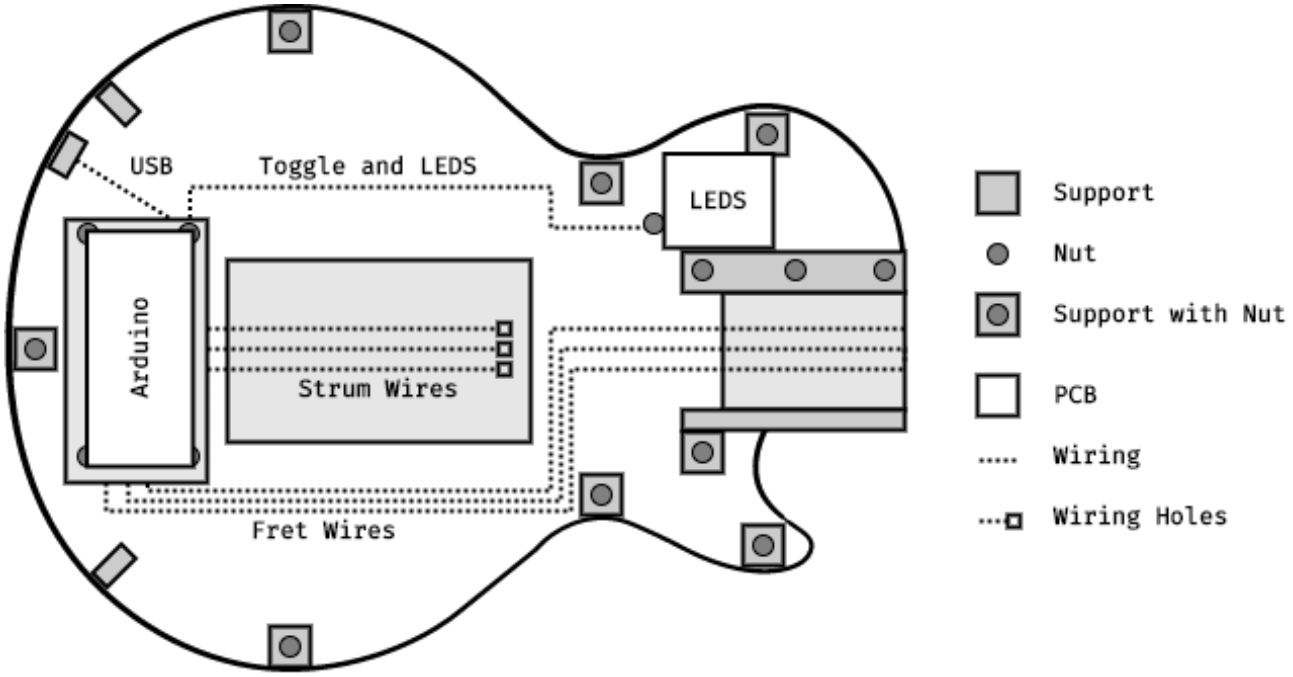


Figure-43 Body (Appendix/Body_Support.drawio)

4.1.4.2. STRUM OPTIONS

Vibration Sensor: can detect strummed strings, capturing additional information, like pressure and note length. Its drawback is that these sensors are sensitive to external factors and need extensive calibration.

Pressure Sensor: can measure the forces applied to strings. However, the cost and space requirement is significant.

Capacitive Sensors: may also read string strums. They are relatively budget-friendly and would fit in a limited space, but their implementation may be complex, and they are sensitive to environmental factors, such as humidity and temperature.

Optical Sensors: may involve a light source above each string. When the user strums the strings, the sensor can read the changes in the light source. This intriguing solution, however, requires a lot of experimentation.

Magnetic Sensors: We may read the magnetic property changes on the strings as the user touches them. Unfortunately, this may result in false readings, as the user may touch the strings without the intention to apply any force on them.

Strum Bar: We can cut corners using six Guitar Hero Strum Bars on the guitar, but these units are difficult to find and require more space than available.

Tactile Switches: The most straightforward solution is to use six buttons to trigger strum events. This method is the least user-friendly, as it does not simulate the guitar strumming experience but is an excellent backup if everything else fails.

4.1.4.3. STRUM SOLUTION

Our chosen method is a ***hybrid*** compromise between switches and pressure sensors. Strings are fixed to momentary switches and act on lateral forces to trigger strum events. The tension may be set with tuning pegs or screws, and springs may be applied to adjust the pressure sensitivity of strums. A medium-difficulty solution that resembles more of the actual instrument mechanism. They fit the available space, can be delivered quickly and has the accuracy to produce an authentic experience.

On the downside, the string switches are not analogue inputs and are less versatile than sensor readings, and extensive fine-tuning is required to get the desired tension on the strings. The ***levers*** that trigger actions *(Figure-44)* are mechanical components ***prone to physical damage***. Users may be required to adjust the pressure periodically, as strings loosen with time, and users may damage the unit while tuning. Lastly, user opinion may be divided on stringed switches; experienced guitarists may find adjusting strings authentic, while beginners may feel intimidated.

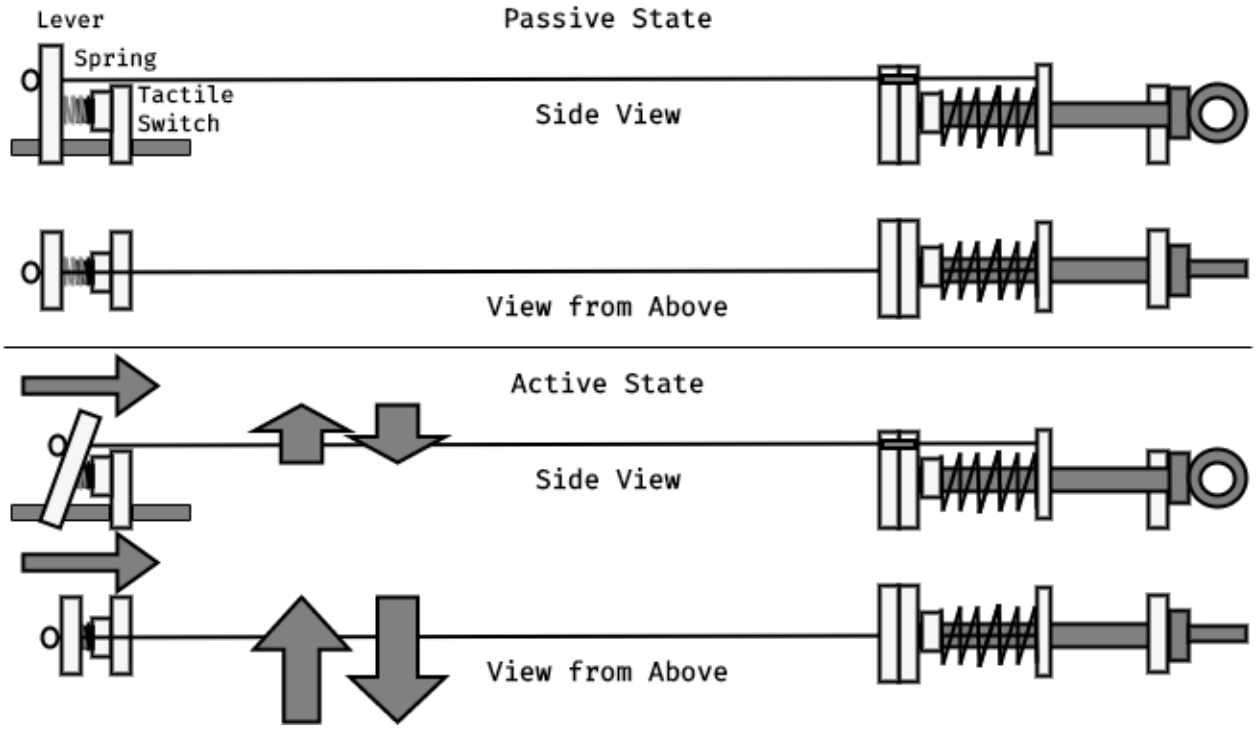


Figure- 44Lever-Switching *(Appendix/Lever_Mechanism.drawio)*

One of the most complex units in terms of physical design is our strum unit, including:

- Lever-Bridge (#1A/#3A),
- Unit Base (#2),
- Bridge (#1B/#3B),
- Six screws, nuts, springs, and levers,
- Hull-base (#4) and sides (#5A/B).

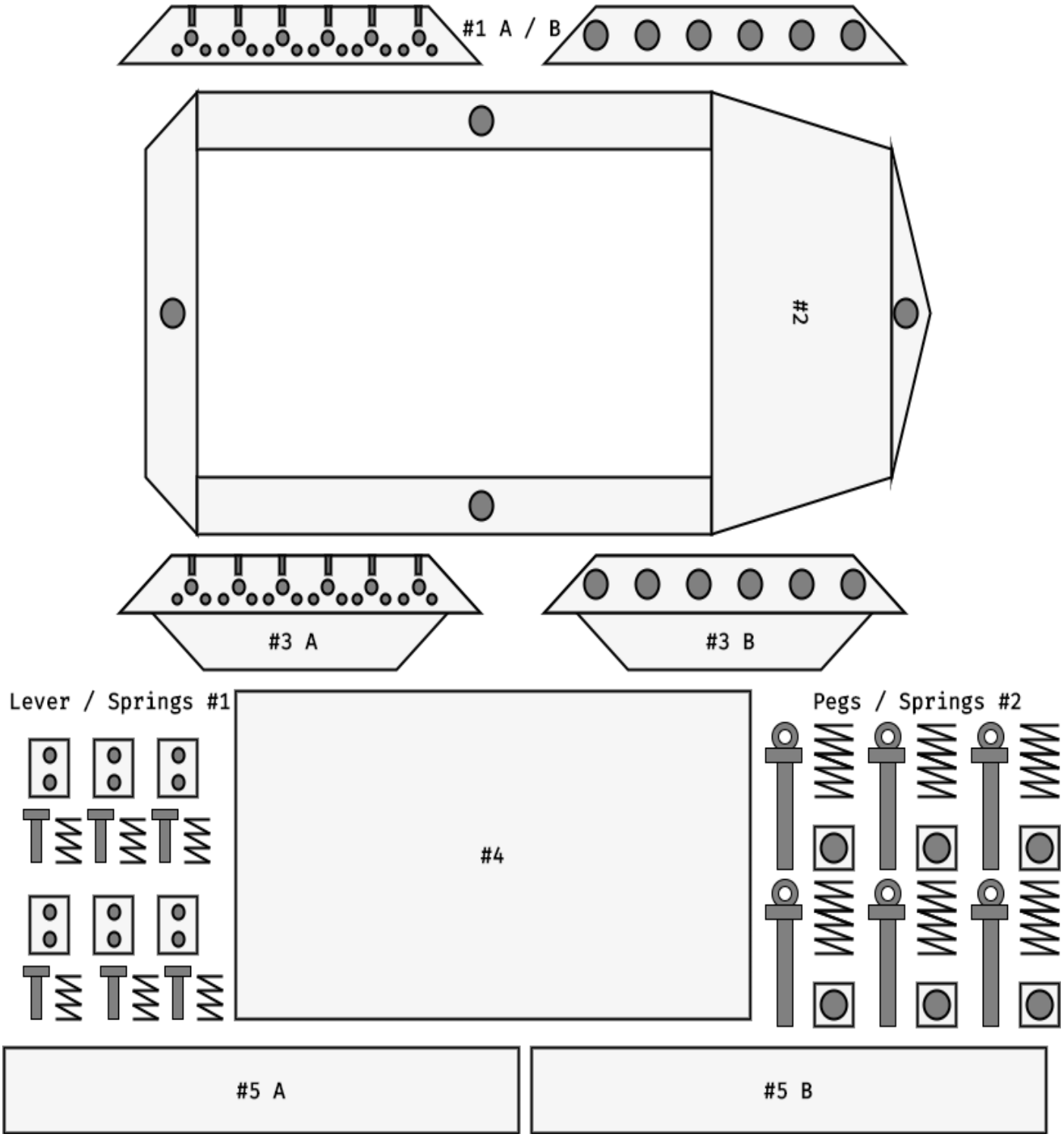


Figure-45 Components (Appendix/Strum-Components.drawio)

The strum is designed to be a ***separate replaceable unit***. The hull sits in the body cavity in a way that requires a minimum elevation from the surface (10mm–15mm) but provides space for comfortable strumming (~30mm).

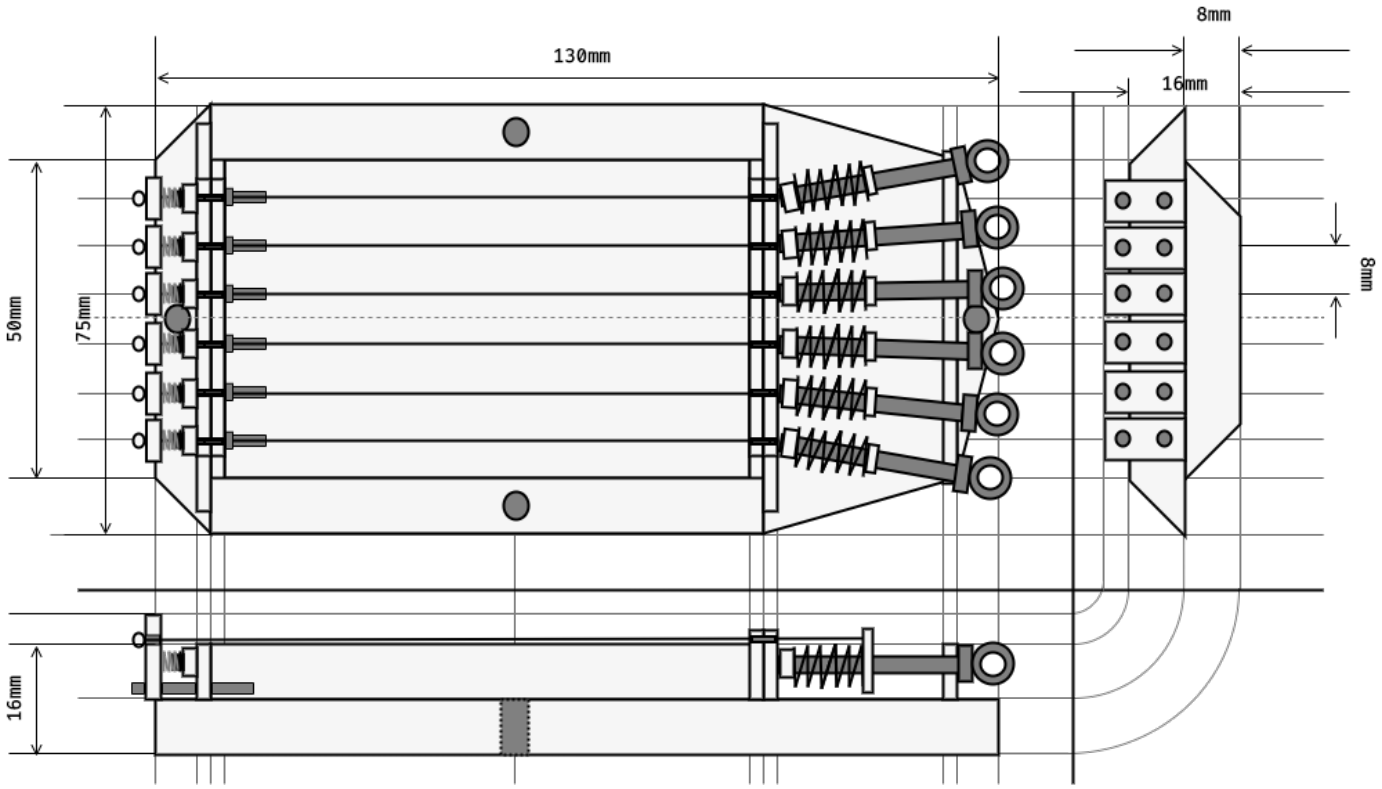


Figure-46 Strum Design (Appendix/Strum_Unit.drawio)

4.1.4.4. TOGGLE AND LEDS

Our specification requires users to interact with the device without disrupting traditional inputs, and USB keyboard communication may depend on the final communication protocol. The device must have an ON/OFF switch that disables any serial communication towards the computer. Additionally, three LEDs will provide further information about the device's state:

- RED: Device connected to USB,
- GREEN: USB communication is ON,
- BLUE: USB data transfer.

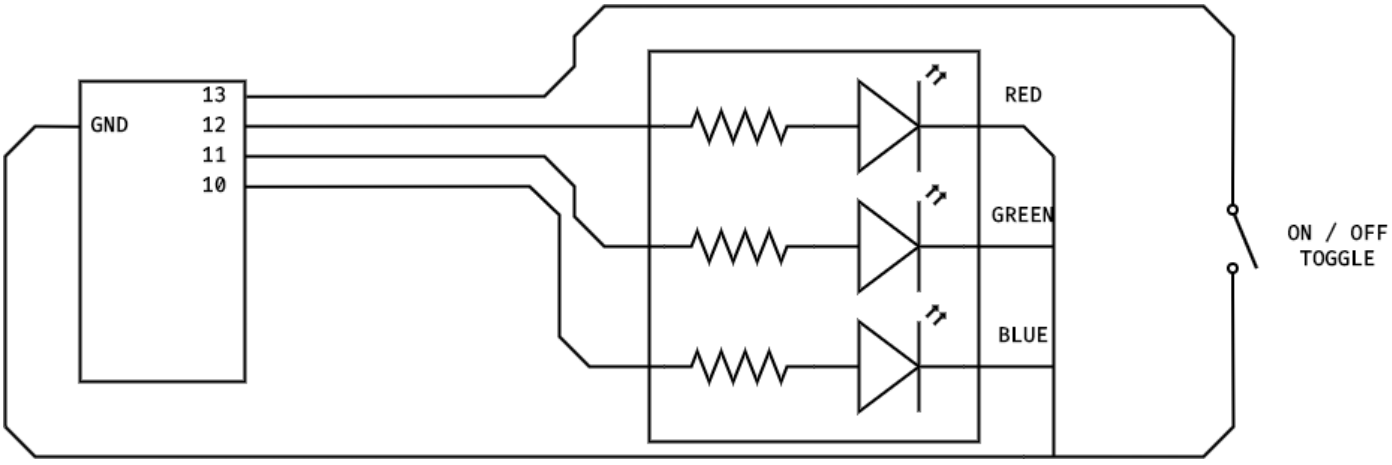


Figure-47 Toggle/LEDs (Appendix/Toggle_and_LEDs.drawio)

4.1.5 HEAD STOCK

Although users can interact with the fret by buttons and use chord patterns on the left-hand side, controls may not feel authentic enough. Therefore, users may string up the fretboard using the headstock, which provides fixed positions to ***attach strings as guidelines***. The headstock unit is an optional accessory and must be attachable/detachable to the neck.

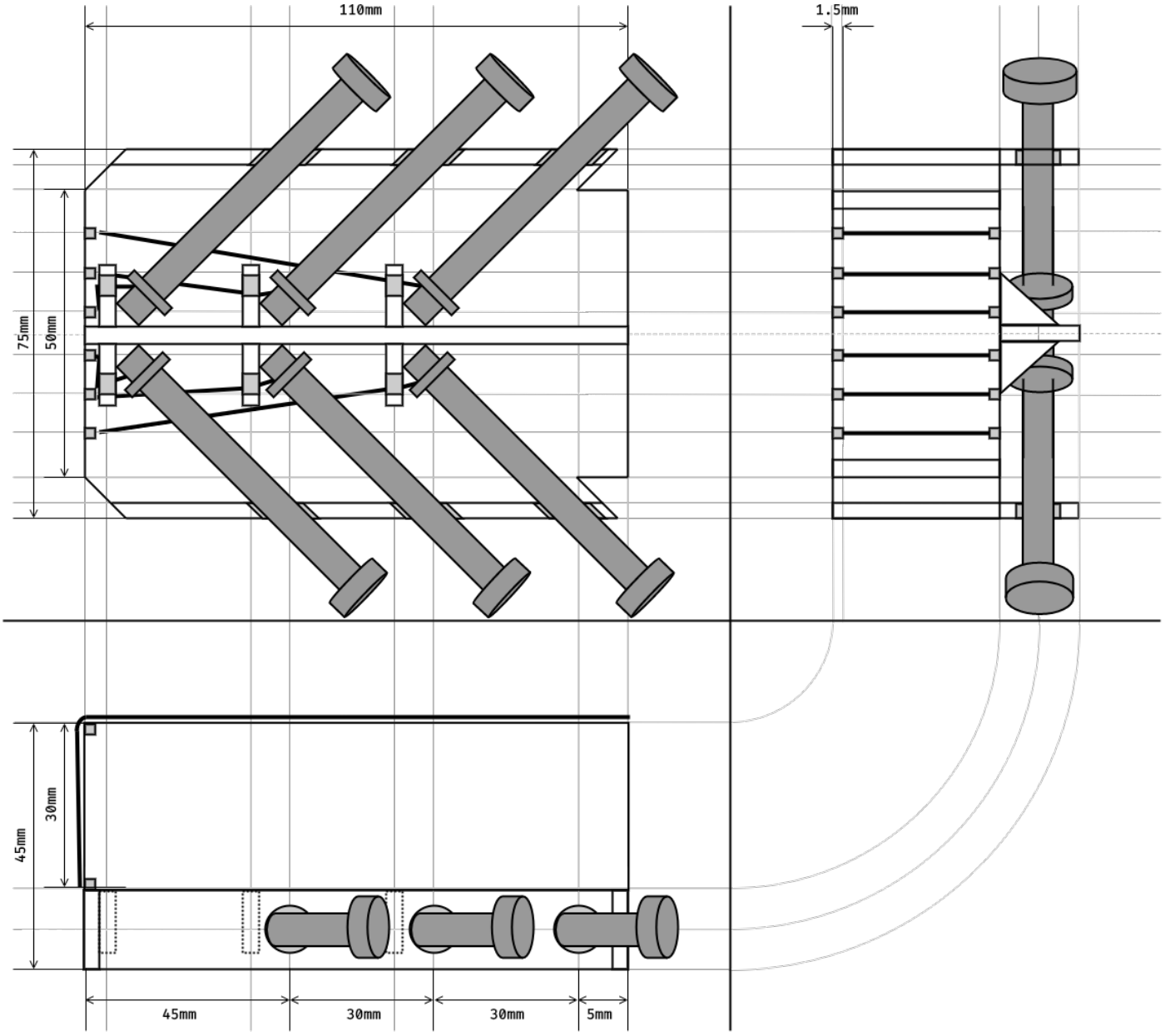


Figure-48 Head Stock (Appendix/Head_Stock.drawio)

4.2. FRONTEND

Based on our wireframes *(Section-1.3)*, the prototype's frontend will be developed using traditional web technologies, such as HTML5, CSS and JavaScript. Feature-rich commercial implementations would use frameworks for app state management (React/Redux) or CSS libraries (Sass/Less), but they will be omitted for now as they add extra complexity.

4.2.1. ACTIVITIES

The key activities that demonstrate the fundamental functionalities of our prototype are as follows:

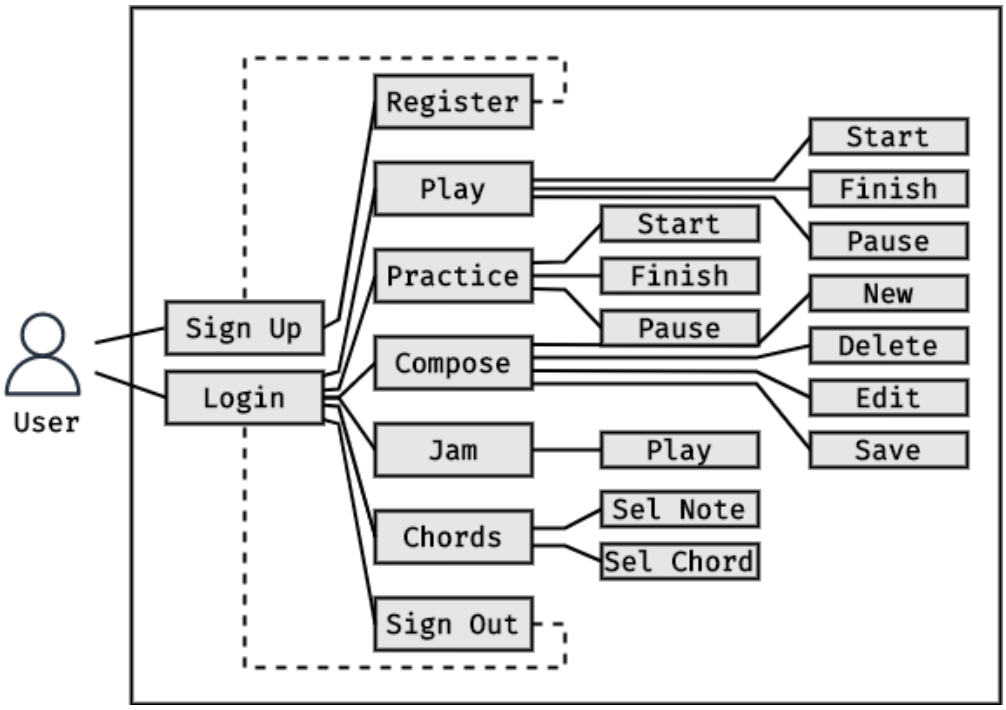


Figure-49 Activity Diagram

The signup/login procedures are not essential for the controller prototype; however, a well-rounded application should include those functionalities, as they clarify the complete processes involved.

4.2.2. USER EXPERIENCE

Because a diverse audience of varying demographics will use the application, the navigation and menu options must be straightforward and clear of unnecessary functionalities. A logical visual design must be used. For instance, the guitar strings will be *colour-coded with high contrast* to help the users play the instrument. We may apply this colour palette as standard guidelines for the rest of the app for consistency. However, using bright colours means the background and the theme should be dark to create a high-contrast UI. Unlike popular mobile-first applications, our desktop app uses a 16:9/16:10 ratio, and responsive CSS for tablet and mobile is out of our scope.

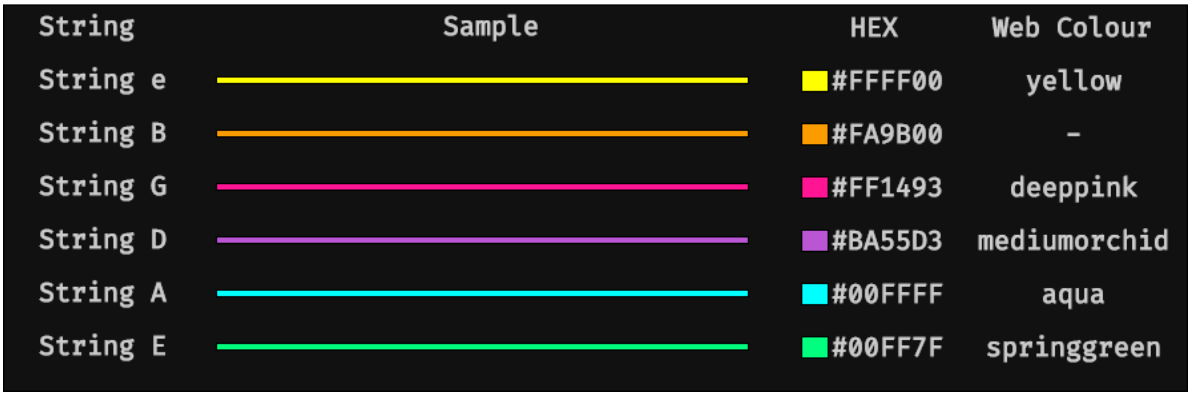


Figure-50 Colour-Palette

4.2.3. STATE AND UTILITIES

React with Redux would offer ways to pass data between components and maintain the state efficiently, but it would be overkill for our prototype. But using vanilla JS doesn't mean giving up on state management; for instance, the state may be stored in a ***global object***. While global objects are simple to implement, our multi-page structure requires holding data in cookies/local storage. It is vital to note that a commercial implementation should be written using frameworks because an extensive, feature-rich scalable app would be cumbersome to maintain using simplistic global states.

The application will rely heavily on DOM addressing and manipulation; some developers prefer libraries for such functionalities (Jquery). Our scenario doesn't justify Jquery's size (78KB–85KB), as only DOM addressing (select single/all), appending, and attribute manipulation would be used often. We may as well create our ***utility functions*** with these functionalities in a non-verbose fashion. We may even implement Jquery syntax, so other developers can easily interpret our code. The drawback of this method is that it is non-standard, and working with a team requires that every team member is familiar with these functions.

```
// DOM Selection
const $ = selector ⇒ document.querySelector(selector);                    // Select a Single DOM Element
const $all = selector ⇒ document.querySelectorAll(selector);              // Select ALL DOM Elements

// Append a Child Element
// Expect an Object as Parameter { tag: STR, className: STR, id: STR, parent: DOM, NS: bool }
const $append = (props) ⇒ {
    const { tag, className, id, parent, ns } = { ...props };              // Deconstruct Parameters
    const isDomElement = (e) ⇒ e instanceof Element || e instanceof HTMLDocument;

    if (!tag) throw Error("No TAG argument was given to create function.");       // Missing Tag
    if (!parent) throw Error("No PARENT argument was given to create function."); // Missing Parent
    if (!isDomElement(parent)) throw Error("Parent is not a DOM Element.");       // Parent is NOT a DOM Element

    const elem = ns                                                       // Has Different Name Space
        ? document.createElementNS(`http://www.w3.org/2000/svg`, tag)     // Create SVG Elements (svg, circle, line ... )
        : document.createElement(tag);                                    // Create Standard HTML DOM Element

    if (id) elem.id = id;                                                 // Add Id
    if (className) elem.classList.add(className);                         // Add Class
    parent.appendChild(elem);                                             // Append the Parent Element

    return elem;                                                          // Return with the Created DOM Element
}
```

Figure-25 Utilities *(Appendix/Code/util/misc.js)*

4.3. BACKEND

We may choose from several options regarding application architecture, such as WebSocket or GraphQL; however, this application will use ***RESTful architecture*** (Representational State Transfer). The main advantage of using RESTful is that it is scalable, stateless, easy to use, and built on HTML standards, using traditional (GET/PUT/POST) request verbs.

4.3.1. ROUTING

When the user interacts with the server, for instance, to access user information, request a guitar tablature or store the scores, the frontend will communicate these intentions with the server side. In exchange, the server sends back the requested information in ***JSON-formatted responses***. We can use JSON Web Tokens (JWT) for authentication, a modern, secure standard where each request header receives JWT tokens.

Register	POST	RESPONSE: SUCCESS: Redirect ⇒ Email Confirm, ERROR: { Message }
Login	POST	RESPONSE: SUCCESS: { JWT Token, User }, ERROR: { Message }
Confirm	POST	RESPONSE: SUCCESS: Redirect, ERROR: { Message }
User/:ID	GET	RESPONSE: SUCCESS: { User }, ERROR: { Message }
User/:ID	PUT	RESPONSE: SUCCESS: { User }, ERROR: { Message }
Profile/:ID	GET	RESPONSE: SUCCESS: { Profile }, ERROR: { Message }
Profile/:ID	PUT	RESPONSE: SUCCESS: { Profile }, ERROR: { Message }
Achievement/:ID	GET	RESPONSE: SUCCESS: { Achievement }, ERROR: { Message }
Tab/:ID	GET	RESPONSE: SUCCESS: { Tab }, ERROR: { Message }
Tab/:ID	POST	RESPONSE: SUCCESS: { Tab }, ERROR: { Message }
Tab/:ID	PUT	RESPONSE: SUCCESS: { Tab }, ERROR: { Message }
Tab/:ID	DEL	RESPONSE: SUCCESS: { DELETE_INFO }, ERROR: { Message }

Figure-52 Routing *(Appendix/API_Endpoints.drawio)*

4.3.2. DATABASE

Our RESTful application needs to communicate with a server, and various options are available, such as Flask, Django or NodeJS. ***NodeJS*** is an ideal candidate; built on the Chrome-V8 JavaScript engine, supports ES6 and is a web-native, easy-to-adapt server environment.

A commercial implementation would most likely be made around a ***MERN stack*** (Mongo-Express-React-NodeJS); therefore, using MongoDB for prototyping is a logical design decision. Nevertheless, our entity relationships may be implemented in any DBMS.

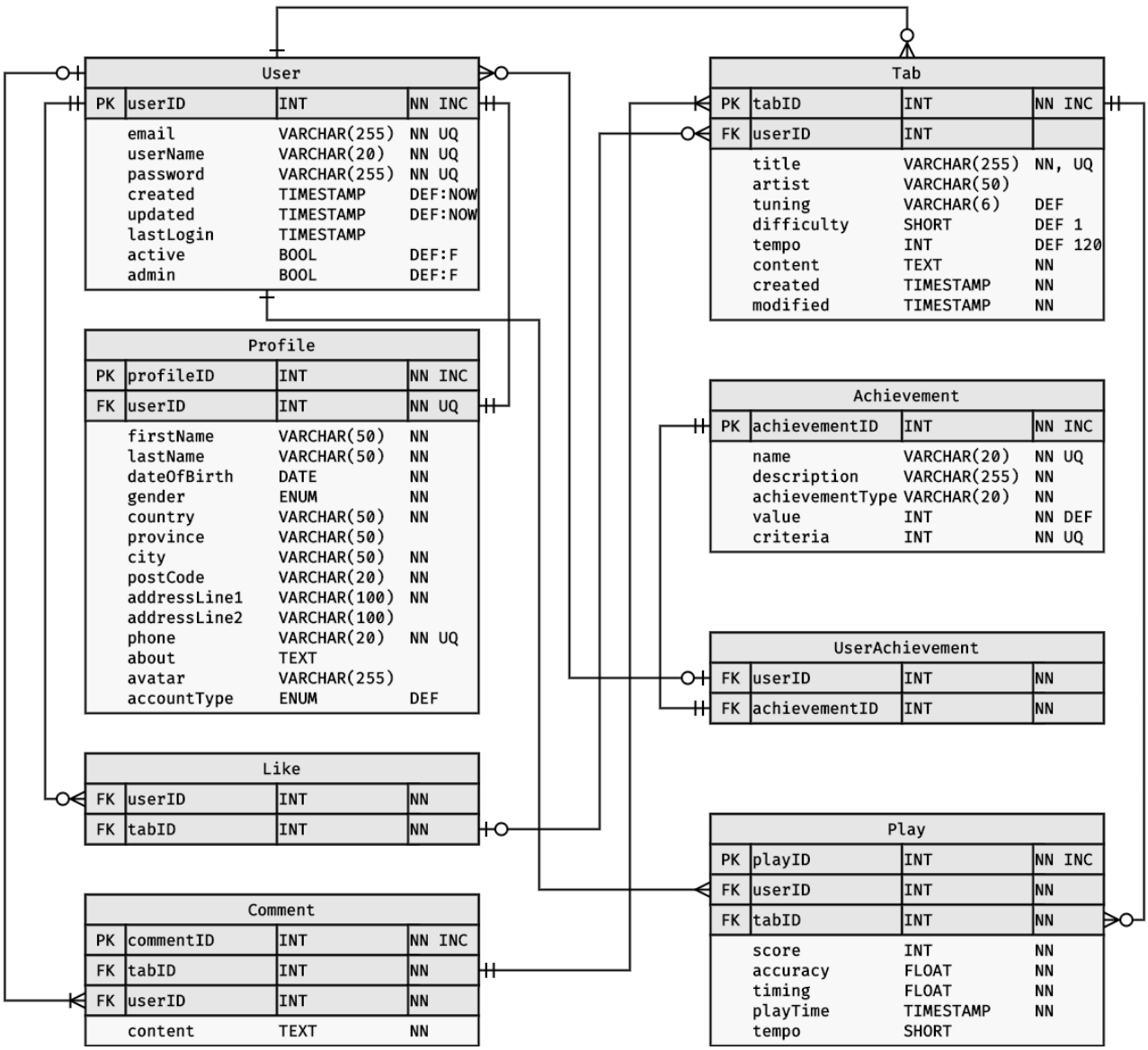


Figure-53 ER Diagram (Appendix/ER.drawio)

Because of the separation of concerns, the User table is used for authentication, and the Profile stores information about the clients. Deleting users may cause referencing problems (empty user references in score charts), and an active status property must be stored on the User table. A separate Profiles allows disconnecting personal information, as only the User table references it. Users can also choose avatars for their Profile, stored as strings (for simplicity). The contact information is in the profile details, which would otherwise receive a separate Address/Contact table and corresponding conjunction tables.

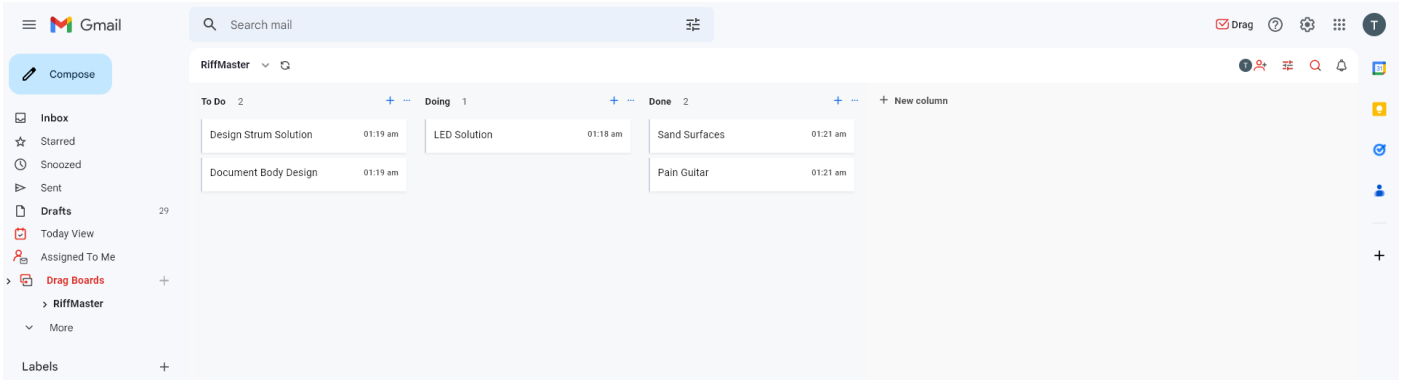
Tablatures are stored in the Tabs with the content and referenced by several other tables. Because we use a no-SQL database, there are options for non-normalised solutions, such as using a list of object IDs or hybrid snapshot storage. A complete commercial solution may feature tables like Message, Follow, Instrument, Contact, Skill, or Comment.

5. DEVELOPMENT

5.1. HARDWARE

The hardware design in previous chapters *(Section-4.1)* focused on ***handcrafted manufacturing processes*** with limited tools and materials. Despite the abundance of third-party services, DIY seemed the most feasible option, considering the time frame, cost and complexity. For instance, 3D and PCB designs require additional modelling, manufacturing and shipping time.

However, even handcrafting requires project management tools beyond Post-It Notes, and many software options are available to work with KanBan. Some of the most recommended tools are Monday (paid application), Trello, with limited features, or GoogleDraft. Draft and Excel are free and can perfectly cover our scenario's requirements.



	Date	Time	Task	Type	Mins	Comment
287	20.02.2023	16:30	Practice 1: General	Test	80	Get Familiar with the Instrument and Find Improvements
288	20.02.2023	18:40	Measure Audio Delay	Code	20	There is No Significant Time Delay (1 - 3ms), So the Delay Problem May be Caused by Something Else
289	20.02.2023	19:00	Troubleshooting Audio Card	Troubleshooting	55	Try Application on Other Devices: Significantly Less Delay (App is Fast Enough), Audo Card May be a Source of the Delay (Fixed: Firefox w
290	20.02.2023	22:00	Jam Option: Stop Notes 2	Code	50	Stop Notes from Playing When Finger Released on Position
291	20.02.2023	22:55	Equalizer Foundations	Code	230	Create Structure of the Equalizer Dinamically: 45 Columns / Text Node and 32 Beat Divs
292	21.02.2023	02:50	Equalizer Functions	Code	165	Equalizer Starts Animation on Columns, Stores Timer Functions, 2000ms Reverse, Short Circuit Timers on New Function Calls
293	21.02.2023	13:30	Design Head Stock	Design	75	Headstock Comprises of 5 Main Components, Upper / Lower Covers, 2 x Neck Support, Peg Placement
294	21.02.2023	14:45	Cut Head Stock Material	Build	115	Cut, Sand Head Stock Elements #1 - #5
295	21.02.2023	16:40	Adjusting and Gluing	Build	110	Adjust Sizes to Perfectly Fit Together
296	21.02.2023	20:15	Sanding Corners	Build	25	Sand Corners 45 Degree
297	21.02.2023	20:40	Drill and Triangles	Build	60	Drill Tuning Peg Holes and Small Triangles to Hold Screw in Position
298	22.02.2023	00:00	Practice 2: General	Test	30	Some Base Notes are Left Highlighted, Needs Debugging
299	22.02.2023	00:40	Debug Note Highlights	Code	55	The Highlight Removal was Incomplete and in Some Cases the Base String Highlight Stayed on (Fixed)
300	22.02.2023	01:35	Work Monitor	Code	110	Create a Labour Monitor in Python Notebook
301	22.02.2023	Next	Drill Holes for Neck Strings			
302	22.02.2023	Next	Handle for Tuning Pegs			
303			String Up Device			
304			Button Cap Cutting			
305			Button Cap Drilling			
306			Button Cap Grooves			
307			Button Cap Pasting			
308			Button Fret Groove Adjustment			
309			Lever Spring Change			
310			Fill Up Uneven Surfaces			
311			Sand Device Completely			
312			Repaint			
313						
314			Strum Components Diagram			
315			Strum Assembly Diagram			
316			Strum Paragraph			
317			Toggle and LED Electric Diagram			
318			Toggle Paragraph			
319			Head Stock Diagram			
320			Head Stock Paragraph			
321						
322						
323				Total Minutes	28205	
324				Total Hours	470.1	

Figure-54 Draft/Logbook *(Appendix/Logbook.drawio)*

5.1.1. MATERIALS

We will use ***polystyrene*** sheets for the prototype in two thickness sizes: 2mm for bending and 3mm for structure. Polystyrene sheets are available in most hardware shops; they are cheap (£30-50), bend on heat well, are relatively light, can be cut without specialised tools, and are excellent electrical insulators.

However, a commercial product should be built with ***carbon fibre***, which is more durable for heavy use. It doesn't snap, is flexible, and is ideal for manufacturing. Most importantly, it is more eco-friendly and would be an environmentally conscious choice for mass production.

5.1.2. CONSOLE DEVELOPMENT

The prototype development started with the ***fretboard*** because it is the heart of the device and the most likely failure point. As expected, the 240 components (switches/diodes), the wiring and the more than 1000 soldering points were complicated to cram in a 50/500mm space and required more than 60 hours of assembly.

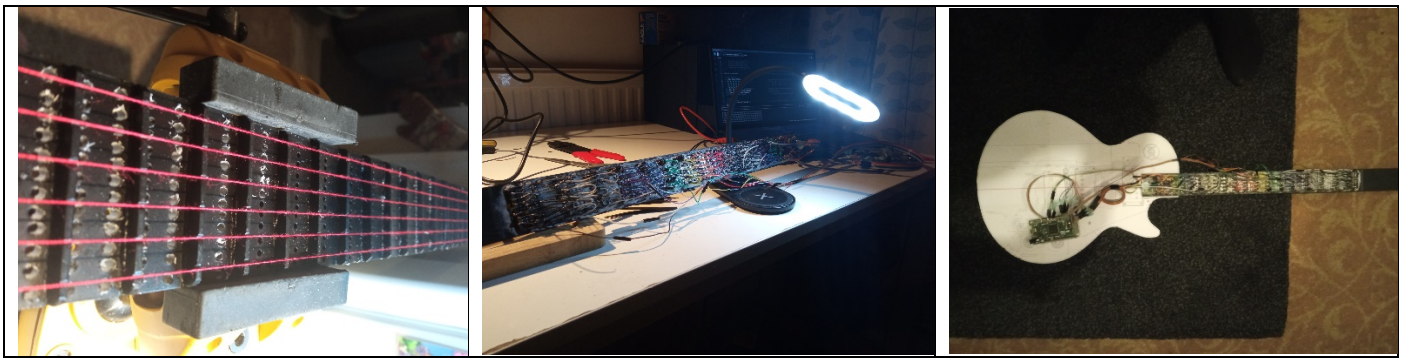


Figure-55 Building the Neck

Next, the console ***body*** covers were cut, and a wooden template body was created to bend the side walls to the appropriate curvatures because several attempts of free-handed bending resulted in broken pieces of already cut material.

Figure-56 Covers

The 2mm side wall material required many rounds of heating and clamping to the template, forcing it into shape.

Figure-57 Guitar Body

Body supporting components and the side/back cover of the neck were attached to the device, and nuts were milled and pasted to the drilling holes.

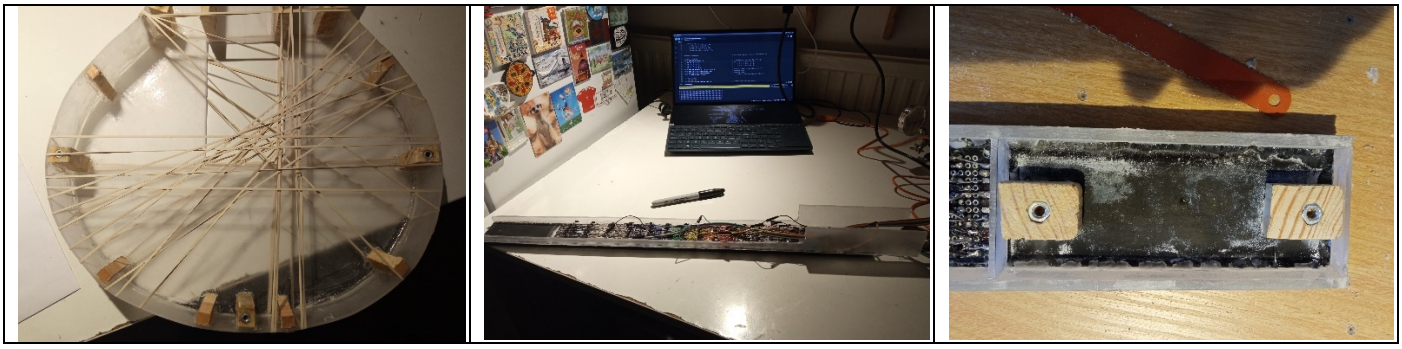


Figure-58 Supporting Components

The neck attachment was the riskiest part of the assembly. Should the neck have failed to hold together precisely, the whole project might have been endangered, as Gorilla glue and CT1 cannot be removed without severe damage. After coats of matt black paint, the microcontroller, wiring, USB socket, LEDs, and toggle also found their final place.

Figure-59 Housing

The strum unit was reasonably straightforward to assemble; however, the screws, levers, and string tension adjustments were time-consuming.

Figure-60 Strum Unit

Finally, we attached the ***headstock*** and strung up the device. As the strings were slipping aside on the switches, they received individually grooved and milled custom caps.

Figure-61 Head Stock

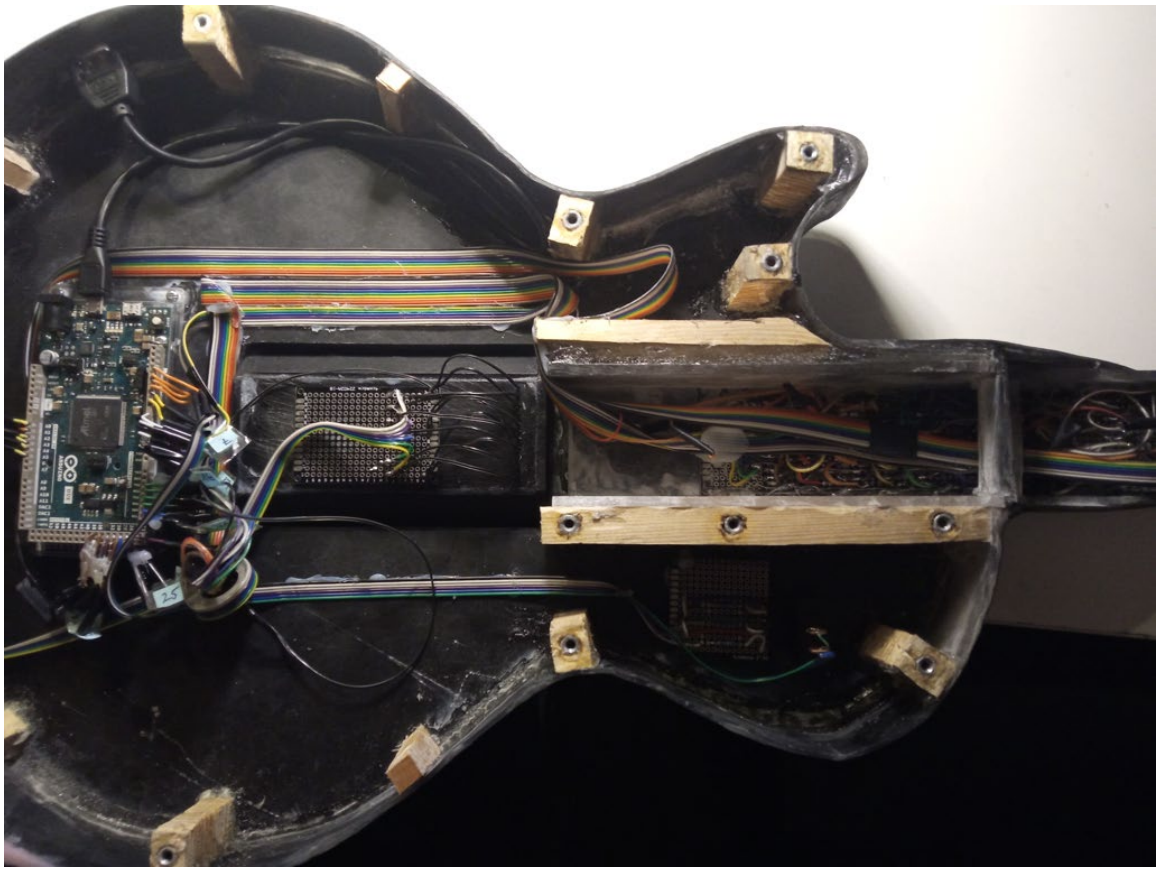


Figure-62 Inside

Figure-63 Final Prototype

The total manufacturing time (200+ hours) exceeded even the worst pessimistic estimations, and the early project start paid off its dividends. The complete ***photo gallery*** (180+ images) is available on the link in the Appendices.

5.1.3. MICROCONTROLLER

5.1.3.1 OPTIONS

Our microcontroller must be Wi-Fi-capable with at least 26 digital pins and bidirectional USB communication, leaving us with Arduino Leonardo, Due and Mega.

	Arduino Leonardo	Arduino Due	Arduino Mega
Microcontroller	ATmega32u4	SAM3X8E	ATmega2560
Clock Speed	16MHz	84 MHz	16MHz
Digital / Analogue PINs	20 / 12	54 / 12	54 / 16
Host USB / Device USB	No / Yes	Yes / Yes	Yes / No
Operating Voltage	5V	3V	5V
Flash Memory	32KB	512KB	256KB
Price Range in GBP	16 - 20	30 - 40	35 - 45

Figure-26 Comparision *(Appendix/Arduino_Selection.drawio)*

Arduino Due is not the most economical option. Still, it has the required amount of pins, a powerful processor, and, most importantly, can operate as a bidirectional USB host like conventional keyboards.

5.1.3.2 MICROCONTROLLER PROGRAMMING

Our ultimate goal is to read the device's state. When state change happens, they are communicated through USB as sequences of emulated keyboard presses. We may use simplified ***state machines*** to determine currently activated switches. We must differentiate previous and current conditions for frets/strums. We allocate a pair of 6/20 2D matrices and length-six arrays for the frets and strums. The initial values are reversed as we used pull-up resistors: zero means pressed, and one means released.

```
bool newFretState[ROW_LEN][COL_LEN] = {                      // State of Fret Positions Switches
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
};
bool oldFretState[ROW_LEN][COL_LEN] = {                      // Store Old State to Compare to New in Every Loop
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
};

// Right Hand States
bool newStrumState[6] = { 1, 1, 1, 1, 1, 1 };                // State of Strum String Switches
bool oldStrumState[6] = { 1, 1, 1, 1, 1, 1 };                // Store Old State to Compare to New in Every Loop
```

Figure-65 Initialisation *(Appendix/Code/Hardware/controller.ino)*

We activate the column pins with each loop and read individual row values, updating the previous and current button states.

```
// Set the Controller's State Variables (Global Vars) and Detect Fret and Strum Presses
void setState() {
    transactionState = false;
    for (byte ri = 0; ri < ROW_LEN; ri++) {                   // Traverse Rows (Strings)
        digitalWrite(ROW_PINS[ri], LOW);                      // Set Row Voltage LOW

        for (byte ci = 0; ci < COL_LEN; ci++) {               // Traverse Columns (Frets)
            oldFretState[ri][ci] = newFretState[ri][ci];      // Store Previous State

            const byte COL_READ = digitalRead(COLUMN_PINS[ci]);  // Read Column Voltage
            newFretState[ri][ci] = COL_READ;                  // Set New Fret State

            if (COL_READ == 0) transactionState = true;       // Set Blue LED ON when User Interacts with Fret Board
        }

        digitalWrite(ROW_PINS[ri], HIGH);                     // Reset Row Voltage HIGH
    }

    for (byte i = 0; i < 6; i++) {                            // Traverse Strums
        oldStrumState[i] = newStrumState[i];                  // Store Previous State
        newStrumState[i] = digitalRead(STRUM_PINS[i]);        // Set New Strum State

        if (newStrumState[i] == 0) transactionState = true;   // Set Blue LED ON when User Interacts with Fret Board
    }
}
```

Figure-66 State Update *(Appendix/Code/Hardware/controller.ino)*

Messages are sent by comparing pairs of array positions in current and previous readings and emulating keystroke actions with state change properties.

```
// Send State Changes as Keyboard Events
void sendState() {
    // Read Fret State
    for (byte ri = 0; ri < ROW_LEN; ri++) {                    // Iterate Rows (Strings)
        for (byte ci = 0; ci < COL_LEN; ci++) {                // Iterate Columns (Frets)
            bool oldState = oldFretState[ri][ci];               // Get Old Fret Position
            bool newState = newFretState[ri][ci];               // Get New Fret Position

            if (oldState ≠ newState) {                         // If Fret State Changed
                bool event = !newState;                        // Negate Pressed State to get HIGH = 1 and LOW = 0
                int fret = ci + 1;                             // Add One as Frets are Numbered from 1 - 20
                int string = ri + 1;                           // Add One as Strings are Numbered from 1 - 6

                Keyboard.print(event);
                if (fret < 10) Keyboard.print("0");            // Pad Single Digits with a 0
                Keyboard.print(fret);
                Keyboard.println(string);
            }
        }
    }

    // Read Strum State
    for (byte i = 0; i < 6; i++) {                             // Traverse Strums
        if (oldStrumState[i] ≠ newStrumState[i]) {             // If Change Detected
            bool event = !newStrumState[i];                    // Negate Pressed State to get HIGH = 1 and LOW = 0
            Keyboard.print(event);
            Keyboard.print("00");
            Keyboard.println(i + 1);
        }
    }
```

Figure-6727 Messaging *(Appendix/Code/Hardware/controller.ino)*

5.1.3.3. DEBOUNCING

As discussed previously *(Section-3.4.4)*, momentary tactile switches are not guaranteed to be immune to physical bouncing and may trigger unsolicited actions from the controller. Furthermore, our device has two types of inputs with directly opposing behaviour.

Fret Buttons

From the traditional perspective, inputs register events ***when pressed***, just like our fret switches. When activation happens, we assume that state changes are intentional after a particular interval, typically ~50-100ms.

Strum Switches

Unlike frets, strum switches are activated after strings are put under pressure and ***released***. We may also expect extra noise, as our switches use strings, levers and springs that may oscillate. After the first release, we measure the interval of state changes and register if the last change happened after the ***debounce time allowance***.

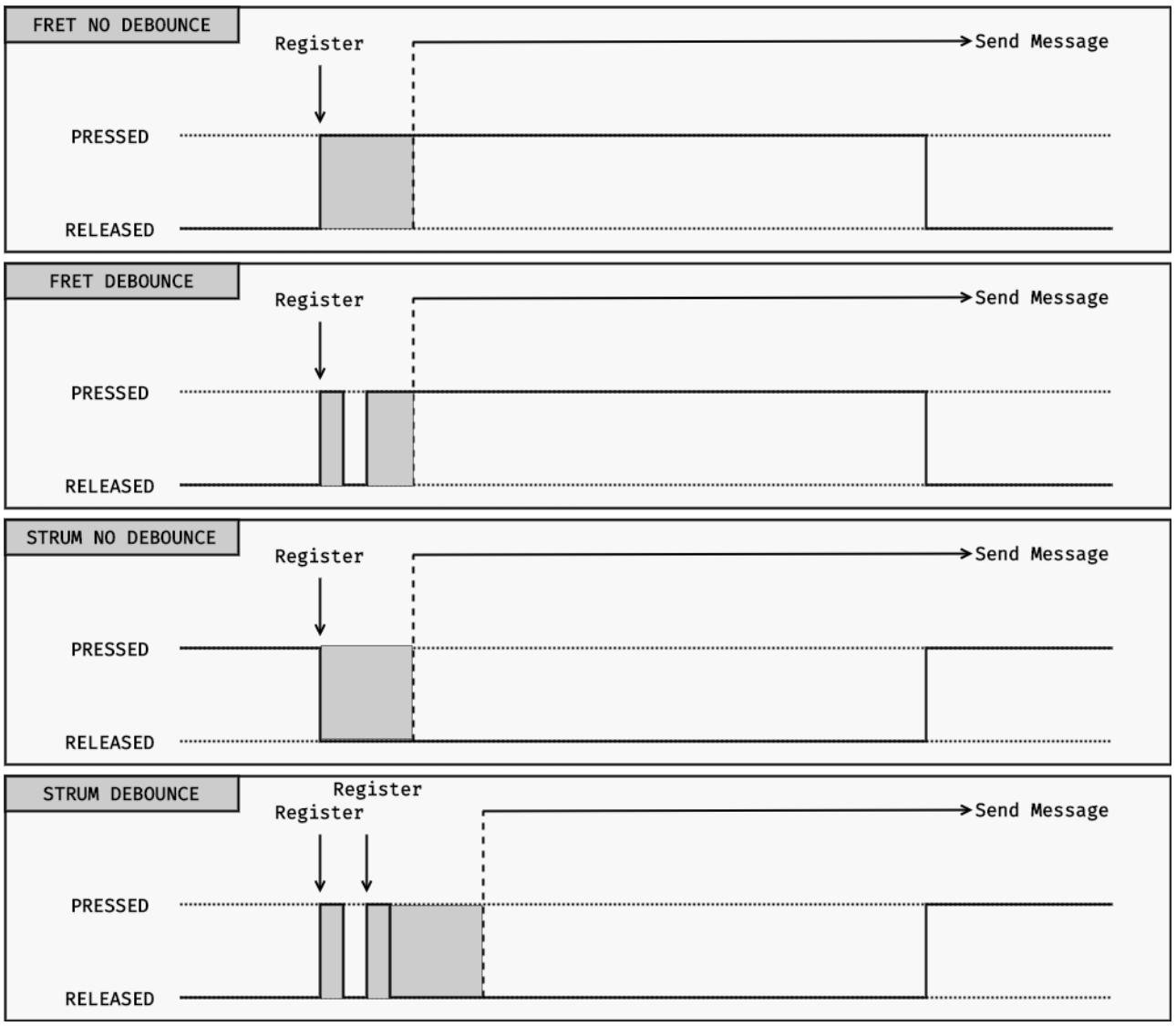


Figure-68 Debouncing

```
// Send State Changes as Keyboard Events
void sendState() {
    const unsigned long currentTime = millis();                      // Get the Current Time
    // Read Fret State
    for (byte ri = 0; ri < ROW_LEN; ri++) {                          // Iterate Rows (Strings)
        for (byte ci = 0; ci < COL_LEN; ci++) {                      // Iterate Columns (Frets)
            bool oldState = oldFretState[ri][ci];                    // Get Old Fret Position
            bool newState = newFretState[ri][ci];                    // Get New Fret Position
            const unsigned long lastPressed = debounceFret[ri][ci];  // Read Last Time a Particular Fret Has Been Pressed
            const unsigned long elapsedTime =  currentTime - lastPressed; // Calculate Elapsed Time
            if (oldState ≠ newState) {
                debounceFret[ri][ci] =  currentTime;                 // Register Fret Interaction Time for Debounce If State Changes
                if (elapsedTime > maxAllowedDebounceTime) {          // Send Keyboard Presses
                bool event = !newState;                              // Negate Pressed State to get HIGH = 1 and LOW = 0
                int fret = ci + 1;                                   // Add One as Frets are Numbered from 1 - 20
                int string = ri + 1;                                 // Add One as Strings are Numbered from 1 -
                Keyboard.print(event);                               // Add Event
                if (fret < 10) Keyboard.print("0");                  // Pad Single Digits with a 0
                  Keyboard.print(fret);                              // Add Fret Number
                  Keyboard.println(string);                          // Add String Number
                }
            }
        }
    }
```

Figure-69 Debounce Handling

5.1.3.4. PROTOCOL

As the messages between console device and host are sent as keypresses, we must agree on a suitable and efficient protocol.

MIDI/OSI: the commercial version of the guitar console would use MIDI (industry standard). However, we agreed on a simplified communication for prototypes, and MIDI would be difficult and time-consuming to implement.

Binary: Binary is one of the most efficient ways of communicating with the computer, but translating state changes to binary may end up with longer messages than other solutions.

Textual: Text-based communication offers an easy-to-implement solution, as we can describe specific states with relatively few characters. Individual keypresses may take some time to travel through the DOMs event propagation hierarchy; shorter messages are more efficient.

Numerical: solutions are essentially specified textual solutions with the same strengths as sending text keypresses but with restricted data types.

We used the numerical solution for its simplicity and efficiency. Messages consist of four digits: digit #1 is the event (pressed/released), digits #2/#3 signify the fret position, and digit #4 is the string number.

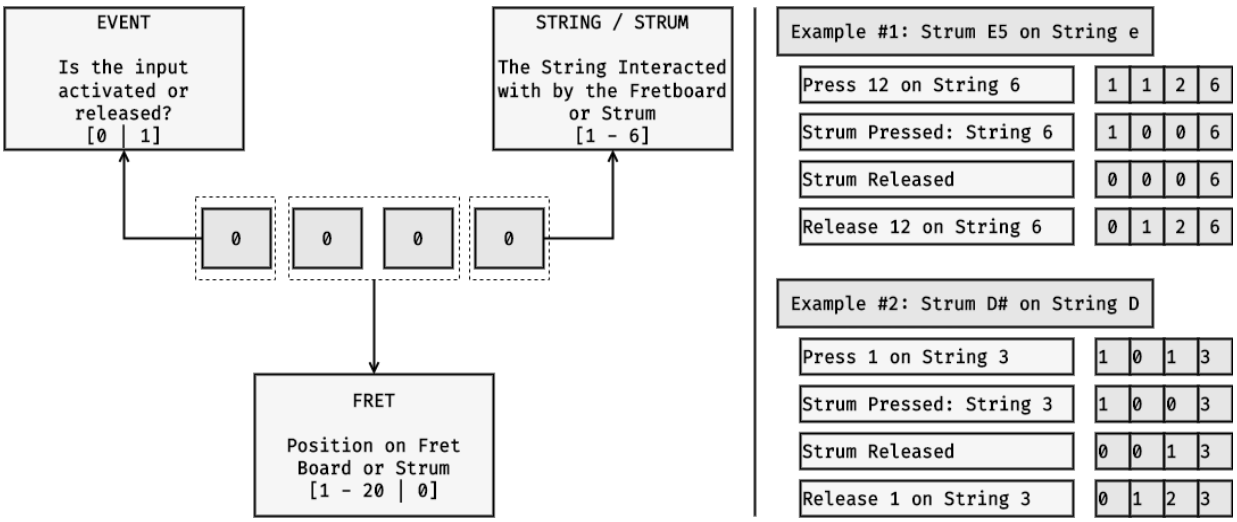


Figure-70 Protocol (Appendix/Protocol.drawio)

The USB connection is visible to the computer as a conventional keyboard, and the *browser' key-down listener* is responsible for monitoring, translating messages and invoking appropriate functions.

```
// Read a Print Line from Console or a Sequence of Keyboard Events
function controllerListener(event) {
    if (!app.musicAgreement) return;

    // Listen to Only Key Downs as Message is Passed as a Sequence of KeyStrokes
    if (event.type !== "keydown") {
        // If Key is Digit
        if (event.key >= "0" && event.key <= "9") {                                    // If Digit
            app.controllerState.message += event.key;                                  // Concatinate Message

            if (app.controllerState.message.length === 4) {                            // Attempt to Decode Every 4 Digit
                translateConsoleMessage(app.controllerState.message);                  // Call Decode
            }
        }
        else app.controllerState.message = "";                                         // Otherwise Clear Message
    }
}


// Translate a Keypress Message Containing 4 Digits
function translateConsoleMessage(message) {
    const event = parseInt(message[0]);                                                // Pressed or Released
    const fret =  parseInt(message[1] + message[2]);                                   // Fret 0 - 20
    const string = parseInt(message[3]);                                               // String 0 - 6

    if (event > 1 || string > 6 || string === 0 || fret > 20) return app.controllerState.message = ""; // Reset

    // Strum
    if (fret === 0) {
        handleStrumActivated(event, string);                                           // Strum Event
    }

    // Fret Position
    else {                                                                             // Fret Event
        let positionsOnString = app.controllerState.highestFretPositions[string - 1];  // Get Finger Positions on String
        // If Fret Pressed
        if (event === 1) {
            positionsOnString.push(fret);                                              // Append Current Positions
            positionsString = positionsOnString.sort((a, b) => a - b);                 // Sort Ascending
            positionsOnString = [ ...(new Set(positionsOnString)) ];                   // Disallow Repeatition
            app.controllerState.highestFretPositions[string - 1] = positionsOnString;  // Save Position
        }
```

Figure-71 Deciphering (Appendix/Frontend/console.js)

5.2. BACKEND

Requests flow through a NodeExpress pipeline in the following sequence:

1. JWT signature validation restricts access, and calls are logged,
2. The server routes our requests, and parameters and body content are validated,
3. The server communicates to the database according to the CRUD method,
4. The server sends back the response to the client.

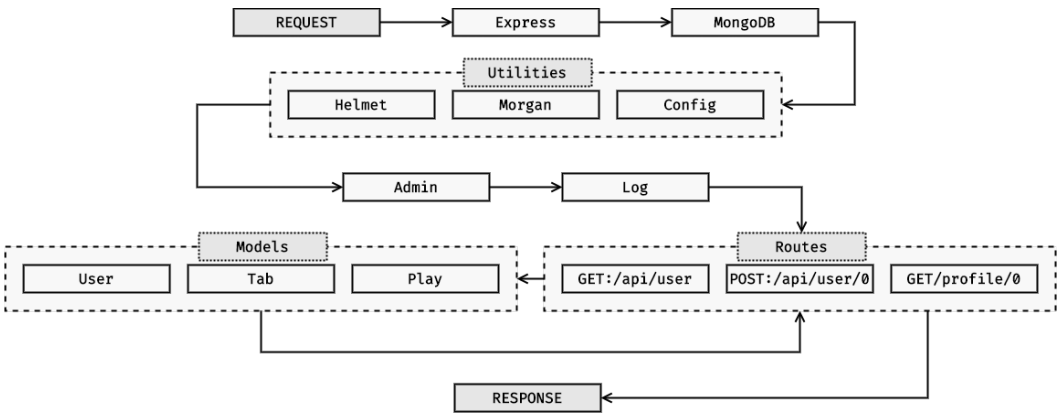


Figure-72 Pipeline (Appendix/Pipeline.drawio)

5.2.1. SCHEMAS

Although we can use SQL-style referencing in MongoDB, one benefit is breaking away from strict referential table structures with two methods. ***Embedding*** is when the parent table stores information about a referenced table, providing snapshots of related data. This high-performance method may sometimes be memory intensive. ***Object references*** require less memory but may necessitate multiple requests to the database. For instance, instead of creating separate conjunction tables for user achievements, we can store them in a list of object IDs and query them as needed.

```
    avatar: {                                                   // AVATAR: STRING MAX: 255
        type: String,
        maxlength: 255,
        trim: true,
    },
    achievements: {                                             // ACHIEVEMENTS: ARRAY OF OBJECTIDS
        type: [ mongoose.Schema.Types.ObjectId ],
        ref: "Achievement",
        default: []
    },
    accountType: {                                              // ACCOUNTTYPE: ENUM REQIURED, LOWERCASE
        type: String,
        enum: ACCOUNT_TYPES,
        default: "free",                                        // Accounts Default Free
        lowercase: true,
        trim: true
    },
});
```

Figure-73 Profile Schema *(Appendix/Code/Backend/Models/profile.js)*

5.2.2. ENDPOINTS

Request results are returned in JSON format with the properties: success, message, and data. Some API endpoints must redirect our users, such as when registration is complete or when email links are activated to confirm users. This type of ***two-phase registration*** is useful against resource consumption attacks (DOS) because an attacker cannot create floods of registered users, as email addresses must be unique and confirmed.

```
    // Send Email Confirmation Link
    const EMAIL_ADDRESS = config.get('emailAddress');           // Get Email From Environmental Variables
    const EMAIL_PASSWORD = config.get('emailPassword');         // Get Email Server Password From Environmental Variables
    const EMAIL_SERVER = config.get('emailServer');             // Get Server Type From Environmental Variables
    const jwtPrivateKey = config.get("jwtPrivateKey");          // Get JWT Private Key
    const tokenString = jwt.sign(req.body.user, jwtPrivateKey); // Create a JWT Signature
    const url = `http://127.0.0.1:${ process.env.PORT }/api/confirm/`; // TEMP URL to Localhost
    const confirmationLinkURL = url + tokenString;
    const token = await new Token({ token: tokenString }, config.get('jwtPrivateKey')); // Create Token on Database
    await token.save();                                         // Save Token

    const mailOptions = {                                       // Email Specifications
        from: EMAIL_ADDRESS,
        to: email,                                              // From Request
        subject: 'RiffMaster | Email Address Verification',
        html: `
            <h1>RiffMaster Account Verification</h1>
            <h2>Welcome to my RiffMaster subscriptions!</h2>
            <p>
                We couldn't be more excited to have you join our amazing guitar learning platform!
                <br />
                Please verify your email address by clicking on the link below.
            </p>
            <a href="${ confirmationLinkURL }">Verify Email Address</a>`
    };
```

Figure-74 Nodemailer *(Appendix/Code/Backend/Routes/Subscribe.js)*

Confirmation email:

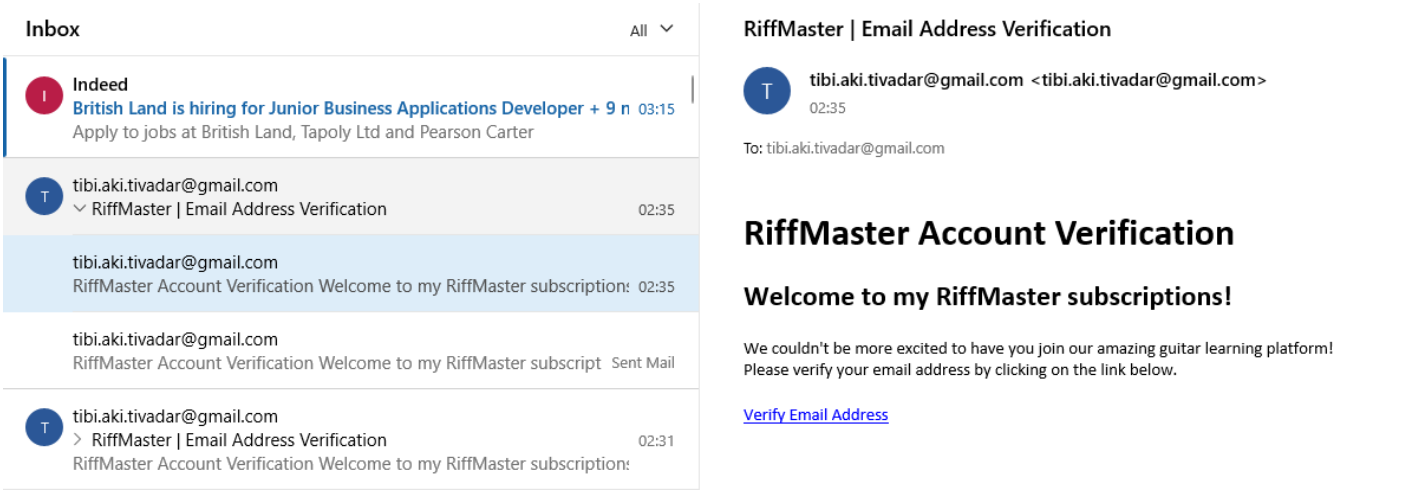


Figure-75 Confirmation

5.2.3. SECURITY

JSON Web Tokens (JWT) are appended in request headers for secure client-server communication. JWT stores information about the data payload, like user ID, admin privileges, expiry, and encryption algorithm and generates unique signatures. JWT private keys are stored in the server's ***environment variables***, so they cannot be decrypted by third parties unless negligently exposed by the Git repository with uncareful commits.

When users register on RiffMaster, they must send their authentication information to our server, including the password. Two directly opposing opinions on approaching this problem are ***hashing*** on the application's client and server sides. The first approach asserts that no password information should be sent to a server in plain-text format, and passwords should be hashed before submitting a request.

This approach dramatically reduces the chances of ***Man-in-the-Middle*** attacks intercepting our password on transfers. But they are vulnerable to ***Pass-the-Hash*** attacks, as the passwords stored on the database have already been hashed for any malevolent activities. They can also be used for ***Replay*** attacks, where the complete HTTP request is recorded and replayed, gaining access to unauthorised resources. Regarding JWT, the accepted method is to sign and verify signatures on the backend of our application, as private keys are secure, and ***tampered tokens are automatically refused***.

```
60    // Hash Password
61    const salt = await bcrypt.genSalt(10);                    // Generate Salt
62    const hashed = await bcrypt.hash(req.body.password, salt);    // Encrypt with Salt
63
64    const user = new User(req.body);                          // Create User
65    user.password = hashed;                                   // Store Encrypted Password
66    await user.save();
67
68    const updated = { ...user }._doc;                         // Dereference User (Cannot Delete Password on Prototype)
```

Figure-76 Encryption *(Appendix/Code/Backend/Routes/user.js)*

5.3. FRONT-END

Most app pages have unusual requirements; algorithmic solutions take some out-of-the-box thinking and unconventional approaches. Therefore, only these specific solutions will be documented in-depth and more standard parts are omitted for brevity, as the end product extends several thousand lines of code.

5.3.1. MULTIMEDIA

5.3.1.1. AUDIO EDITING

We use Audacity for audio editing. As the guitar needs to produce sounds on the Jam Session page, our first audio source is the guitar notes. Unfortunately, finding quality licence-free audio with the ***complete E2-C6 note range*** is futile. Therefore, we custom-made our metal-string acoustic guitar samples.

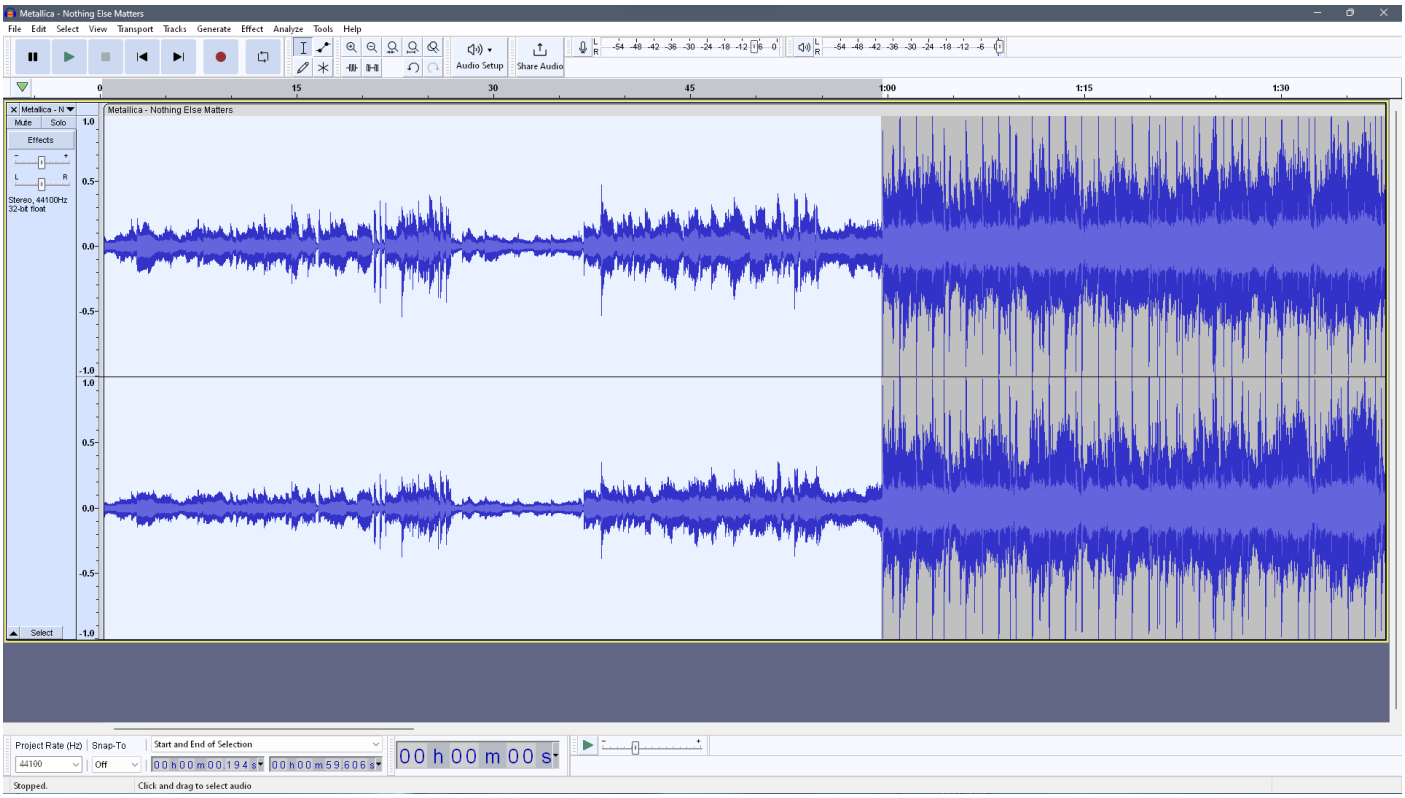


Figure-77 Audio Editing

5.3.1.2. CHORDS AND AVATARS

We employed an extensive, freely available JSON list with (10000+) finger patterns to display chords in the Chord Explorer.

```
{
    "C":[
        {
            "positions":["x","3","2","0","1","0"],
            "fingerings":[["0","3","2","0","1","0"]]
        },
        {
            "positions":["x","3","5","5","5","3"],
            "fingerings":[["0","1","2","3","4","1"]]
        }
        ,{
            "positions":["x","3","2","0","1","3"],
            "fingerings":[["0","3","2","0","1","4"]]}
```

Figure-78 Chords.JSON

Lastly, we created licence-free images with ***MidJourney's Artificial Intelligence*** application. We generated a range of guitar-related avatars from which users can choose.

Figure-79 Avatars

5.3.2. VISUAL DESIGN

One pivotal point of our pages is the extensive use of gradients and predefined colour sets.

```
menu li:nth-child(1) a:hover { background: linear-gradient(360deg, rgb(255, 255, 118), yellow, rgba(255, 255, 0, 0.05)); }
menu li:nth-child(2) a:hover { background: linear-gradient(360deg, rgb(253, 182, 66), rgb(250, 155, 0), rgba(250, 155, 0, 0.05)); }
menu li:nth-child(3) a:hover { background: linear-gradient(360deg, rgb(244, 79, 167), deeppink, rgba(255, 20, 145, 0.05)); }
menu li:nth-child(4) a:hover { background: linear-gradient(360deg, rgb(184, 111, 202), mediumorchid, rgba(186, 85, 211, 0.05)); }
menu li:nth-child(5) a:hover { background: linear-gradient(360deg, rgb(109, 255, 255), aqua, rgba(0, 255, 255, 0.05)); }
menu li:nth-child(6) a:hover { background: linear-gradient(360deg, rgb(123, 255, 189), springgreen, rgba(0, 255, 128, 0.05)); }
```

Figure-80 Gradients

The application draws inspiration from ***neumorphic*** web design compromising between ***flat composition*** and realistic ***skeuomorphism***. Elements like buttons, inputs and focused components will have soft 3D UIs with slightly rounded corners. 3D features are built by mixing gradients, partial transparency and inner/outer shadows.

```css
button {
    position: relative;
    height: 50px;
    width: 50px;
    margin: 0 10px;
    background-color: transparent;
    border: 2px solid #333;
    border-radius: 10px;
    box-shadow: 10px 10px 20px black, 10px 10px 20px inset black, -10px -10px 10px rgba(51, 51, 51, 0.3), -10px -10px 20px inset rgba(51, 51, 51, 0.5);
    color: #aaa;
}

button:hover {
    color: springgreen;
    box-shadow: -10px -10px 20px black, -10px -10px 20px inset black, 10px 10px 10px rgba(51, 51, 51, 0.3), 10px 10px 20px inset rgba(51, 51, 51, 0.5);
    cursor: pointer;
}
```

Figure-81 Neumorphism *(Appendix/Code/Frontend/index.css)*

5.3.3. REGISTRATION AND LOGIN

When users first visit our landing page, they are directed to a simple login. Passwords are not displayed, but users can opt for the reveal password icon to check input texts. When users send login requests to the server and credentials are correct, the server sends back a JSON signature token with the complete user profile object. ***Without this object, the page automatically redirects to login.***

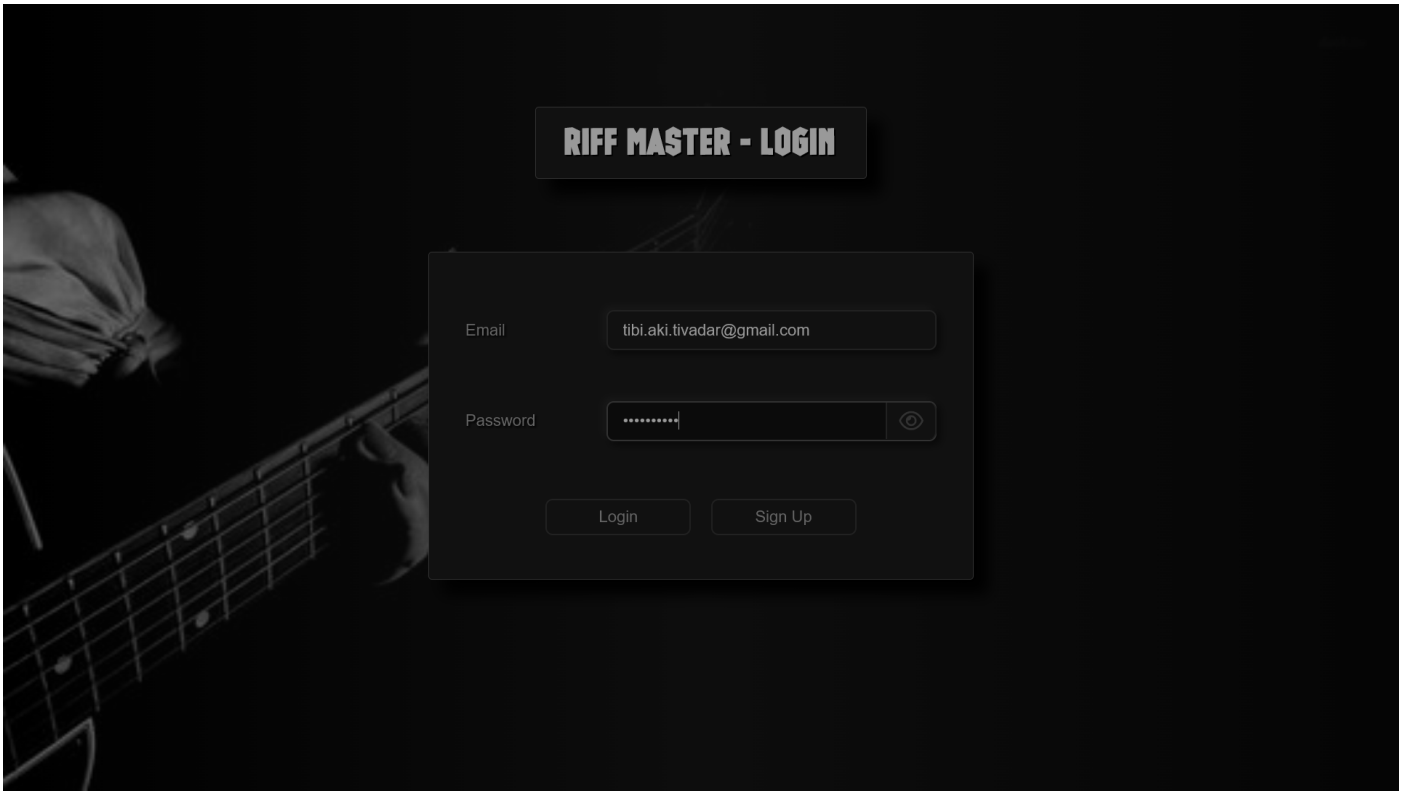


Figure-82 Login

We used *multi-step registration*. First, when the users create an account, the app redirects to the signup.

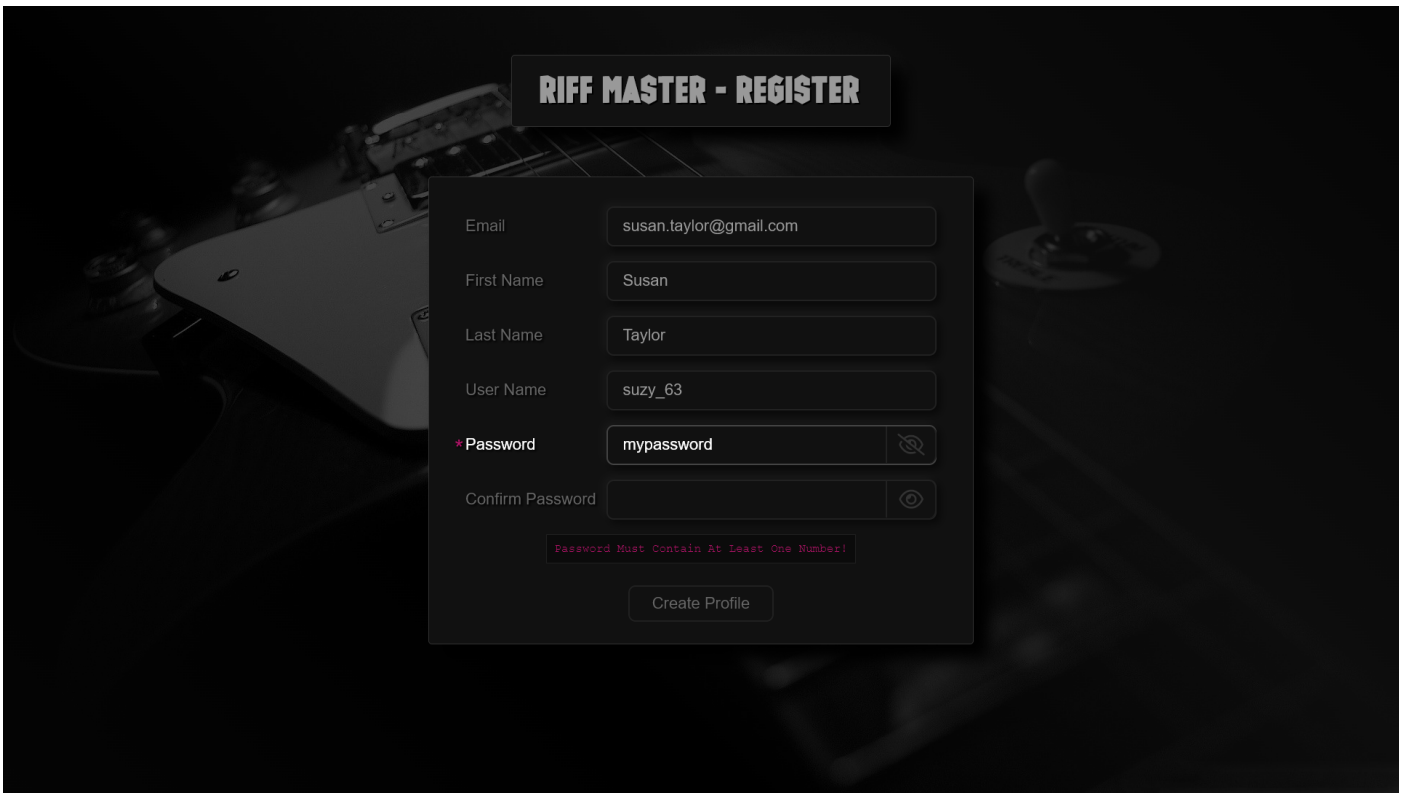


Figure-83 Register

After successful input validation, we ensure no user exists with the same email/user name and password-confirm inputs match. Errors are highlighted, and messages are shown for error clarification.

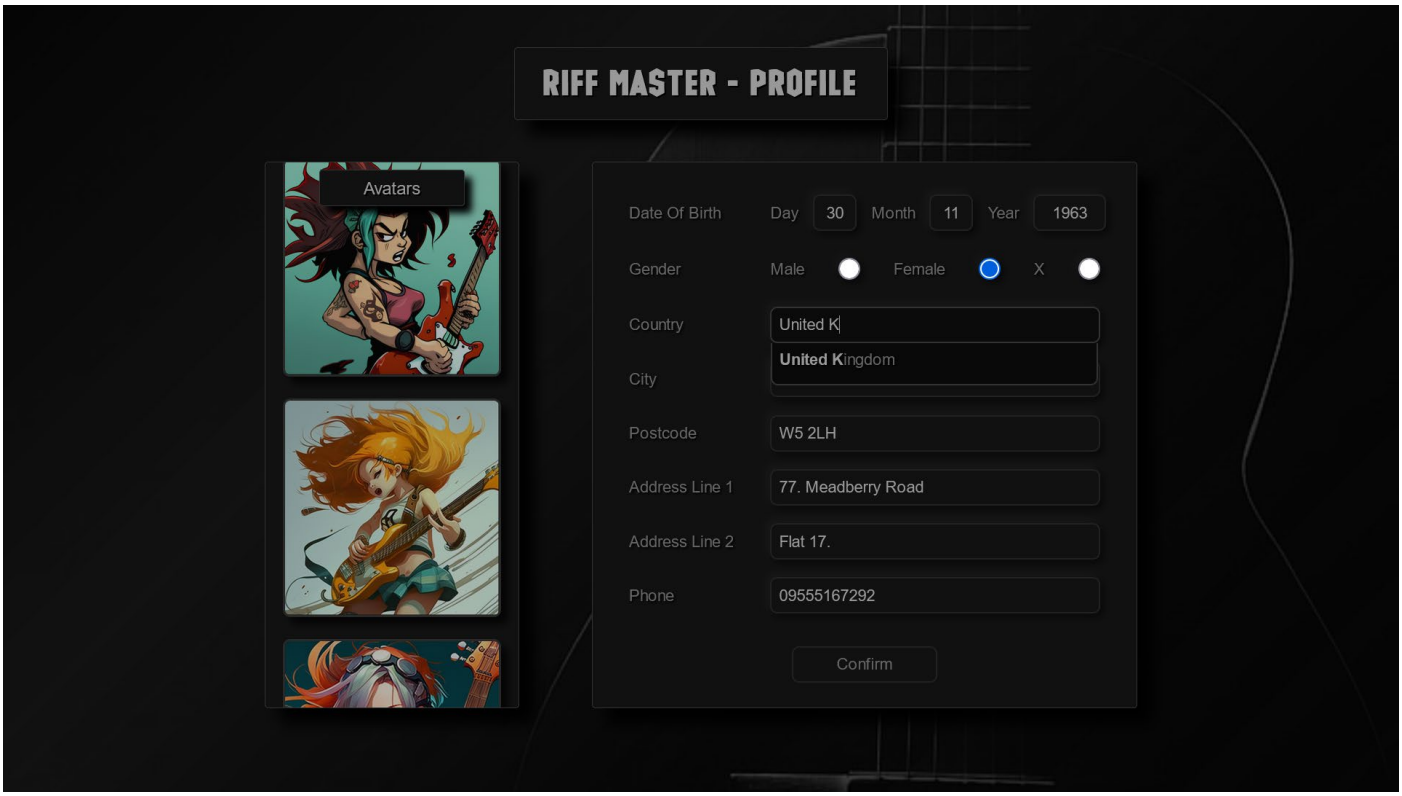


Figure-84 Profile

The profile layout consists of info and an avatar. The application is age restricted and calculates the users' age by the provided DoB, so those under 14 must consult an adult to set up an account. Gender may be selected or opted out, and the country input field uses autocomplete for speed.

```
function autocomplete(inp, arr) {
    let currentFocus;

    inp.addEventListener("input", function(e) {                    // Execute Auto Complete When Someone Writes in the Text Field
        const value = this.value;                                  // Get Value
        closeAllLists();                                           // Close Any Already Open Lists of Autocompleted Values
        if (!value) return false;                                  // If No Value Provided Return False
        currentFocus = -1;                                         // Take Off Focus

        const div = document.createElement("DIV");                 // Create a DIV Element that Will Contain the Items (Values)
        div.setAttribute("id", this.id + "autocomplete-list");     // Add ID
        div.setAttribute("class", "autocomplete-items");           // Add Class Name

        this.parentNode.appendChild(div);                          // Append the DIV element as a Child of the Autocomplete Container
        for (i = 0; i < arr.length; i++) {                         // Travers Items
          if (arr[i].substr(0, value.length).toUpperCase() == value.toUpperCase()) { // Check If Item Starts with Value
            elem = document.createElement("DIV");                  // Create Item Element
            elem.innerHTML = "<strong>" + arr[i].substr(0, value.length) + "</strong>";  // Add Emphasis on Matching String
            elem.innerHTML += arr[i].substr(value.length);         // Rest of the Text
            elem.innerHTML += "<input type='hidden' value='" + arr[i] + "'>";        // Insert Input that Holds the Current Array Item's Value
                elem.addEventListener("click", function(e) {       // Execute a Callback on Click
                inp.value = this.getElementsByTagName("input")[0].value;             // Insert the Value for the Autocomplete
                closeAllLists();                                   // Close Any Opened Lists
            });
            div.appendChild(elem);                                 // Append List
          }
        }
    });
```

Figure-85 Autocomplete

5.3.4. HOME

When users register, an email with a ***confirmation link*** arrives at the provided address. After a successful login, the home page is accessible, with three main parts: profile, main menu and songs.

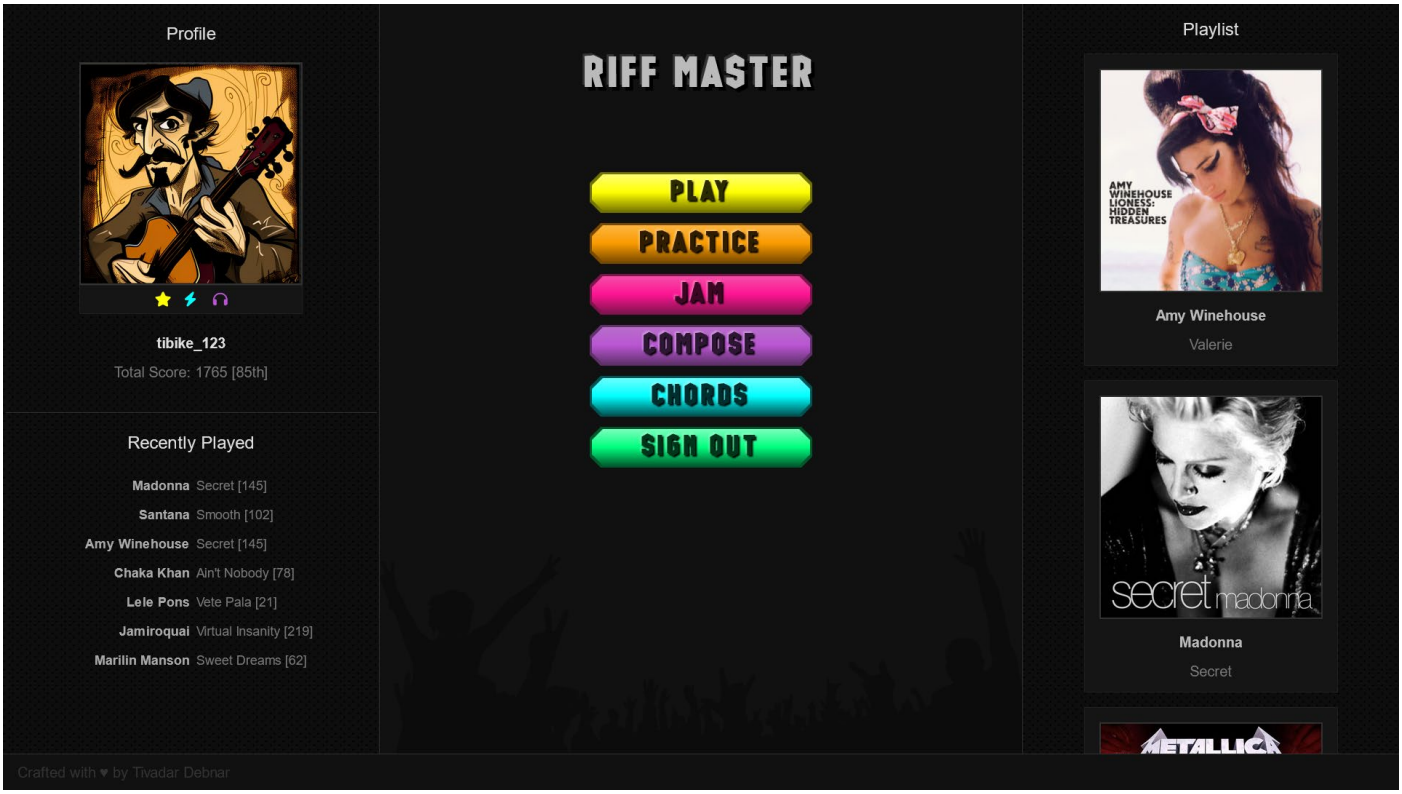


Figure-86 Home

Our home page loads additional data, such as the public tablatures. There are three ways to do asynchronous function calls with JavaScript: callbacks, promises with then-catch clauses, and async-await. ***Async-await*** is used for consistency, as it can be read just like synchronous programming and, by now, is a fully compatible programming technique.

```
async function getPublicTabs() {                                    // Get Tabs with isPublic True from DB
    try {
        const url = "http://localhost:5000/api/tabs";               // API Endpoint
        const option = {
            "method": "GET",                                        // Get Tabs
            "Content-Type": "application/json",
            "Accept": "application/json",
            "headers": { "x-auth-token": token }                    // Send JSON Signature Token
        }
        const result = await fetch(url, option);                    // Consume Request
        const json = await result.json();                           // Transform to JSON
        return json.tabs.filter(tab ⇒ tab.isPublic);                // Get Public Tabs
    }
    catch (err) { console.log(err); }                               // Temp: Just Console (There Will be a Message Box)
}
```

Figure-87 Async-Await

5.3.5. MANIPULATING AUDIO

Every page that reads the controller's state uses controller event listeners, providing function calls that may be different for each page. The Jam Session is all about making the instrument play audio when the user interacts with it and displays these interactions on a neck board. The app registers/deregisters arrays of notes for each string, and the ***highest finger positions*** are played for strummed strings.

```
function handleStrumActivated(event, strum) {
    const positions = app.controllerState.highestFretPositions[strum - 1];  // Finger Positions on a String Row
    const upperMost =  positions[positions.length - 1];                     // Topmost Pressed Position

    const note = guitarNotes[strumOffsets[strum - 1] + upperMost];          // Note Name that Plays

    if (event ≠≠ 0) {                                                       // Strum Pressed
        const notesOnString = positions                                     // Find Strings Currently Playing
            .map(position ⇒ guitarNotes[strumOffsets[strum - 1] + position]);
        const notesPlaying = notesOnString                                  // Find Notes Currently Playing
            .filter(note ⇒ app.audio[note].playing());
        notesPlaying.forEach(note ⇒ stopNote(note));

        displayActionOnBoard(upperMost, strum, strum, !event);              // Highlight Played Note and String
    }
    else {                                                                  // Strum Released
        if (upperMost ≡≡ undefined) return;
        playNote(note, strum);
        if (upperMost ≠≠ 0) {                                               // If Finger Position is Not 0
            displayActionOnBoard(upperMost, strum, -1, false);              // Put Back Semi Highlight
        }
        else {
            displayActionOnBoard(upperMost, strum, strum, !event);          // Clear Highlight
        }
    }
}
```

Figure-89 Callback Handling

The display and equaliser are dynamically created when the page loads. Because they consist of several hundred DOM elements, they are stored in the app ***cache*** for fast lookup.

```
function getDOMElements() {
    app.DOM.board_LI = [ ... $all("#guitar li.fret-note")];                  // List Element (Note)
    app.DOM.board_SPAN = [ ... $all("#guitar span.fret-note")];              // Span Element (Note's Text)
    app.DOM.board_BTN = [ ... $all("#guitar button")];                       // String Buttons
    app.DOM.boardStrings = [ ... $all(".guitar__string")];                   // Strings
    app.DOM.equalizerBeats_DIV = [ ... $all(".equalizer__beat")];            // Small Square on the Equaliser
}
```

Figure-90 Caching

Strum actions activate interval timers on the equaliser component to animate the ***stereo-style interface***. Each note beat column has a counter associated, set to 2000ms to reach the top and 500ms to drop.

These individual interval cycles may be safely interrupted with new strum actions, resetting timers and clearing previous interval functions from memory.

```
    if (start) {                                                    // Strum Actions Triggers Beat Animation
        let counter = 0;                                            // Reset Counter
        app.equalizerTimers[noteIndex] = setInterval(function() {   // Start Interval
            app.equalizer[noteIndex] = index + counter;             // Store the Equaliser on App State
            drawColumn(index, index + counter, string);             // Draw Elements

            if (counter ⩾ 32) {                                     // If Counter Over Flows
                clearInterval(app.equalizerTimers[noteIndex]);      // Delete Interval
                app.equalizerTimers[noteIndex] = null;              // Clear Storage As Well
                return displayActionOnEqualizer(note, false, string);  // Recurse to Down Cycle
            }
            counter++;                                              // Increment Counter
        }, 2000 / BEATS);                                           // 32 Beats in 2 Second
    }
    else {                                                          // If Reverse
        let counter = app.equalizer[noteIndex];                     // Get NoteIndex
        app.equalizerTimers[noteIndex] = setInterval(function() {   // Create Timer
            app.equalizer[noteIndex] = index + counter;             // Reset Equaliser Counter
            drawColumn(index, counter, string);                     // Draw Element

            if (counter < 0) {                                      // If Reached 0
                clearInterval(app.equalizerTimers[noteIndex]);      // Clear Timer
                app.equalizerTimers[noteIndex] = null;              // Reset App State
                return;
            }
            counter--;                                              // Decrement Here
        }, 1);                                                      // 32 Beats Each 1ms
    }
}
```

Figure-91 Equaliser

Each ***development iteration*** appends the layout strengthening the educational aspect of Jam Sessions, such as the guitar neck or equaliser, which are improved to show ***annotations***, ***colour-coded strings*** for consistency and react visually to strum events by ***highlights***.

Figure-92 Jam

5.3.6. CHORDS

We must apply an efficient lookup to filter finger position arrays efficiently. Our chord list (+120000 items) is eventually stored as an object with position names as keys and variation arrays as properties. This type of ***object structuring*** means we may look up the items by key, which is faster and more memory efficient than array iterations.

```
if (all) {
    // Create a Complete Chord List with ALL the Items Under ONE Specific Name
    if (chordNames.length > 1) throw Error("Display set to DISPLAY ALL! Names parameter expect an array of string of 1");

    // Display Chords
    const chordList = chords[chordNames[0]] || [];                  // Extract List with Key Name
    for (let i = 0; i < chordList.length; i++)                      // Iterate List
        displayChord(chordList[i], parent, chordNames[0]);          // Display ALL Chords

    // Set Chord List Filter and Chord Info
    const filterInfoDOM = $("#chord_search__filter-info");          // Get Info Element
    const chordsLength = Object.keys(chords).length;                // Get List Length
    filterInfoText = `${ chordList.length } / ${ chordsLength }`;   // Create Info Text
    filterInfoDOM.innerHTML = filterInfoText;                       // Add Text
    const chordInfoDOM = $("#chord_search__chord-info");            // Get Info Element
    chordInfoDOM.innerHTML = getChordName();
}
```

Figure-93 Filter Chords

Now that we have our finger position variations array, we may ***construct cards dynamically***. As these components may be placed in several different fret heights, we must also calculate our starting positions:

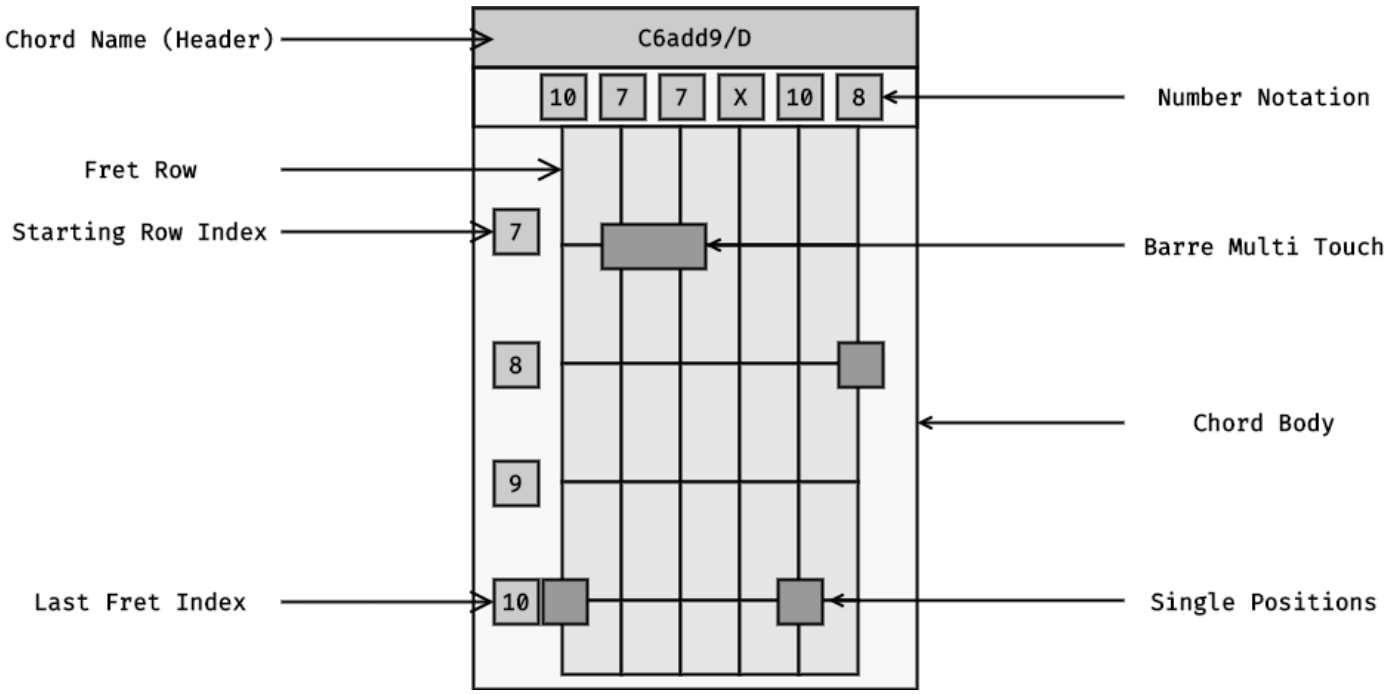


Figure-94 Chord-Card Design

Barre chords have multiple locations on the neck and require special attention. We may identify and display multiple barre touches by calculating consecutive fret position occurrences within chords.

```
// Find Barre Fret Positions (When One Finger Touches Across Multiple Strings)
function getBarres(pos, fing) {
    const multiTouches = []                                          // Create Result Array
    const noFinger0 = fing.filter(f ⇒ f ≠= "0");                     // 0 Finger Cannot be Barre
    const isMultitouch = noFinger0.length ≠= new Set(noFinger0).size; // Check for Size Difference
    if (!isMultitouch) return multiTouches;                          // Return Empty If No Difference

    for (let i = 0; i < 6; i++) {                                    // Iterate Finger Positions
        if (fing[i] ≠= "0" && !multiTouches.find(t ⇒ t.finger == fing[i])) { // If F Position is NOT 0
            const start = i;                                         // Start Counting from I
            let end = i;                                             // Set End Initially I

            for (let j = i + 1; j < 6; j++) {                        // Look Ahead
                if (fing[i] == fing[j]) end = j;                     // If Same Finger Found Set End
            }
            if (start ≠= end) multiTouches.push({
                finger: fing[i],
                position: pos[i],
                start, end
            });
        }
    }
    return multiTouches;
}
const barres = getBarres(pos, fin);
```

Figure-95 Barres (chords.js)

The chord page has two main parts: a card list and a signature section with note, base and type filters. Users may search for chords and finger positions and play chord cards.

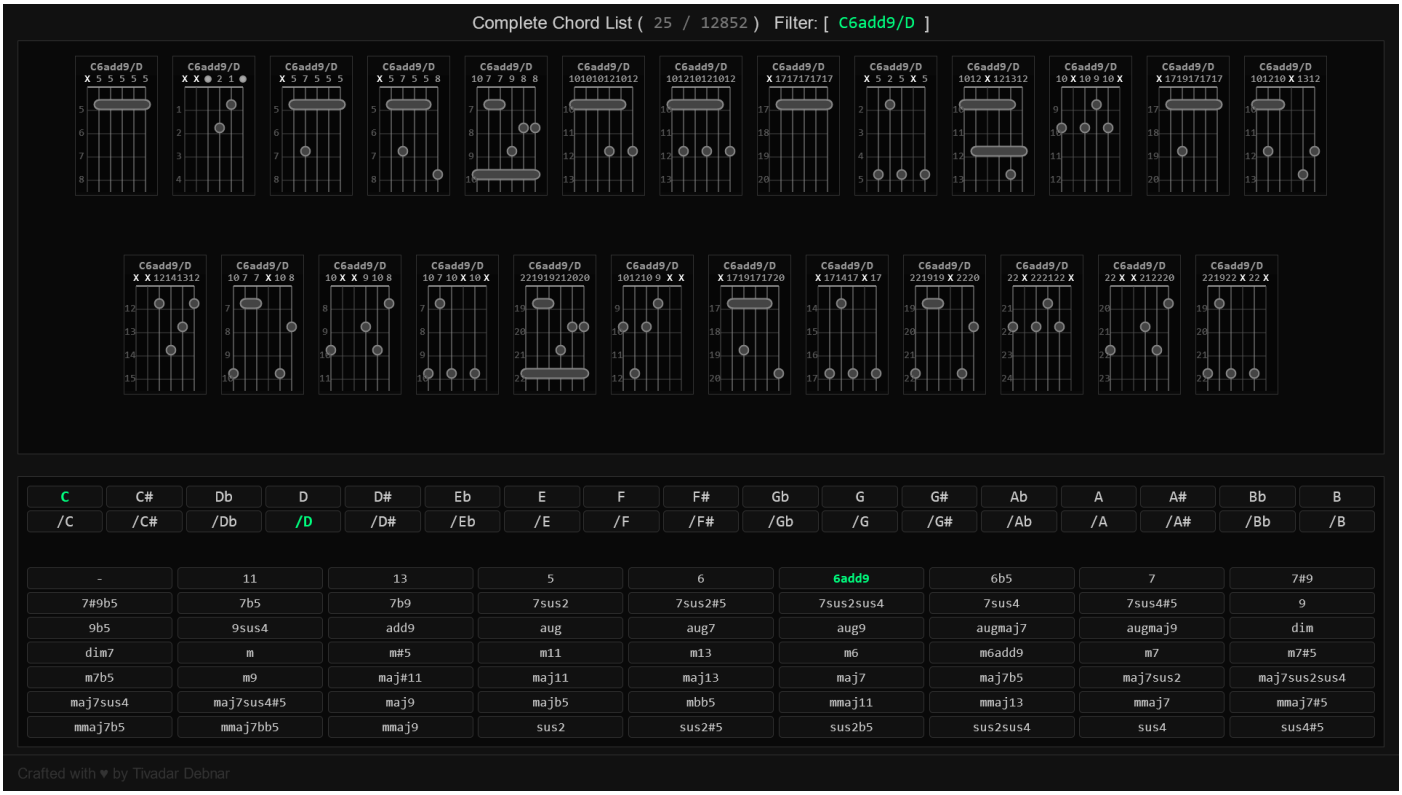


Figure-96 Chords

5.3.7. MUSIC STUDIO

The tablature component is the most time-consuming software module to develop, and commercial solutions would require additional months of work. However, for the prototype, we covered every fundamental feature of our specification, for instance:

- displaying single-line and multi-line tab windows,
- parsing tablature notation into staffs (rows) and bars (columns),
- using note duration icons,
- editable tablature sheets,
- instrument recording.

Figure-97 Music Studio

When the player presses the start button, the applications game loop starts. On every beat, the positions of the notes are recalculated to fit their relative bar index and trigger audio play/stop actions on the current note index.

```
function gameLoop() {                                       // Run on Every 32nd Beat
    if (app.play) {                                         // If User Pressed Play Button
        updateTabIndexInfo();
        if (app.noteIndex === 0) {
            centerCurrentBarInTab();                        // On First Beat Redraw for Centering Active Bar and Notes
            highlightCurrentBar();                          // Highlight Current Bar

            const backButtons = $all("#fast-backward, #backward"); // Disable Buttons
            backButtons.forEach(b => b.disabled = true);
            const foreButtons = $all("#fast-forward, #forward");   // Disable Buttons
            foreButtons.forEach(b => b.disabled = true);
        }
        highlightCurrentNotes();                            // Highlight

        // Play Tab Notes
        const bar = app.tab.bars[app.barIndex];
        bar.forEach((note, noteIndex) => {                   // Traverse  Bars
            const [ noteStr, start, duration] = note.split(":");    // Get Beats

            if (start == app.noteIndex) {                    // If Note Should Start
                const indNotesElem = $("#current-note");     // Get The Current Note Element in Header
                indNotesElem.innerHTML = noteIndex + 1;      // Display the Note Index in Header

                noteStr.split(",").forEach(n => {            // Split Note String
                    const letter = n.match(/[a-z]/gi)[0];    // Get Letter
                    const number = n.match(/[0-9]+/gi)[0];   // Get Number
                    const offSet = stringJumps[letter];      // Get Offset for Jumps
                    const audioIndex = Number(number) + Number(offSet);  // Get Audion Note
                    const audioName = guitarNotes[audioIndex];       // Get Audio Name
                    playNote(audioName);                     // Play the Note

                    const durationMS = getNoteDurationInMS(duration);   // Calculate the Duration of the Note
                    const noteTimer = setTimeout(() => {     // Create a Timer Function to Stpo the Note
                        stopNote(audioName);                 // Stop
                        clearTimeout(noteTimer);             // Delete Timer
                    }, durationMS);
                });
            }
        });
```

Figure-98 Game Loop

When editing is enabled, users can modify the tablature. However, as the tablature is a deeply nested DOM component, thousands of elements may require event listeners. Therefore, we used ***event delegation***, an old and tried web development technique. Instead of binding listeners for every component, we use the parent component and check if the event happened on a particular child element (target).

```
// Edit Tablature Note
$("#tab-sheet").addEventListener("click", editBeat);              // Use Event Delegation
function editBeat(event) {                                        // Edit Function
    const body = $("body");                                       // Get Body
    const elem = event.target;                                    // Get Target
    if (!app.tabEditingEnabled) return;                           // Do Nothing If Tab Editing is Disabled
    if (!elem.classList.contains("beat")) return;                 // Do Nothing If Click Not Happened on a Beat

    const prevEditForms = $("#beat-edit-bg");                     // Get Previously Opened Editors
    if (prevEditForms) body.removeChild(prevEditForms);           // Remove Them

    const beatInfo = getBeatInfoForEditing(event);
    app.editFormInfo = {};

    // Align Edit Form Around the Clicked Element
    const { width, height } = elem.getBoundingClientRect();        // Form Height and Width
    const windowWidth = window.innerWidth || body.clientWidth;     // Window Dimenstions
    const windowHeight = window.innerHeight || body.clientHeight;
    const { x, y } = beatInfo.rect;                                // Bounding Client Rectangle
    let placeLeft = x >= windowWidth / 2;                          // Mid Points
    let placeTop = y >= windowHeight / 2;
```

Figure-99 Event Delegation

By clicking on beats, note edit forms appear, where users can set chord names, string fret values, or select note durations. Additionally, music notes can be moved around or deleted, and bars can be inserted/removed.

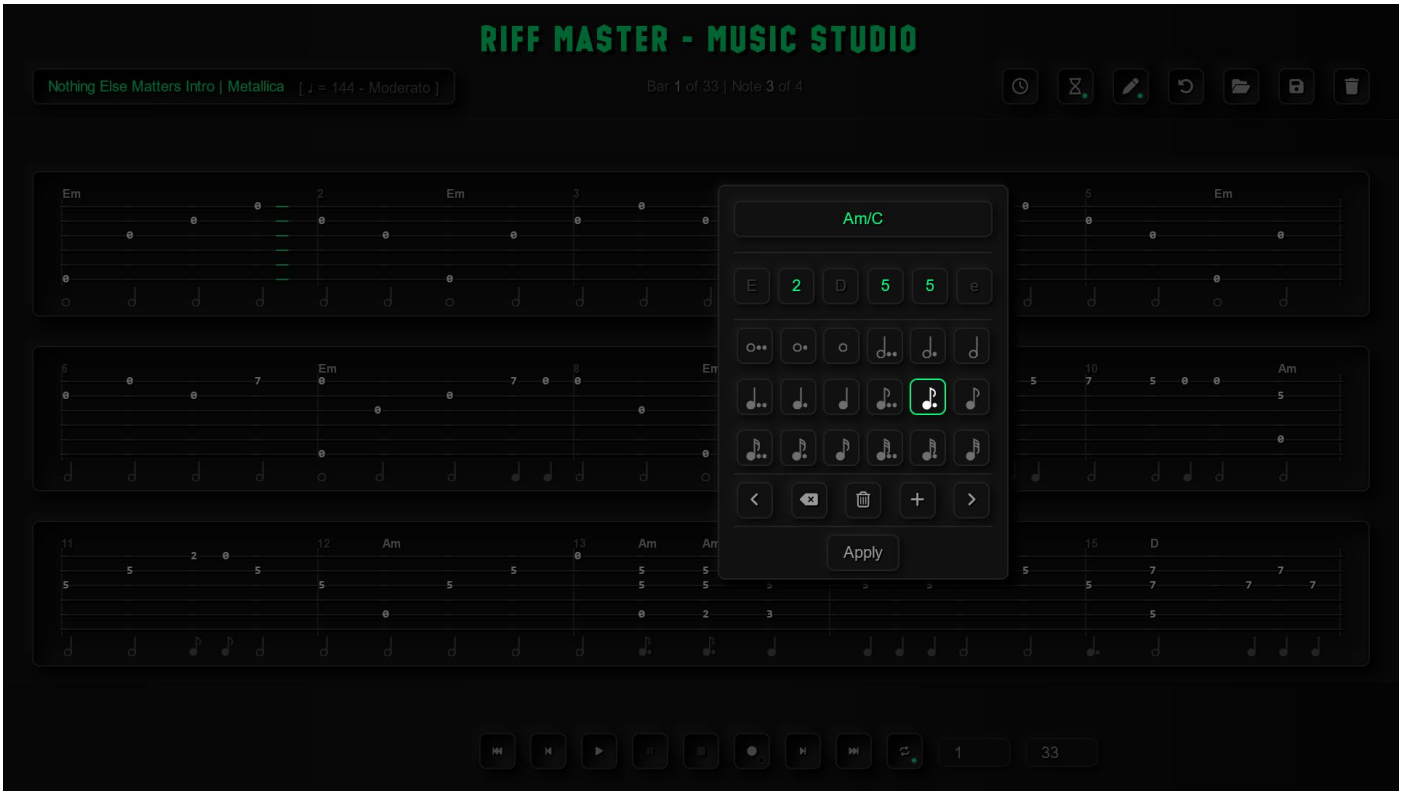


Figure-100 Editing

By clicking the ***undo*** button, the last actions are revoked. We can undo multiple steps because app history is stored as an array of stringified states and ***pops like traditional Stacks***.

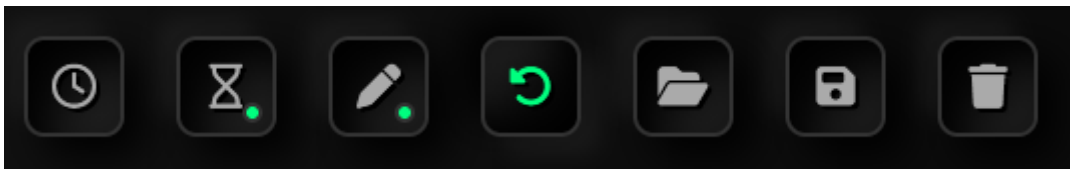


Figure-101 Undo

The app listens to controller inputs and reads individual guitar states while the record button is active. However, there are some restrictions because of the edge cases:

1. Thirty-second note lengths were set as the ***smallest unit of measurement*** in our tab structure and translation. Activated notes hence belong to the app state beat index, and multiple notes on an index would automatically be concatenated into beats.

```
    // Condense Notes with the Same Start Position
    if (existNoteIndex) {                                          // If Note Exists on the String
        for (let i = 0; i < bar.length; i++) {                     // Iterate Bar
            const [ n, s, d, c ] = bar[i].split(":");              // Extract Note, String, Duration and Chord Name
            if (Number(s) === noteIndex) {                         // If String Found on Note Index
                const strings = n.split(",").map(s => s.replace(/\d+/g, "")); // Dissect Note to Separate Strings to Find If Duplicate

                if (strings.includes(string)) {
                    finaliseNote(strum);
                    break;
                }

                // New Condensed Note
                const condensed = `${ n },${ string }${ upperMostFret }:${ s }:${ d }:${ c }`;
                app.tab.bars[barIndex][i] = condensed;
            }
        }
```

Figure-102 Concatenation (compose.js)

2. If the user presses the note longer than the ***maximum translatable duration*** ($1\frac{3}{4}$), we handle unfinalised notes by triggering timer callbacks.

```
        const removeTimeout = setTimeout(() => removeWithLongestDuration(unfinalised, removeTimeout), removeTime);
    }

    function removeWithLongestDuration(unfinalisedNoteStr, timeout) {
        const [ _, start, __, barIndex ] = unfinalisedNoteStr.split(":");
        const index = app.unfinalisedNotes.findIndex(n => n === unfinalisedNoteStr);
        if (index !== -1) {
            app.unfinalisedNotes.splice(index, 1);                 // Remove from Unfinalised Note Store
            const bar = app.tab.bars[barIndex];

            for (let i = 0; i < bar.length; i++) {
                const curr_n = bar[i];
                const [ n, s, _, __ ] = curr_n.split(":");          // Notes, Start, Duration, Chord Name
                const finalised = `${ n }:${ s }:W..`;             // The Longest Possible Note

                // Finalise Note
                if (s === start) app.tab.bars[barIndex][i] = finalised;
                centerCurrentBarInTab(app.tab.bars[barIndex]);
            }
        }
        clearTimeout(timeout);
    }
    centerCurrentBarInTab();
}
```

Figure-103 Note Collector (compose.js)

Loops can be activated with start and end bar indices. However, loop activation only works if the record is off, as the record function pushes the bar list indefinitely.

Figure-104 Loops

Users save their work in simplified tab formats, and title, difficulty and other parameters may be modified. The default privacy setting makes the content ***inaccessible to other users***, and access can be set public when tablatures are ready for publishing.

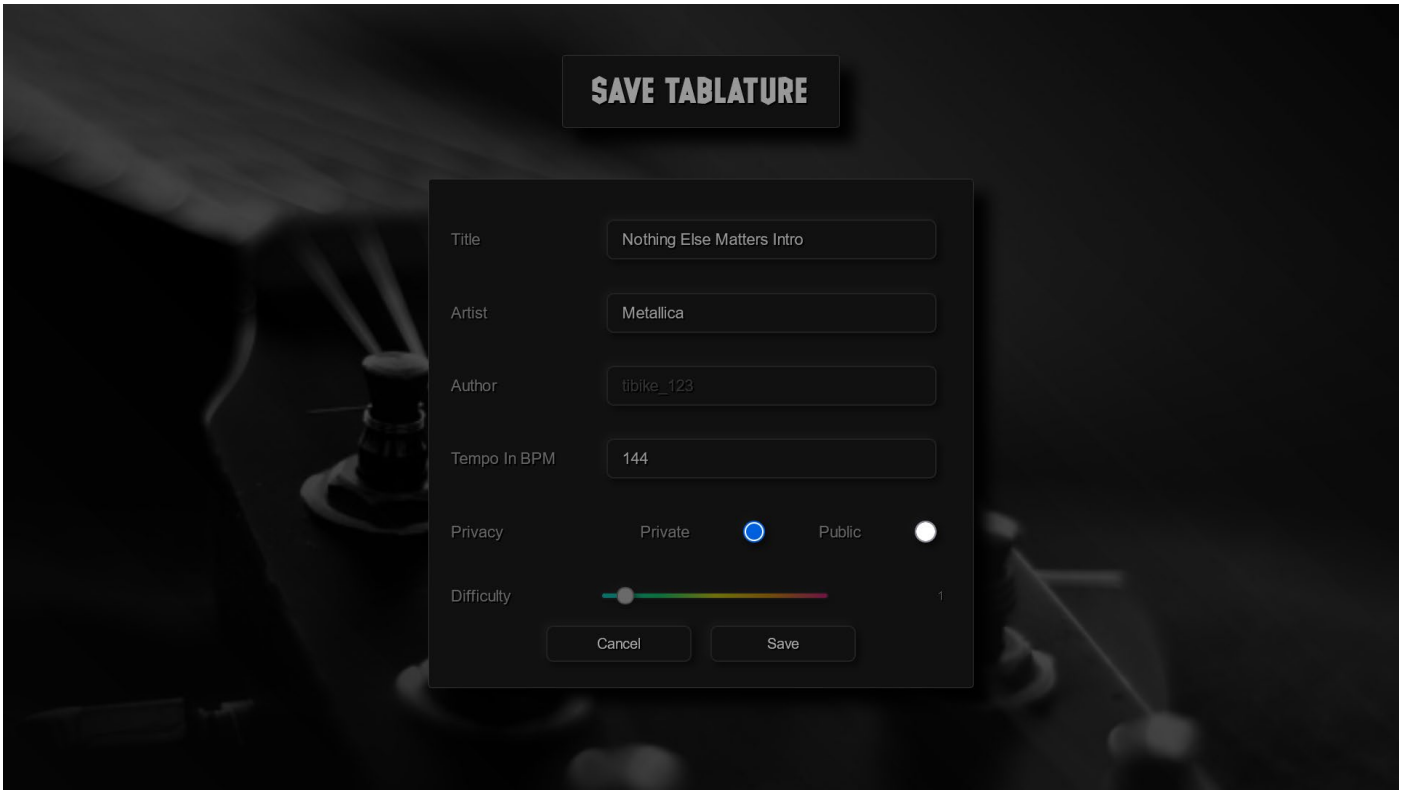


Figure-105 Saving

It is worth noting that there is no restriction on how many versions can be composed and saved under a single artist title combination.

We can also continue tablature composing by loading them into our tab sheet editor. When an editor loads, the complete tab history is emptied, and the application must be reset to its ***default initial state***. For simplicity, features such as saving previous work warnings are omitted.

Figure-106 Loading

5.3.8. GAME

The game is built by iteratively fine-tuning representations of tablature notations with the help of utility functions:

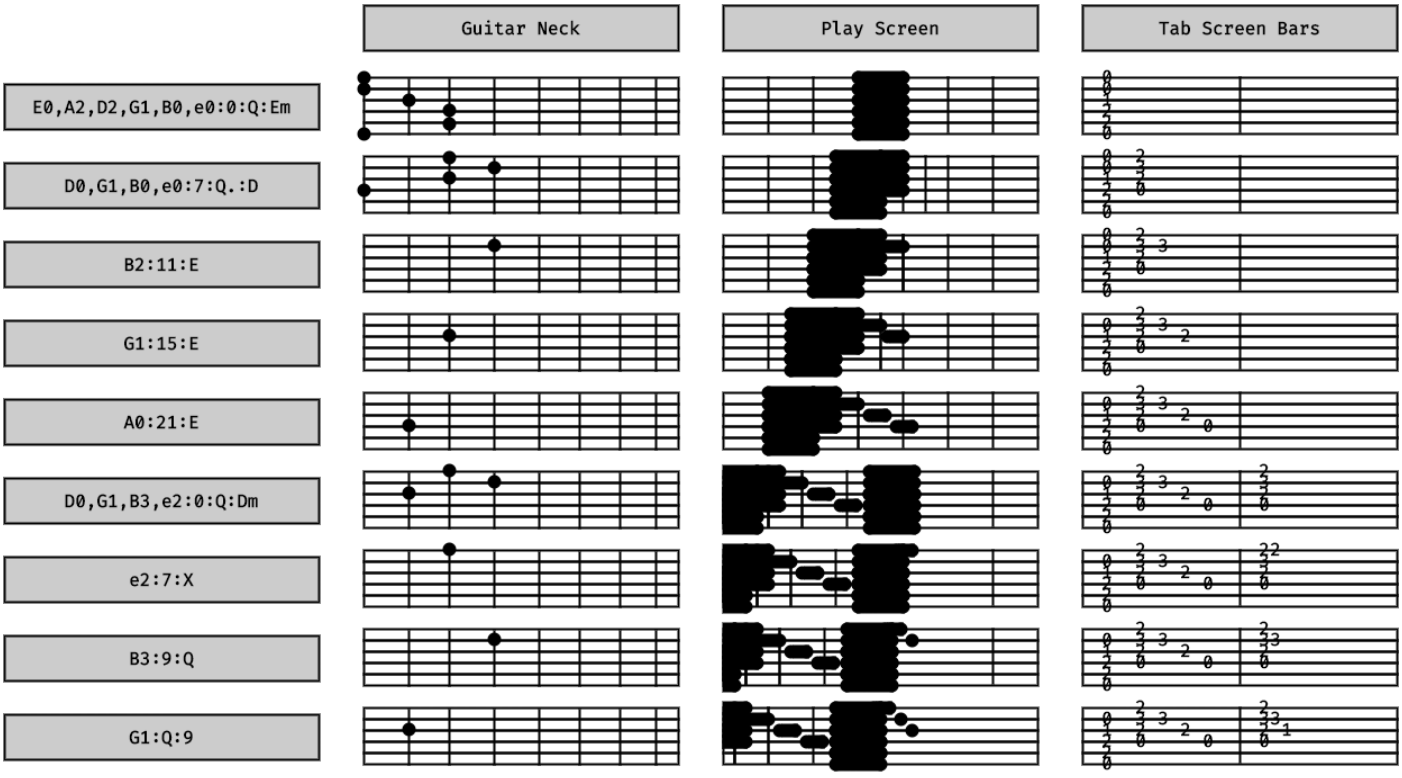


Figure-107 Translations and Data Representations

We built refined types of displays solely relying on our simplified tab structure. Games that emulate guitars must have a graphical representation of the music being played. This user interface component is often called ***note highway*** or track in the gaming community. We used skeleton scaffolding for tracks and modified style sheets on each incrementation by manipulating CSS class names.

```
    // Display Notes Based on String Map
    const colors = ["yellow", "orange", "red", "purple", "blue", "green"];  // Colour Scheme
    app.noteTrackMap = noteTrackMap;                                        // Save or Update Map for Reference in Game Loop
    for (let string_i = 0; string_i < STRING_NUM; string_i++) {             // Iterate Strings
        for (let beat_i = 0; beat_i < totalBeats; beat_i++) {               // Iterate Notes
            const id = `#note-track__beat-${ beat_i }-string-${ string_i }`; // Create the Index String for the DOM Element
            const noteElem = $(id);                                         // Get the DOM Element with ID

            const mapPosition = noteTrackMap[string_i][beat_i];             // Get Element with Position
            noteElem.classList.remove( ... noteElem.classList);             // Remove All Classes
            noteElem.classList.add("note-track__string");                   // Add Back Original Class Name
            noteElem.innerHTML = "";                                        // Delete Text Content
            if (mapPosition === undefined) continue;                        // Next Iteration If No Item on Map Position

            const isStartOfNote = mapPosition[0] === "<";                   // Decide If Beat is The Beggining of a Note Press
            const isEndOfNote = mapPosition[mapPosition.length - 1] === ">"; // Decide If Beat is The End of a Note Press
            const fretNumber = mapPosition.match(/\d+/g)[0];                // Get Fret Number from Map
            if (isStartOfNote) {                                            // Start Notes Look Different
                noteElem.innerHTML = fretNumber;                            // Add Fret Number for Starter Notes
                noteElem.classList.add("start");                            // Add Special Class to Them
            }
            if (isEndOfNote) noteElem.classList.add("end");                 // End Notes Look Different As Well
            if (beat_i < PREV_BEATS) noteElem.classList.add("prev");// Dim Played Notes
            if (beat_i === PREV_BEATS) noteElem.classList.add("current"); // Highlight Actual Note
            noteElem.classList.add("active");                               // Add Activated to Show Note on Track
            noteElem.classList.add(colors[string_i]);                       // Add Colour Class
        }
    }
}
```

Figure-108 Displaying Notes (play.js)

The track is divided into past, current and future sections and received specialised stylings.

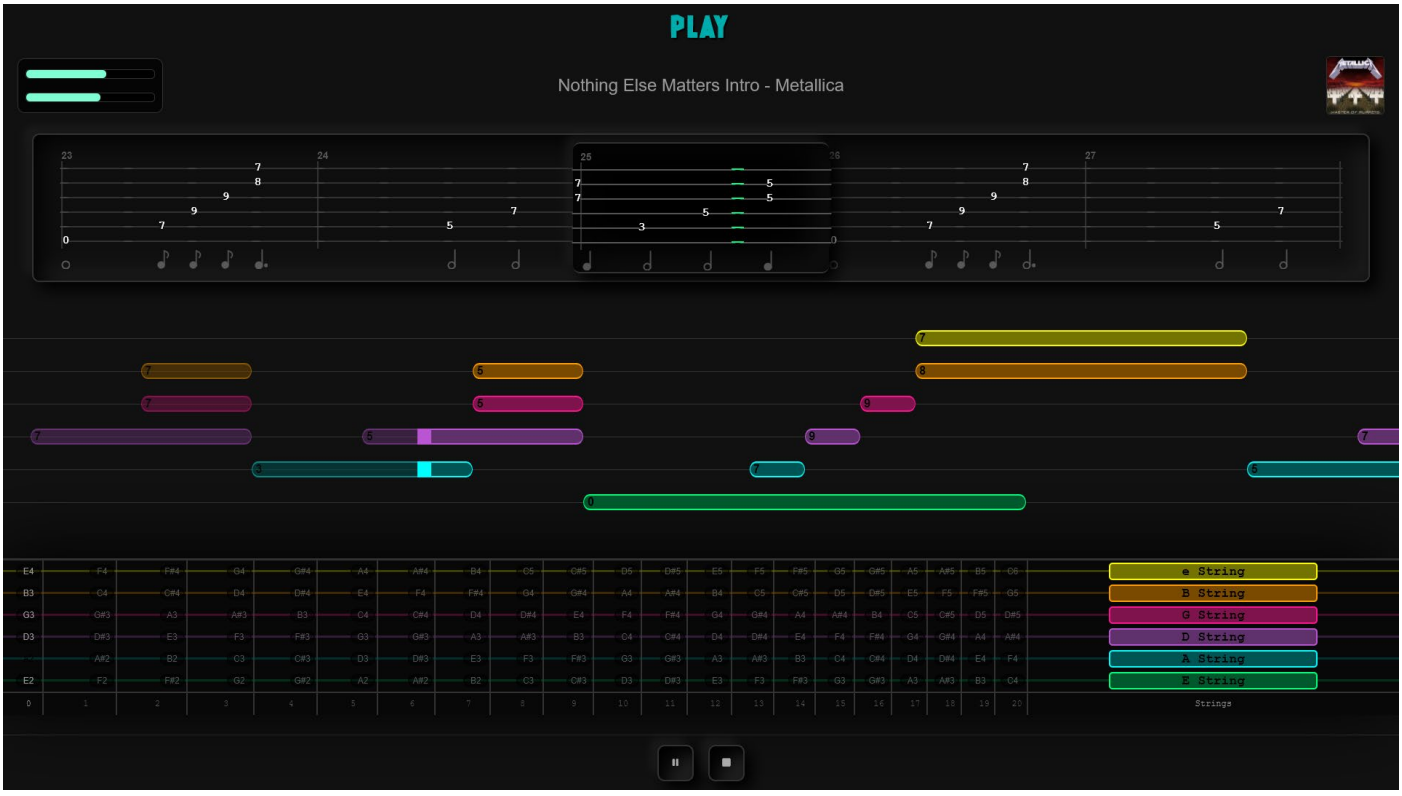


Figure-109 Gaming

The tab and track components operate a ***single game loop*** to spare computation. When the loop begins, the music audio starts playing, and with each incrementation, the tab and the track are redrawn.

```
    // Display Notes Based on String Map
    const colors = ["yellow", "orange", "red", "purple", "blue", "green"];  // Colour Scheme
    app.noteTrackMap = noteTrackMap;                                        // Save or Update Map for Reference in Game Loop
    for (let string_i = 0; string_i < STRING_NUM; string_i++) {             // Iterate Strings
        for (let beat_i = 0; beat_i < totalBeats; beat_i++) {               // Iterate Notes
            const id = `#note-track__beat-${ beat_i }-string-${ string_i }`; // Create the Index String for the DOM Element
            const noteElem = $(id);                                         // Get the DOM Element with ID

            const mapPosition = noteTrackMap[string_i][beat_i];             // Get Element with Position
            noteElem.classList.remove( ... noteElem.classList);             // Remove All Classes
            noteElem.classList.add("note-track__string");                   // Add Back Original Class Name
            noteElem.innerHTML = "";                                        // Delete Text Content
            if (mapPosition === undefined) continue;                        // Next Iteration If No Item on Map Position

            const isStartOfNote = mapPosition[0] === "<";                   // Decide If Beat is The Beggining of a Note Press
            const isEndOfNote = mapPosition[mapPosition.length - 1] === ">"; // Decide If Beat is The End of a Note Press
            const fretNumber = mapPosition.match(/\d+/g)[0];                // Get Fret Number from Map
            if (isStartOfNote) {                                            // Start Notes Look Different
                noteElem.innerHTML = fretNumber;                            // Add Fret Number for Starter Notes
                noteElem.classList.add("start");                            // Add Special Class to Them
            }
            if (isEndOfNote) noteElem.classList.add("end");                 // End Notes Look Different As Well
            if (beat_i < PREV_BEATS) noteElem.classList.add("prev");// Dim Played Notes
            if (beat_i === PREV_BEATS) noteElem.classList.add("current"); // Highlight Actual Note
            noteElem.classList.add("active");                               // Add Activated to Show Note on Track
            noteElem.classList.add(colors[string_i]);                       // Add Colour Class
        }
    }
}
```

Figure-110 Drawing Track (play.js)

The users' performance is measured in accuracy and time precision, and the total score will be their average. To calculate accuracy, we must compare the expected and actual guitar state on each iteration of 32nd -s.

As we have six strings, we may get average accuracy for beats and add them to counters. For performance, we apply ***timed callbacks*** when interacting with the controller. The callbacks will look behind or ahead ten beats, and if the activated note is present on that string, we adjust the performance score.

```
    // GET PRECISION
    if (app.play && event === 0) {
        let precisionScore = 0;
        let upcomingPrecisionScore = 0;
        let pastPrecisionScore = 0
        const string = 6 - strum;                                                       // Which String is Playing
        const currentIndex = app.noteTrack.dimensions.playedNotes;                      // Get Currently Played Note
        const upcomingMapSection = app.noteTrackMap[string].slice(currentIndex, currentIndex + 10);     // Get Upcoming 10 Notes on the String
        const upcomingSectionIndex = upcomingMapSection.findIndex(note => note === "<" + upperMost);    // Find if any of the notes has a starting with uppermost
        if (upcomingSectionIndex !== -1) {                                              // If Found
            upcomingPrecisionScore = [100, 90, 80, 70, 60, 50, 40, 30, 20, 10][upcomingSectionIndex];   // Score Closest
        }

        const pastMapSection = app.noteTrackMap[string].slice(currentIndex - 9, currentIndex + 1);      // Get Past 10 Notes on String
        const pastSectionIndex = pastMapSection.findIndex(note => note === "<" + upperMost);            // Find if any of the notes has a starting with uppermost
        if (pastSectionIndex !== -1) {                                                  // If Found
            pastPrecisionScore = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100][pastSectionIndex];        // Score Closest
        }
        precisionScore = Math.max(upcomingPrecisionScore, pastPrecisionScore);                          // Get Closest Precision (Multiple Occurance)
        app.precisionForStrums.push(precisionScore);
        const avgPrecision = app.precisionForStrums.reduce((a, b) => a + b) / app.precisionForStrums.length;
        const precisionBarElem = $("#precision");
        precisionBarElem.style.width = avgPrecision + "%";
    }
}
```

Figure-111 Precision (Appendix/Code/play.js)

6. TESTING

With a complex system that integrates digital guitar controllers and a web application, thorough testing becomes paramount. We must test our hardware and software components separately and conjointly to ensure prototype quality. As the controller is the foundation of this project, we start with the guitar device.

6.1. HARDWARE

We have created separate functions for printing the app state intuitively and used them while developing the console. Each string and fret displays its current state (0-pressed, 1-inactive) and is printed in a console table on the serial monitor.

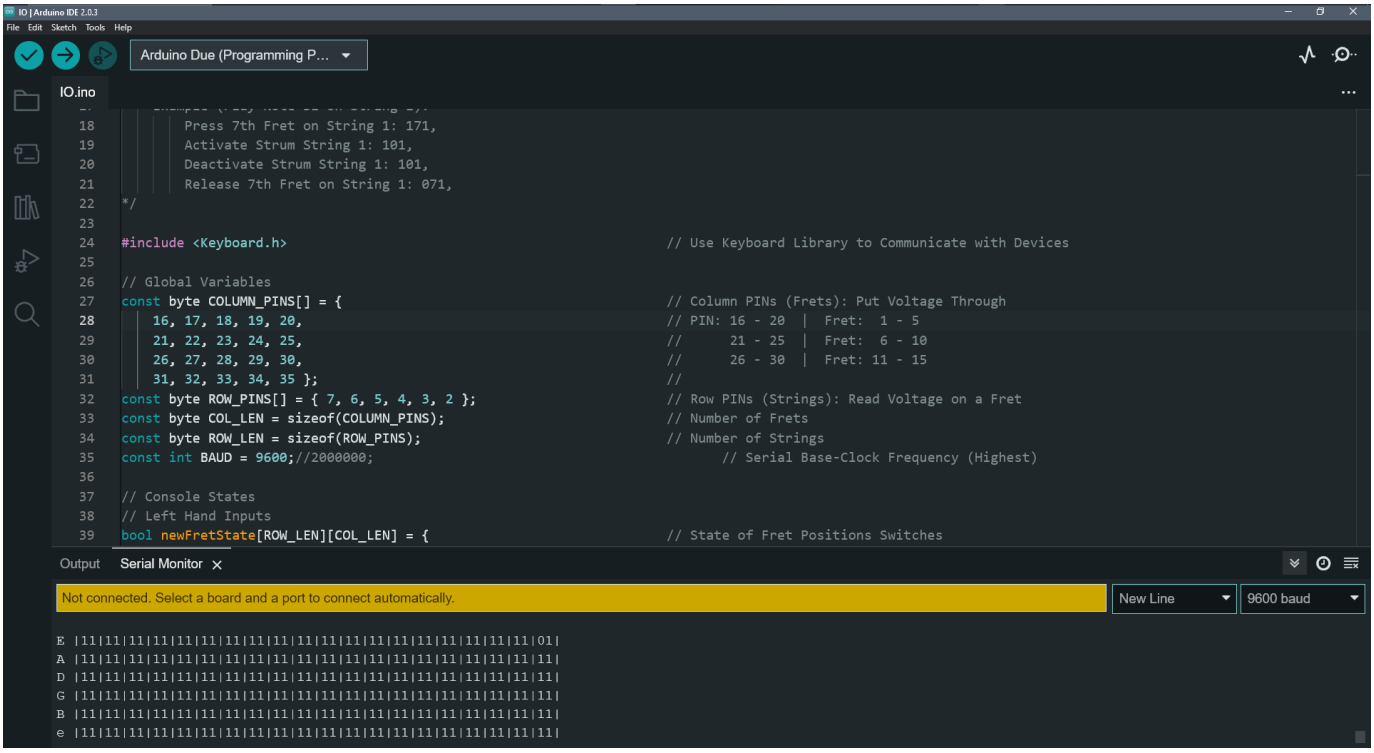


Figure-112 Serial Monitor

After troubleshooting some misbehaving buttons, the readings were correct; this phase passed all tests.

6.1.1. INTERACTION TESTING

However, while reading the device state is an intuitive way of testing, it completely ignores that the ultimate goal of our testing is to ***ensure that interactions send the corresponding messages***. Therefore, we can open an empty notepad and manually check the guitar while comparing actions with the expected messages. We must test individual fret and string actions and an extensive combination of multiple fret presses and chords.

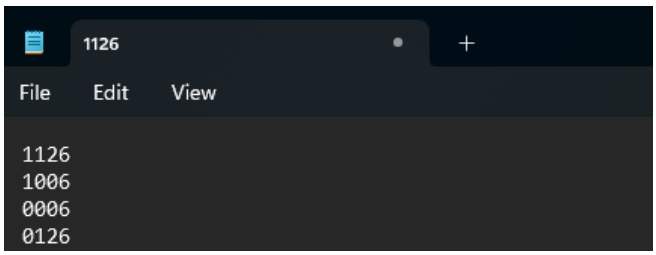


Figure-113 Testing on Notepad

	A	B	C
1	**Type**	**Result**	**Comment**
2	No Fret Only String	Pass	
3	Fret No String	Pass	
4	Fret X	Pass	
5	2 Adjacent Frets	Pass	In places only as barre (uncomfortable)
6	3 Adjecent Frets	Pass	Same as 2 adjecents
7	Triangle 1	Pass	
8	Triangle 2	Pass	
9	Triangle 3	Pass	
10	Triangle 4	Pass	
11	Square	Pass	
12	Major Chords	Pass	Some Chords are difficult to press
13	Minor Chords	Pass	Some Chords are difficult to press
14	Multiple Strings 2 - 6	Fail	In cases one string skips

Figure-114 Hardware Tests

Technically all test cases are passing. However, some minor issues came to light regarding our console device. First, some combinations are challenging to play on this handcrafted instrument due to *minor inaccuracies in manufacturing* execution. Secondly, strings must have high-tension settings, requiring more force than a player would apply on a guitar naturally.

6.2. END-TO-END TESTING

Most frontend features rely on API requests to backend servers. These endpoints were built separately from the UI and were tested using Postman. The following criteria were used to ensure that the API works correctly:

3. API route must be available,
4. Route ids must have the correct length and format (castable),
5. Request headers may require JWT signatures,
6. Route body must be tested for:
    - Correct property names,
    - Value constrictions.

Each route passes our tests if the responses are an appropriate JSON object with the requested data, response message or an error message. The following test cases were collected:

	A	B	C	D	E	F	G	H	I
1	API Endpoint	Test	Result	API Endpoint	Test	Result	API Endpoint	Test	Result
2	/api/users POST	Request Body Required	Pass	/api/login POST	No Request Body	Pass	/api/tabs GET ID	Return Tab Object	Pass
3	/api/users POST	There is No Request Body	Pass	/api/login POST	Unauthorized	Pass	/api/tabs GET USER_ID	ID is Castable to ObjectID	Pass
4	/api/users POST	Body Properties are Valid	Pass	/api/login POST	Return User Object	Pass	/api/tabs GET USER_ID	Could not Find ID	Pass
5	/api/users POST	User with Email Already Exists	Pass	/api/login POST	Exclude Password	Pass	/api/tabs GET USER_ID	Return Tab List with User	Pass
6	/api/users POST	User with Username Already Exists	Pass	/api/profiles GET ID	ID is Castable to ObjectID	Pass	/api/tabs GET USER_ID	Return Empty List	Pass
7	/api/users POST	Return User without Hashed Pswd	Pass	/api/profiles GET ID	Could not Find ID	Pass	/api/tabs GET USER_ID	Exclude Tab Content	Pass
8	/api/users GET	Return JSON with Array of Users	Pass	/api/profiles GET ID	Return User Profile if Found	Pass	/api/tabs POST	No Request Body	Pass
9	/api/users GET	Return Empty Array if No User	Pass	/api/profiles POST	Missing Request Body	Pass	/api/tabs POST	ID is Castable to ObjectID	Pass
10	/api/users GET ID	ID is Castable to ObjectID	Pass	/api/profiles POST	Invalid Request Body	Pass	/api/tabs POST	Could not Find User with ID	Pass
11	/api/users GET ID	User Not Found with ID	Pass	/api/profiles POST	ID is Castable to ObjectID	Pass	/api/tabs POST	Title Already Exists	Pass
12	/api/users GET ID	User Returned if Found	Pass	/api/profiles POST	Could not Find ID	Pass	/api/tabs POST	Return Tab Object	Pass
13	/api/users GET ID	No Password is Retured	Pass	/api/profiles POST	Profile Already Exists with User ID	Pass	/api/tabs PUT	No Request Body	Pass
14	/api/users PUT	ID is Castable to ObjectID	Pass	/api/profiles POST	Return User Profile if Found	Pass	/api/tabs PUT	ID is Castable to ObjectID	Pass
15	/api/users PUT	User Not Found with ID	Pass	/api/profiles PUT	Missing Request Body	Pass	/api/tabs PUT	Could not Find User with ID	Pass
16	/api/users PUT	There is No Request Body	Pass	/api/profiles PUT	Invalid Request Body	Pass	/api/tabs PUT	Could not Find Tab with ID	Pass
17	/api/users PUT	Body Properties are Valid	Pass	/api/profiles PUT	ID is Castable to ObjectID	Pass	/api/tabs PUT	Not Authorized to Modify	Pass
18	/api/users PUT	User with Email Already Exists	Pass	/api/profiles PUT	Could not Find ID	Pass	/api/tabs PUT	Return Tab Object	Pass
19	/api/users PUT	User with Username Already Exists	Pass	/api/profiles PUT	Profile Already Exists with User ID	Pass	/api/tabs DELETE	ID is Castable to ObjectID	Pass
20	/api/users PUT	Return Update User Properties	Pass	/api/profiles PUT	Access Denied Invalid JWT Token	Pass	/api/tabs DELETE	Could not Find Tab with ID	Pass
21	/api/users PUT	Exclude Password	Pass	/api/profiles PUT	Could not Find Achievement with ID	Pass	/api/tabs DELETE	Return Success Boolean	Pass
22	/api/users GET email:name	User Exists with Email	Pass	/api/profiles PUT	Return User Profile if Found	Pass	/api/play GET	Return Plays (All)	Pass
23	/api/users GET email:name	User Exists with User Name	Pass	/api/profiles DELETE	Not Implemented	Fail	/api/play GET USER/ID	ID is Castable to ObjectID	Pass
24	/api/subscribe POST	Existing User	Pass	/api/achievements GET	Return an Array of Ach if Found	Pass	/api/play GET USER/ID	Could not Find User with ID	Pass
25	/api/subscribe POST	Confirmation Sent	Pass	/api/achievements GET	Return Empty Array if Not Found	Pass	/api/play GET USER/ID	Return Array of Plays	Pass
26	/api/subscribe POST	Confirmation Error	Pass	/api/achievements GET name	Could not Find Achiev with Name	Pass	/api/play POST	No Request Body	Pass
27	/api/confirm GET	No User in Request Token	Pass	/api/achievements GET name	Return Achievement Object	Pass	/api/play POST	ID is Castable to ObjectID	Pass
28	/api/confirm GET	Unmatching Token	Pass	/api/tabs GET	Return a List of Tabs	Pass	/api/play POST	Could not Find User with ID	Pass
29	/api/confirm GET	Invalid JWT Access Denied	Pass	/api/tabs GET	Return Empty List if No Tabs	Pass	/api/play POST	Could not Cast Tab ID	Pass
30	/api/confirm GET	No Request Body	Pass	/api/tabs GET ID	ID is Castable to ObjectID	Pass	/api/play POST	Could not Find Tab with ID	Pass
31	/api/confirm GET	Existing User	Pass	/api/tabs GET ID	Could not Find ID	Pass	/api/play POST	Return Play Object	Pass

Figure-115 API Tests *(Appendix/API_Test_Sheet.xlsx)*

Each API test case is executed on Postman similarly to the below snapshot. Only the profile delete route fails, as this functionality has not yet been implemented due to ***time constraints and prioritisation***.

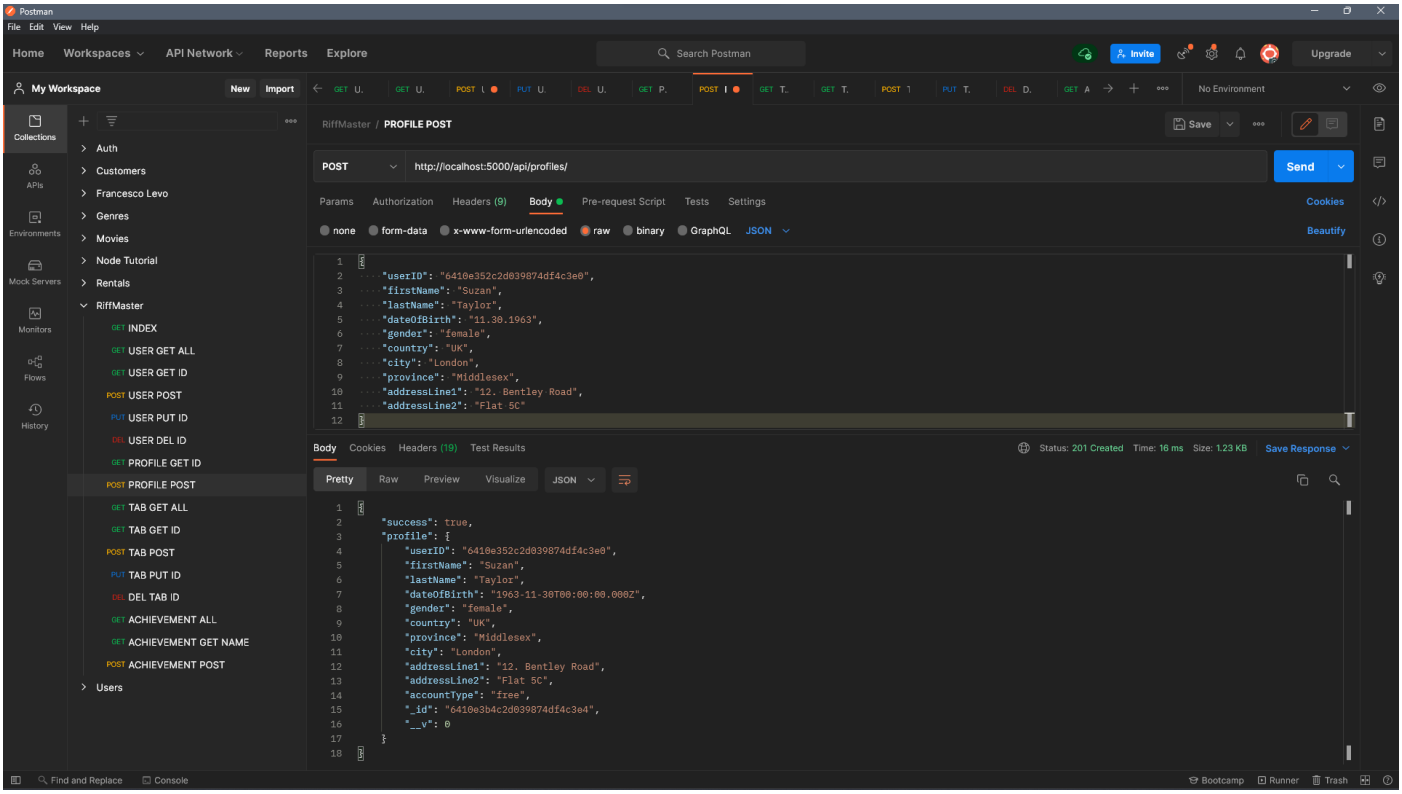


Figure-116 Postman

6.3. AUTOMATED TESTING

Automated testing helps us test code more frequently in less time, and we may be able to catch bugs before deploying them to a release. Additionally, we can securely refactor code without worrying about ***breaking existing functionality***. We used Jest to test our code on the front and backend sides.

6.3.1 UNIT TESTING

Unit tests are functions without external dependencies like databases or web services. They are cheap to write and execute fast but don't give much confidence in the program's reliability because of the separation of dependencies. Some developers have misconceptions that testing two or more functions together describes unit testing. In reality, ***integration testing inspects the components with external dependencies***. They take longer to execute because of the I/O on files or databases but give confidence in the general health of the application.

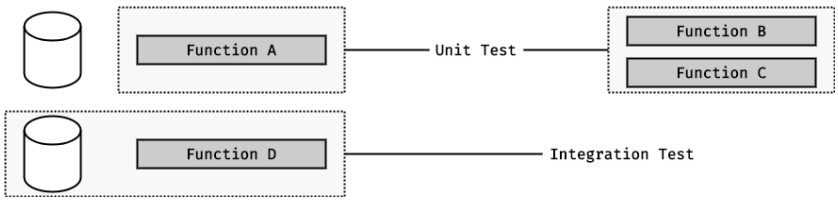


Figure-117 Unit versus Integration Testing (Appendix/Unit.drawio)

We have functions ideal for unit testing, such as our web token signature generator:

```
const { User } = require("../models/user");
const jwt = require("jsonwebtoken");
const config = require("config");
const mongoose = require("mongoose");

describe("generateAuthenticationToken", () => {
    it ("Sould return a JWT Authentication Token", () => {
        const payload = { _id: new mongoose.Types.ObjectId().toHexString(), admin: false }
        const user = new User(payload);
        const token = user.generateAuthenticationToken();

        expect(token).not.toBeNull();
        expect(token._id).not.toBe(null);
        expect(token.admin).not.toBe(null);
        const decoded = jwt.verify(token, config.get("jwtPrivateKey"));
        expect(decoded).toMatchObject(payload);
    });
});
```

```
PS C:\Users\tibia\OneDrive\Desktop\Uni\Uni Project\Code\Source Code\Software\Backend> jest
 PASS  tests/edit.test.js
 PASS  tests/chords.test.js
 PASS  tests/misc.test.js
 PASS  tests/play.test.js
 PASS  tests/subscribe.test.js
 PASS  tests/index.test.js

Test Suites: 6 passed, 6 total
Tests:       30 passed, 30 total
Snapshots:   0 total
Time:        1.261 s
Ran all test suites.
PS C:\Users\tibia\OneDrive\Desktop\Uni\Uni Project\Code\Source Code\Software\Backend>
```

Figure-118 Unit Testing

6.3.2 INTEGRATION TESTING

We need to access and modify external resources for integration testing, like databases. As testing on a live database is not ideal, we must create a ***separate database for testing***.

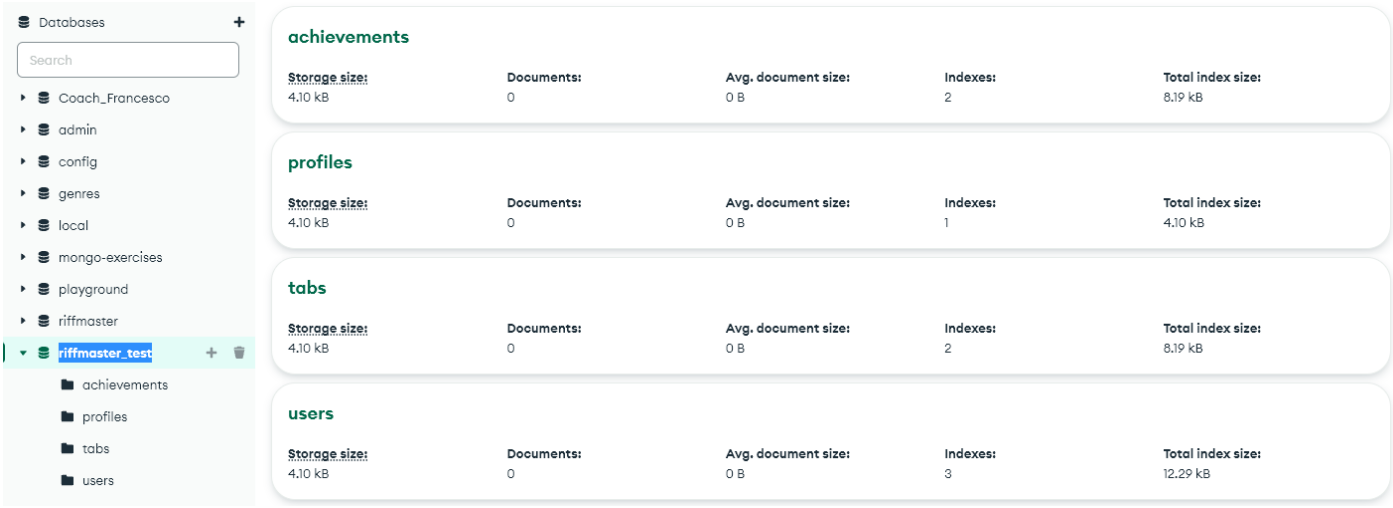


Figure-119 Test DB

Test databases can be used for development and testing without accidentally breaking our deployed database. However, external dependencies, such as our NodeMailer API calls, must be mocked.

```
it('should create a new user and send email confirmation link', async () => {

  User.findOne.mockResolvedValueOnce(null);                    // Mock User.findOne to return null (user does not exist)
  jwt.sign.mockReturnValueOnce('mock-token');                  // Mock jwt.sign to return a token
  Token.prototype.save.mockResolvedValueOnce();                // Mock Token.save to resolve successfully

  const mockTransporter = {                                    // Mock nodemailer.createTransport to return a transporter
    sendMail: jest.fn().mockResolvedValueOnce({ accepted: ['test@example.com'] })
  };
  nodemailer.createTransport.mockReturnValueOnce(mockTransporter);

  await router(req, res);

  expect(User).toHaveBeenCalledWith({ email: req.body.user.email, password: req.body.password });
  expect(jwt.sign).toHaveBeenCalledWith(req.body.user, expect.any(String));
  expect(Token).toHaveBeenCalledWith({ token: 'mock-token' }, expect.any(String));
  expect(Token.prototype.save).toHaveBeenCalled();
  expect(nodemailer.createTransport).toHaveBeenCalledWith({
    auth: {
      user: expect.any(String),
      pass: expect.any(String)
    },
    secure: true,
    port: 465,
    tls: { rejectUnauthorized: false },
    host: expect.any(String)
  });
  expect(mockTransporter.sendMail).toHaveBeenCalledWith(expect.objectContaining({
    to: req.body.user.email,
    html: expect.stringContaining('Verify Email Address')
  }));
```

Figure-120 Test Subscription

Issues: when testing the backend, Jest reads all tests but ignores front-end tests, where package JSON is located. When separate tests are created for the front-end, the project moves up one folder to get the entire front/back project. This move causes breaking changes for our NPM dependencies. As the project presentation deadline was closing, finalising our development was a priority over troubleshooting automated tests, which would require complete refactoring of modules.

6.4. REQUIREMENT TESTS

We have to *test* our project *against the requirement statements* from the specifications to ensure we implemented every feature and functionality and to summarise our project. For this reason, we can tabularise our requirement statements:

#	Description	Result	#	Description	Result	#	Description	Result
T 1.1	User Access to Controller	Pass	T 4.6.2	No Submit Until Valid	Pass	T 7.5	Missed Note Reset	Pass
T 1.2	USB Access 1 (Laptop)	Pass	T 4.6.3	Match Hashed Passwords	Pass	T 7.6	Strum Reset	Pass
T 1.3	USB Access 2 (Controller)	Pass	T 4.6.4	Login Error Messages	Pass	T 7.7	Community Score Com	N / A
T 1.4	Internet Access	Pass	T 4.6.5	Login Attempt Counting	Pass	T 7.8.1	General Accuracy	Pass
T 1.5	Domain Name Access	N / A	T 4.6.6	Redirect to Home Page	Pass	T 7.8.2	Average Score	Pass
T 2.1.a	20 Fret Buttons	Pass	T 5.1	Home Page	Pass	T 7.8.3	Time Precision	Pass
T 2.1.b	6 String Strum Switches	Pass	T 5.2.1	Avatar and User Name	Pass	T 7.9	Saving Performance	Pass
T 2.1.c	Transfer Toggle Switch	Pass	T 5.2.2	Achievements and Scores	Pass	T 7.9.1	Save Accuracy and Scor	Pass
T 2.2	Mouse and Keyboard Interacti	Pass	T 5.2.3	List of Songs	Pass	T 7.9.2	Save Timestamp	Pass
T 2.3	Console Interaction	Pass	T 5.2.4	Author and Titles	Pass	T 7.9.3	Personal Best	N / A
T 2.4	No Batteries Required	Pass	T 5.2.5	Optional Scrollbar	Pass	T 8	Additional Options	Pass
T 2.5	Console Transfer Off Option	Pass	T 5.3.1	Album Cards	Pass	T 8.1	Practice Option	Pass
T 3.1	Web Based GUI	Pass	T 5.3.2	Title and Image	Pass	T 8.2	Audio Warning Msg	Pass
T 3.2	Console Support Control	Pass	T 5.3.3	Clickable Cards	Pass	T 8.3	Jam Session Animatior	Pass
T 3.3	Browser Compatibility	Fail	T 5.3.4	Optional Scrollbar	Pass	T 8.4	Produce Guitar Sounds	Pass
T 3.4	Dark Theme	Pass	T 5.4	Colour Scheme	Pass	T 8.5	Compose Record Opt	Pass
T 3.5	Sign Up / Login Options	Pass	T 5.5	Footer	Pass	T 8.6	Translate Tablatures	Pass
T 4.1	Authentication	Pass	T 6.1	Play Menu Option	Pass	T 8.7	Delete Editor Sheets	Pass
T 4.2.1	Redirect to Register	Pass	T 6.2	Countdown Display	Pass	T 9	Chord Explorer	Pass
T 4.2.2	Register Fields Complete	Pass	T 6.3	Music Audio	Pass	T 9.1	10000+ Chords	Pass
T 4.2.3	Disable Register till Validation	Pass	T 6.4	Pause, Stop Buttons	Pass	T 9.2	Display Chord Cards	Pass
T 4.2.4	Error Messages and Highlights	Pass	T 6.5	Header Title and Author	Pass	T 9.3	Dynamic Card Translat	Pass
T 4.2.5	Refuse Invalid Forms	Pass	T 6.6	Tablature Display	Pass	T 9.4	Barre Chord Display	Pass
T 4.2.6	Name Field Restrictions	Pass	T 6.7	One Line Tab Display	Pass	T 9.5	Root and Type Selectic	Pass
T 4.2.7	Email Format Validation	Pass	T 6.7.1	Base Chord Display	Pass	T 9.6	Play Chords	Pass
T 4.2.8	Password Validation	Pass	T 6.7.2	Finger Positions Display	Pass			
T 4.2.9	Password Lower Upper Min 8	Pass	T 6.7.3	Rhythm Notation	Pass			
T 4.2.10	Matching Confirm Password	Pass	T 6.8	Guitar Animation	Pass			
T 4.2.11	Terms And Conditions Check	N / A	T 6.8.1	Show Frets, Strums	Pass			
T 4.2.12	Terms And Conditions Link	N / A	T 6.8.2	Highlighted Active Positions	Pass			
T 4.3	Disable Submit Button	Pass	T 6.8.3	Missed Error Msg	Fail			
T 4.4	Activation Email Alert	Pass	T 7.1	Current Score	Pass			
T 4.5	Activation Email Link	Pass	T 7.2	Score Multipliers	N / A			
T 4.6	Sign In	Pass	T 7.3	Time Precision	Pass			
T 4.6.1	Input Fields and Submit Button	Pass	T 7.4	Streak Accuracy	Pass			

Figure-121 Requirement Tests (Appendix/Requirement_Shhet.xlsx)

Our prototype has failed tests regarding *browser compatibility*. The application can be opened on different browsers; however, the audio performance is *only acceptable on Google Chrome*. Audio performance issues for specific browsers

are a highly researched area in web development, and significant improvements can only be achieved with extensive experimentation and algorithm restructuring. Lastly, ***several features not specified by the initial requirements have been implemented***, such as a detailed music editor or microcontroller communication protocols.

6.5 USER-ACCEPTANCE TESTING

Finally, we must ensure that our system is easy-to-use for our customers. Our UAT checklist includes every functionality that users must be able to perform:

#	Task	User A	User B	User C	User D
1	Find Sign Up Page	Pass	N/A	Pass	Pass
2	Fill Out User Form	Pass	N/A	Pass	Pass
3	Fill Out User Profile Form	Fail	N/A	Pass	Pass
4	Receive Email Confirmation Link	Pass	N/A	Pass	Pass
5	Login	Pass	N/A	Pass	Pass
6	Find All Finger Positions with Chord F	Pass	Pass	Pass	Pass
7	Find All Finger Positions with Chord G/B	Pass	Pass	Pass	Pass
8	Find All Finger Positions with Chord Asus7	Pass	Pass	Pass	Pass
9	Make Any of the Chords Play (Sound)	Pass	Pass	Pass	Pass
10	Jam: Try Random Jamming	Pass	Pass	Pass	Pass
11	Jam: Try Playing Chords (If User is Familiar with Chords)	Improvements	Pass	Pass	N/A
12	Studio: Try Recording Any Guitar Song with Controller	Fail	Pass	Pass	Pass
13	Set the Title and The Band	Pass	Pass	Pass	N/A
14	Modify a Note's Duration	Pass	Pass	Pass	N/A
15	Modify a Notes Finger Position on the String	Pass	Pass	Pass	N/A
16	Add Chord Name above a Finger Position	Pass	Pass	Pass	N/A
17	Move a Note to the Left or the Right	Pass	Pass	Pass	N/A
18	Insert a Bar to the Left or the Right	Pass	Pass	Pass	N/A
19	Delete an Arbitrary Note	Pass	Pass	Pass	N/A
20	Delete a Complete Bar	Pass	Pass	Pass	N/A
21	Reverse the Last Action (Undo)	Pass	Pass	Pass	Pass
22	Play the Composition	Pass	Pass	Pass	Pass
23	Play One Bar of the Composition on Repeat Mode	Pass	Pass	Pass	Pass
24	Switch off Metronome	Pass	Pass	Pass	Pass
25	Locate the Saving Functionality	Pass	Pass	Pass	Pass
26	Set the Composition Acces to Public	Pass	Pass	Pass	Pass
27	Set the Difficulty to an Arbitrary Value	Pass	Pass	Pass	Pass
28	Change the Tempo of the Composition to 200 BPM	Pass	Pass	Pass	Pass
29	Save the Composition	Pass	Pass	Pass	Pass
30	Open an Arbitrary Composition with High Difficulty Settings	Pass	Pass	Pass	Pass
31	Locate the Play Menu Option	Pass	Pass	Pass	Pass
32	Start the Sample Game	Pass	Pass	Pass	Pass
33	Follow the Game to the End	Pass	Pass	Fail	Fail
34	Give the Sample Game an Other Go Improve Previous Score	Pass	Pass	Pass	Fail
35	Sign Out	Improvements	Pass	Pass	Pass

User A: Sam - Proficient Guitar Player
User B: Leila - Plays Basic Guitar in School
User C: Adrienn - Tried Playing only Few Times in Her Life
User D: Cliff - Never Played on Instruments

Figure-122 UAT *(Appendix/UAT.slsx)*

Our test shows that some of the application's functionality was not as straightforward as expected. ***Those less experienced in guitar playing*** (Users-C/D) also ***found difficulties with some technical aspects of music***. Our test subjects were requested to provide recommendations for each section:

	User A	User B	User C	User D
Device	Strings Need More Sensitivity / Touch Sensor	Device is Too Big for Children	Unfamiliar with Guitars	The Guitar Looks Realistic
Home	-	I Love the Avatars	Cool Layout	-
Chord Explorer	Chord Card Hover is Not Straightforward	-	I don't Know Chords	-
Jam Session	Hard to Find Finger Positions	Enjoyed Playing but it is Difficult	Notation Helps Finding Notes	Never Played Guitar Before, So it was New
Music Studio	Record Should Automatically Play	Had Great Fun with it	Easy to Find Functions	I don't Know Music Enough
Play/Practice	Backing Track Should be Great / No Guitar Sounds	Difficult to Reach High Score	Guitar Also Should have Sound	Sample Song Too Difficult

Figure-123 Feedbacks *(Appendix/Feedback.slsx)*

The next chapter will address these excellent recommendations that shed light on some design issues regarding our instrument's playability and some missing features that'd help complete beginners start exploring guitars.

7. REFLECTION

7.1. IMPROVE PROTOTYPE

This project will continue after the submission as a pet project, and some parts will need to be rebuilt to improve the prototype console:

- Because of the inaccuracies of handcrafting, we should create ***3D guitar models*** and print parts. In addition, 3D printing would result in an enhanced UX, as uneven fret button distances made some tests fail.
- The plastic used (polystyrene) is not eco-friendly and is only used for the initial prototype because of its availability. Instead, we should build plastic parts using ***carbon fibre***.
- We must redesign the guitar neck for ***better ergonomics***. Our current solution works well for fundamental usage but is not user-friendly because users must apply significant effort to play advanced finger positioning. A redesigned guitar neck would omit exposed buttons and rely on strings.
- The strum solution can be improved using ***optical or vibration sensors*** that allow analogue string readings, and the strength of strums and string muting would be possible, just like on authentic instruments.
- While USB is a reliable connection, ***wireless*** solutions should be available.
- We used keypress simulations to convey messages towards the application, and it works seamlessly without delay. Unfortunately, this solution with our simplified protocol is viable only for private projects, but commercial implementations should change to ***MIDI***, the accepted industry standard.
- We may add ***adjustable controls*** like base, middle and high tone or volume potentiometers, similar to semi-acoustic guitars.

7.2. IMPROVE APPLICATION

The current guitar application can demonstrate some essential features of a working music software prototype. However, these features are far from the possibilities and barely scratch the surface of what well-rounded software solutions could achieve.

- ***Browser compatibility*** is a constant worry and a must for modern web applications, and we must improve audio quality and performance for every major browser vendor.
- Users can upload their compositions, which may help translate some popular songs. Unfortunately, we cannot rely on our users to provide an ***extensive tablature library***; considering thousands of official music tabs, we may want to find automation to translate traditional tabs to our tab format.
- Our current tab structure has constraints that may limit our future possibilities, such as the smallest unit of 32nd notes or the 4/4 rhythm base. Therefore, we have to consider ***restructuring our current tab object***.
- ***Social features*** of the app must be completed:
    - Messaging: users must be able to send messages and comment on compositions,
    - Like or bookmark songs,
    - Friend requests and recommendations for extending users' social circle,
    - Users must be able to integrate social media (Facebook/Twitter) accounts.
- A beginner-friendly prototype must have ***walkthroughs*** that cover basic editing and playing functionalities, tutorials for music theory, songs and tabs for scale drilling, with a help menu for assistance.
- The application must have a ***song search with*** filters for genre, artist, year, and composer.
- A ***responsive design*** covering tablet/mobile screen sizes, as mobile applications, even with limited functionalities, would be great for guitarists on the go.

7.3. RECOMMENDATIONS FOR COMMERCIALISATION

This project can potentially be ***patented*** (no patent registered for universal guitar console *Figure-124*), and a commercial application can be built to support it. The commercial RiffMaster would be a futuristic way to learn and play the guitar, and it would extend our prototype's features with some intriguing solutions:

- **Virtual Jam Sessions and Tutoring**: songs could be played similarly to today's Facetime apps. Bands and educators could greatly benefit from such features, and ***augmented reality*** could be applied to enhance user experience.
- **Virtual Competitions**: like Jam Sessions, users could organise competitions through our platforms. Competitions may be of several types, like solo improvisation, guitar battles, or song covers.
- **Collaborative Composing:** music editing and collaboration could be built similarly to how software development uses version control to its advantage. Compositions may be forked and merged, contributors added, and music sheets published under an account.
- **Play Session ML Analytics:** may find patterns of players' mistakes and personalise suggestions for scales and tutorials. Analytics can also be shown after each session as a user performance diagram.
- **AI-Assisted Music Recommendations:** it is nothing new under the sun, but a very important one.
- **Music Editor Copilot:** An ultimate futuristic feature that could revolutionise how music is created with the assistance of AI predictions.
- **Popularity Analysis and Music Research**: By building a complex and vast digital online application with AI, we may assist academic institutions and creative companies with extensive volumes of data. Also, by analysing our music sheets, we may reveal patterns about what makes them popular.

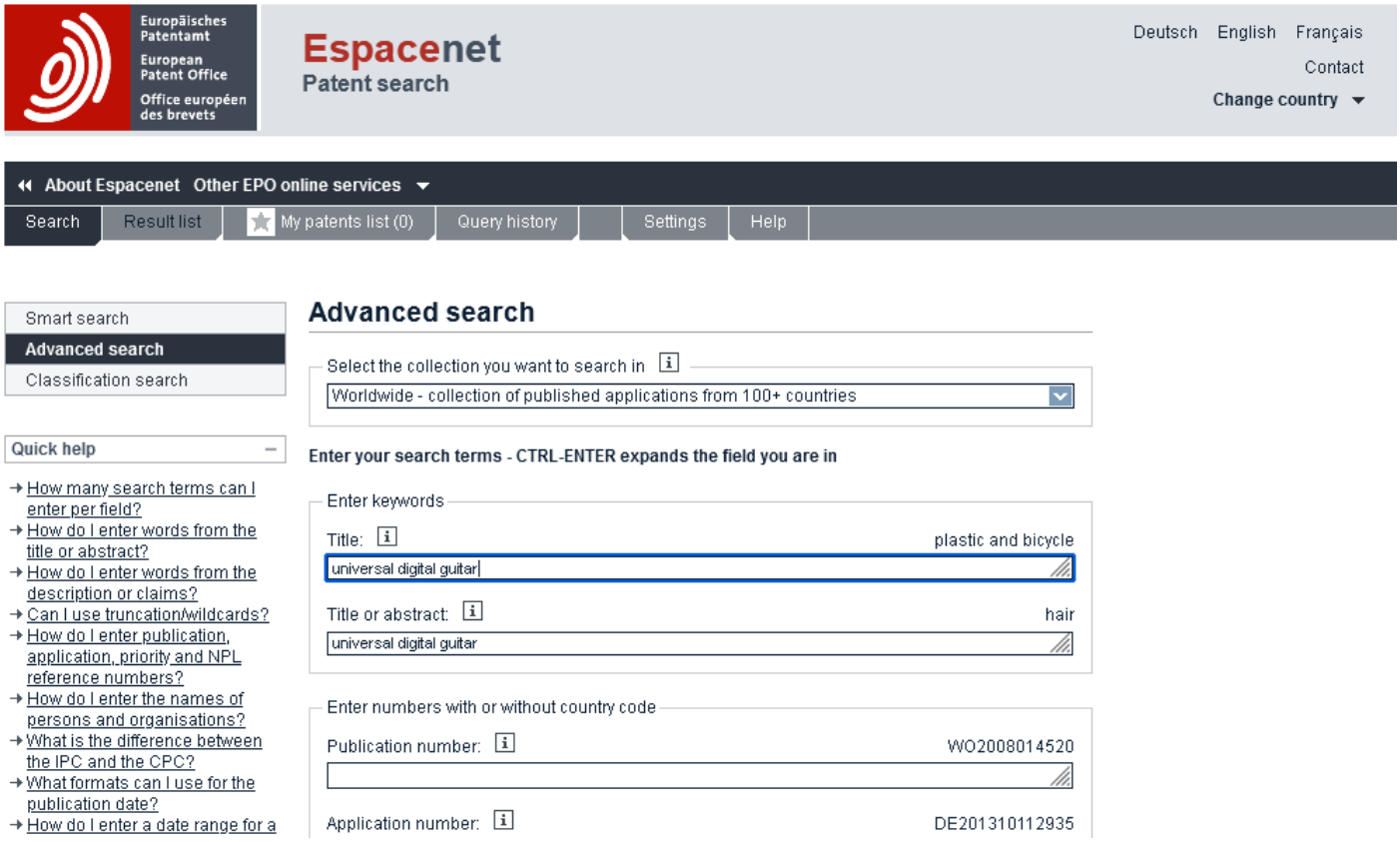


Figure-124 Patent Search on European Patent Office

7.4. PERSONAL DEVELOPMENT

When selecting the topic for my dissertation project, I sought to choose one that meets the assignment criteria and holds potential benefits for our society. As a result, the guitar project emerged as the ultimate fusion of my passion for music and fondness for technologies like JavaScript and web design. However, as I delved deeper into the project, I quickly realised the profound complexity, extensive scope and labour demand of this project. Without a doubt, RiffMaster stands as the most formidable challenge I have encountered thus far in my academic and professional journey.

I had the privilege to design and develop a not yet invented concept, which gave me immense satisfaction. I needed to double my allotted unsupervised project hours (700+) to ensure I could finalise my project within the deadline. Still, there were times when this project's comprehensiveness foreshadowed possible delays or failure, and my priority shifted towards simplified designs that satisfied my proposal. Furthermore, I needed to extend my knowledge with new concepts such as web audio, microcontroller programming and music theory to be able to design my hardware and software components. As a result, I genuinely believe that this intricate project helped me become a better software developer by widening my perspective, and the expertise I gained through these two semesters will help my carrier prospects.

References

Arduino (2022) Available from: https://docs.arduino.cc/built-in-examples/digital/debounce [Accessed 12 October 2022].

Chrzanowski. (2015) *Music Video Game with User Directed Sound Generation.* Patent No. 9061205B2. US:

Csíkszentmihályi, M. (2009) *Flow: The Psychology of Optimal Experience.* Harper-Perennial.

Dave, V.E. and Rajiv, B. (n.d.) *Reading Matrix and Common Bus Keypads*. Cypress.

DigitalMusicNews (2022) Available from: https://www.digitalmusicnews.com/2016/08/01/mi-guitar-easy-to-learn [Accessed 01 October 2022].

Dribin, D. (2000) *Keyboard Matrix Help*.

Edgard, V. (1917) Available from: https://erickuehnl.com/1994/12/30/a-brief-history-of-computer-music [Accessed 02 October 2022].

Endgadet (2022) Available from: https://www.engadget.com/2009-07-22-guitar-hero-rock-band-sales-slide-reminiscent-of-ddr.html [Accessed 09 October 2022].

Eriksson, J. (2016) *Chord and Modality Analysis.* Speech Communication and Technology.

Esparrago, R. (1988) *Kanban*. Production and Inventory Management Journal.

Fccid (2022) Available from: https://fccid.io/VFIBW95123805/User-Manual/Users-Manual-814804 [Accessed 25 November 2022].

Fitzgerald, S. and Shiloh, M. (2012) *The Arduino Project Book*. Arduino-LLC.

Fowler, M. and Highsmith, J. (2001) The Agile Manifesto. *Software Development*, pp.28-35.

HelloMusic (2022) Available from: https://hellomusictheory.com/learn/parts-of-the-acoustic-guitar/ [Accessed 21 November 2022].

Ibrahim R, M. (2022) *The Feasibility Study.* Available from: https://www.academia.edu/29462582/The_Feasibility_Study [Accessed 14 October 2022].

Janzen, D. and Saiedian, H. (2005) Test-Driven Development Concepts, Taxonomy, and Future Direction. 38. Computer. pp.43-50.

Kader, W.M. and Rashid, H. (2012) *Advancement of CMOS Schmitt Trigger Circuits.* Modern Applied Science.

Maguire, M. and Bevan, N. (2002) *User Requirements Analysis. In IFIP World Computer Congress*. Springer.

Menezes Jr, J. and Gusmão, C. (2013) Defining Indicators for Risk Assessment in Software Development Projects. *Clei Electronic Journal*, 16 (1), p.11.

Murphy, R. (2017) *USB HID Basics with PSoC 3 and PSoC 5LP*. Cypress.

PegHead (2022) Available from: https://pegheadnation.com/string-school/music-notation-guide [Accessed 09 November 2022].

PianoDreamers (2023) Available from: https://www.pianodreamers.com/guitar-buying-guide/ [Accessed 13 February 2023].

Schmidt-Jones, C. (2016) *Common Notation for Guitar Tablature Readers*.

Simon, M. and Goes, J. (2013) *Assumption, Limitations, Delimitations, and Scope of the Study.* [Doctoral Dissertation].

Sulaiman, S. and Siti, M.. (2008) Proceedings - International Symposium on Information Technology. *Web Caching and Prefetching: What, Why, and How?*

UltimateGuitar (2022) Available from: https://www.ultimate-guitar.com [Accessed 09 November 2022].

Van, W. (2017) The Waterfall Model and the Agile Methodologies: A Comparison by Project Characteristics. *Research Gate*, 2, pp.1-6.

Warren, G. (2015) *Exploring the Raspberry Pi 2 with C++.* Springer.

WatElectronics (2022) Available from: https://www.watelectronics.com [Accessed 20 December 2022].