

Personal Audio Mixer Scene Manager

Cooper Santillan, Marius Iacob, Kris Melgar Morales, Brennon Hahs, Brian Miller

Department of Computer Science, California State University, Fullerton

CPSC 491: Senior Capstone Project

Prof. Kyoung Shin

May 14, 2023

Department of Computer Science**CPSC 491 Undergraduate Seminar in Computer Science - Project Report for Capstone Project**

Project title	Personal Audio Mixer Scene Manager		
Semester:	Spring 2022	Report date:	May 14, 2023
Team name:	WeDev		
Member name:	Cooper Santillan	Email	CooperSan@csu.fullerton.edu
Member name:	Marius Iacob	Email:	mariusiacob@csu.fullerton.edu
Member name:	Kris Melgar Morales	Email:	cmelgarmorales@csu.fullerton.edu
Member name:	Brennon Hahs	Email:	brennonhahs@csu.fullerton.edu
Member name:	Brian Miller	Email:	bcmiller@csu.fullerton.edu
Faculty advisor:	Kyoung Shin	Reviewer (optional)	

Abstract

Our group partnered with a live entertainment production venue to solve a weekly frustration regarding their audio distribution during performance sets. In the venue, an audio engineer is responsible for distributing at least 12 different stereo audio mixes to various musicians, all unique levels and pans per channel at multiple settings. Our software is able to capture the current state of the professional audio mixer, save the necessary values, and store captured data for later use. Additionally, the software will be able to retrieve the data previously stored and recall the state to the external professional audio console with only the specified values.

The company we worked with is Eastside Christian Church in Anaheim. They are encountering a problem with their musicians being unable to save their personal mixes per week. Most of their musicians do not play consecutively, but some will be there every week, some come back every other week, and some once a month. Because of their lack of attendance and the inconsistency of musicians using the same mix number, an audio engineer spends a significant amount of time trying to replicate a good level adjustment on their mix that was achieved at the end of a previous week's service. This application allows the audio engineer to spend less time doing repetitive work and free more time doing other duties.

1. Introduction

Throughout the entire live music industry, there is a constant challenge when it comes to a rotating set of musicians from one venue to the next. Our software allows musicians to save their mix, levels, and panning, from one audio console/venue to another venue. This will allow less headache for musicians in the back and forward communication to the audio engineer and enable the engineer to focus on audio processing, such as equalization and compression. Previously, no applications were accessible for musicians to do this under any interface.

There must be an understanding of sending UDP packets under the format of a network entertainment standard known as OSC to have participated in this project. The documentation for this professional audio console we are using is a Yamaha CL5. This console has no official documentation to interfacing with it; however, there is unofficial API documentation from various Github projects recording known commands. To enable our understanding of proper communication between the professional audio console and the software we implement, we utilized WireShark for all UDP networks.

1.1 Related Work

<https://mixingstation.app>: Currently, there is one similar application that is capable of moving personal musician mix settings from one mix to another. However, this application does not support the audio console we are trying to work with. (Schumann, n.d.)

https://cz.yamaha.com/files/download/other_assets/5/1407565/RIVAGE_PM_osc_specs_v102_en.pdf: This is the official documentation for the company's flagship professional audio console. There is no official documentation that the flagship network commands will maintain compatibility with the audio console we will be solving. However, there is virtually complete compatibility upon utilization with others. (Yamaha, n.d.)

https://github.com/btendrich/yamaha_osc_translator: This is an extension of the official flagship models documentation with more nuanced specifications not directly explained by the company. (btendrich, n.d.)

1.3 Problem Definition

Musicians cannot keep their preferred levels and pans on audio consoles from venue to venue. Typically this requires the band to arrive early at the venue and spend time sound checking and trying to tune the mixing station to the band's requirements.

1.3 Goals and Objectives

Our goal was to capture, in mass, a large amount and, without any interruption, real-time data for the production of the show and then to store it in a database specified per musician within the requested mix. The first objective was to understand the network communication from an audio console in order to receive the desired data. For this project, it was the audio levels and panning. The second objective was for us to parse that network response from the audio console into a non-relational database structure. Thirdly, we needed to use the database later to recall that data onto the same or different connected console.

Another goal was for the software to be able to load data from one audio console brand, such as Yamaha, onto another audio console brand, such as Allen & Heath. The first objective was to complete the first goal but in a structure that is abstract enough to communicate data from different brands of consoles.

Our final goal was for our application to be synchronized through interactions with a web server or cloud provider to enable synchronization of console configurations when stored and accessed from a created account.

After completing research into our proposed problem, our initial approach would focus on Android, iPhone and Browser development with a React Native application for both Web and Mobile phones.

At the very least, we wanted to deliver a program that takes audio levels and settings from an audio interface and stores them in a database for later use. Ideally, we wanted to deliver a program that can store data from one type/brand of audio interface and recall that data on a different type/brand. Additionally, a program that could synchronize these audio interfaces through a web server based on a user's account would be fantastic.

We also needed a way to keep track of the process that a user needs to follow in order to get their desired outcome. In essence, this would be a user manual of sorts, potentially written as

a PDF document, detailed in a Wiki format on Github, or implemented directly into our application itself.

If needed, we can deliver all of our source code through our GitHub repository or another file-sharing platform. Since this project is focused on cross-platform development, our goal was for the source code for the React Native application to remain almost identical between versions. From our source code, a user of our application should be able to set up local versions of the application on their Android or iOS device and set up a production build hosted either on their hardware or hardware through a cloud provider. We also planned on delivering prebuilt software packages for users inexperienced with building applications from source code to ensure their accessibility.

For the data and database of our application, we planned on setting up instructions on setting up a database for the cloud/web server build of our product and providing a configuration file for each console we support that interacts with our interfaces for storing and setting console data, to simplify the process of supporting additional consoles, whether it be from our tested and configured consoles, or other consoles that are not yet supported.

1.4. Major Outcomes

Upon first approach to this project, the group was determined on creating an application that was to either work on one operating system at the cost of not hosting to another (macOS rather than Windows; Windows rather than macOS). This was, however, replaced with the realization of utilizing the react native framework and docker system so that it would work for all operating systems. As a group, we have discovered the requirements in order to create a cross platform application that can still interact with the clients local network yet have little extra effort in worrying about semantics of the different operating systems.

Another feat we had overcome was developing a schema for how data would be structured in our JSON data workflow. Considering we were aware of possible sharing of data between users, consoles, and most of all mixes, our application needed to be able to withstand the merging of files while preserving newer data and systems in order to tag what is useful and what is not. As a team, we are highly confident that the foundation of our framework, which is the data being preserved, is never to be compromised no matter how many more features we append to future iterations of this application.

2. Software Development Process

We applied a loose agile process to our software development. With all of our busy school and work schedules, it was not realistic to stage a full fledged agile process. The key takeaways from our process were as follows:

1. Worked on high priority features and items first. Features that were not necessary for a basic build of our app were done last.
2. Work was split between loose groups who focused on either UI, frontend, or backend.
3. Each sprint we would set aside time to discuss where everyone in the group was at with their respective tasks and if there were any issues regarding bugs in the code or disagreements on how things were done.
4. Although this is not how the process should go, most of the time the developers were also the testers. Mostly we did not have time to check each other's code, so we trusted that each person would do their own testing.

The process that we applied was not perfect, but it ended up working fairly well for the situation that we were in.

3. Required Environments and Tools for Testing and Building the Application

GitHub and Git

To ensure easy management of different versions of our source code for the application, GitHub and Git were needed. We hosted our files on GitHub, so that synchronization between builds and developers in our group remains consistent.

React Native

Since we planned to allow our application to run between architectures like x86_64 for PCs running Linux, Windows, and macOS. ARM was used for mobile devices running Android or iOS since we need a framework that allows for cross-architecture development. React Native, as described by Wikipedia, is an “open source UI software framework” which can run the same source code natively on different devices, especially for mobile platforms like iOS and Android. For React Native to run correctly on a development environment for our purpose of sending OSC commands, we will need *Node.js*, *NPM and its Command Line Interface*, and *Expo and its Command Line Interface*.

Node.js

React Native requires a runtime environment for Javascript to run correctly, so we used the javascript runtime environment, Node.js, for our application stack.

NPM and its Command Line Interface

NPM is “the world's largest software registry,” according to their own website at <https://docs.npmjs.com/about-npm> (About npm, nd). There are a large number of packages hosted on the site for retrieval and use in React and React Native applications. We imported packages from NPM to reduce the time needed to develop the application, such as with packages like NPM-OSC, which is a library for interacting with devices through the OSC protocol.

Expo and its Command Line Interface

In order to run the development version of our application, the open-source framework Expo was needed. React Native applications can normally be built using tools provided by the IDE and SDK for Android and iOS, like Android Studio and Xcode. However, minimizing the tools needed to develop for all platforms allowed us to code and build quickly without the need to get used to different development tools and their virtual environments. Expo allows for a React Native application to be hosted and viewed in any browser(for PC development and potential production use) and as an application for Android or iOS. It is unknown to us whether we can create a standalone build for PC without the prior development environment.

Flask

During our research into creating an environment that can send TCP commands, we discovered that when using React Native as a front end, it was unlikely that we would be able to send data in the way the audio console expects. While React Native has methods to transmit information using TCP methods, there is extraneous information that should be excluded, as the console will not recognize the commands. Thus we needed to move the sending of TCP commands to a system that works similarly to our proof of concept with the Netcat executable. Flask is a microservice framework that can receive API calls and execute the needed commands in a python script. Python has packages for sending TCP data, which means that our application will use a backend powered by Rest API through Python.

Docker and Docker Desktop

Since it will be difficult to synchronize the dependencies and development environment between our group members, as we all have our own devices, using Docker and Docker Desktop was the simplest way to ensure everyone on the team can develop our application without errors due to dependencies or different operating systems. Docker is a method for building virtual environments that can be easily executed between devices

and is well known for allowing instances of programs and applications to start and shut down to meet the necessary computation load. For our application, it is essential to use these Docker containers to synchronize development between our different groupmates operating systems and architecture. As long as our members build the pre-configured container on their local operating system and have virtualization enabled, they will have immediate access to the same tools and dependencies as every other group member.

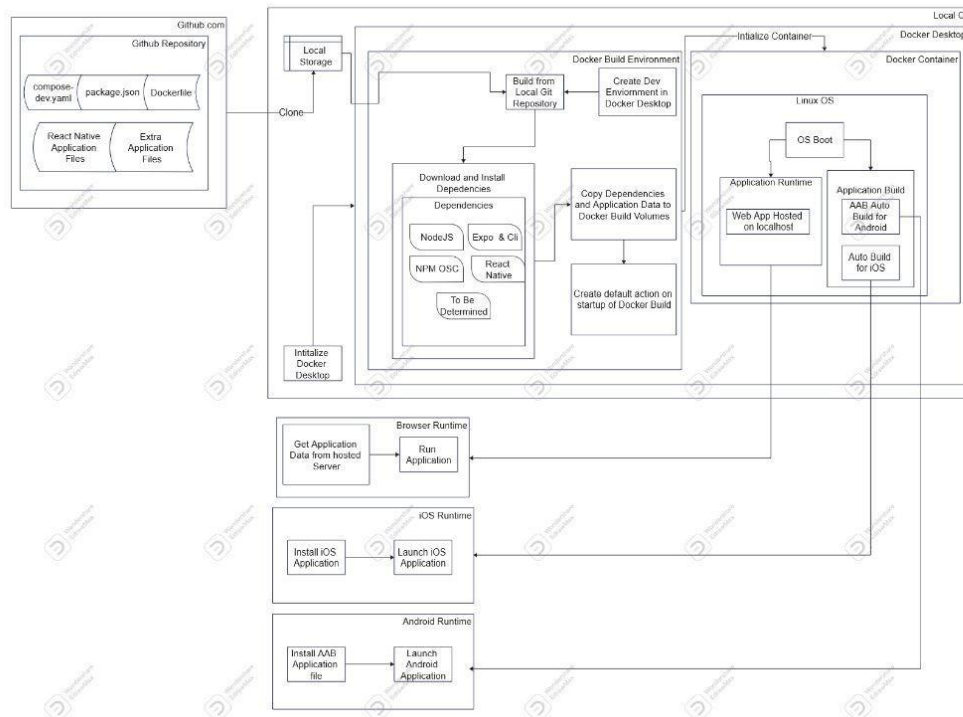


Fig 1. Original Logic for creating a synchronized development environment and runtime builds of the application across platforms.

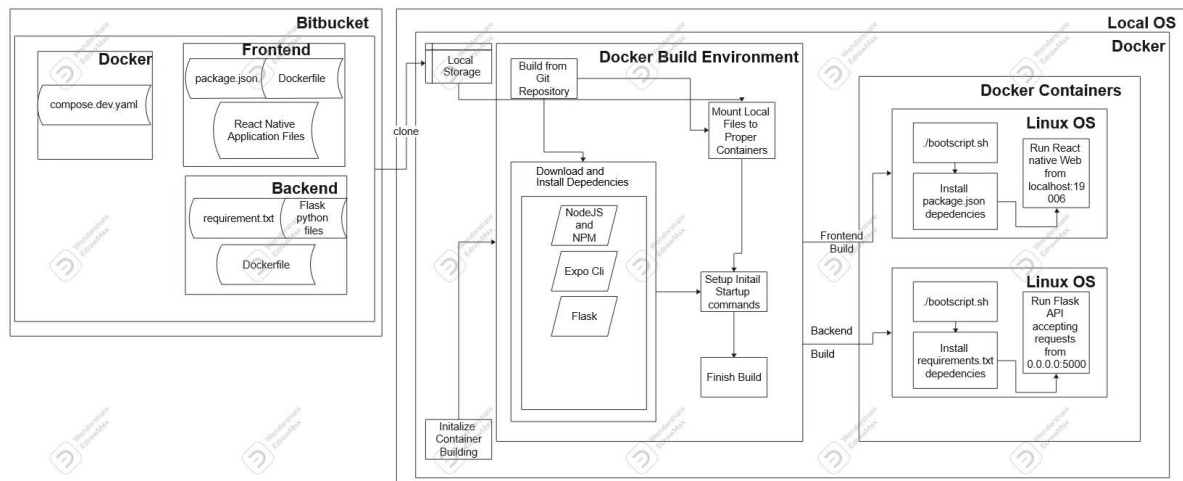


Fig 2. New Logic for creating a synchronized development environment and runtime builds of the application for web development

Figure 1 shows the event chain that we originally planned to implement to get our React Native application built for both mobile and web releases; however as we scrapped the mobile version, our new diagram was changed accordingly to the one in figure 2.

It was necessary for our developers to clone a GitHub repository that contains different configuration files, such as compose-dev.yaml and Dockerfile for building our Docker build and container, as well as package.json for installing the required NPM packages needed to run our React Native Application. This allowed any one of our developers to open up Docker Desktop on their Local Operating System and choose to build our development environment from the Github repository and run it in a Linux Virtual Machine for easy hosting of the Web application.

4. Required Format for Storing Console Data:

Local Storage

While React Native allows for applications to be run on multiple architectures seamlessly, it may not be possible to use the same method to store data locally between PC, Androids, and iOS. We originally planned to use a package called AsyncStorage for React Native to save and load the profile of the audio consoles using this format originally.

```
{
  "Kendalls Mix":[
    "1": {"Name": "Bass", "Level":-4, "Pan":35, "On":false},
    "2": {"Name": "Kick", "Level":-24, "Pan":0, "On":true},
    "3": {"Name": "AG", "Level":0, "Pan":-25, "On":true},
    ...,
    "72": {"Name": "Mic 4", "Level":3, "Pan":5, "On":true}
  ],
  "Toms Mix":[
    "1": {"Name": "Bass", "Level":0, "Pan":0, "On":true},
    "2": {"Name": "Kick", "Level":-37850, "Pan":-20, "On":true},
    "3": {"Name": "AG", "Level":-3, "Pan":5, "On":true},
    ...,
    "72": {"Name": "Mic 4", "Level":-10, "Pan":5, "On":false}
  ]
}
```

Fig 3. Proposed JSON format for storing configuration data from Yamaha CL5

However, as we removed the implementation of the mobile version, we could stick to one solution to solve the storage issues, using the FileReader library provided by the browser themselves to create web links to download our data, and also open the host device native file explorer to load a file. Our final format for saving the data from the different mixes and channels appeared like this for our application when saving and loading single mixes,

```
{
  "filename": "CL5.json",
  "version": "0.1",
  "user": "a5e135c",
  "mixes":
  {
    "1": {
      "Name": "Bass",
      "Level": -32768,
      "Pan": 35,
      "On": false
    },
    "2": {
      "Name": "Guitar",
      "Level": 10,
      "Pan": 35,
      "On": true
    }
  }
}
```

Fig 4. Actual JSON format for storing configuration data from Yamaha CL5 single mix edition

where a JSON object stored at “mixes” contains the profile data like how the original proposal had it, except without the array.

5. Application Requirements

In order for the client to utilize this application, they must access any network-connected device that has any desirable web browser. This application can not be in a VLAN with device isolation profiled on all networked clients, rather for our client devices to be able to discover other network controllable devices.

This application has no utilization without a companion professional audio console (in our current iteration: it must be a Yamaha professional console with CL, QL, or PM prefix/suffix) so one must be within the same local network as the client in need of querying information from the console.

There must be a computer within the network of the audio console that will be able to host the docker container for web hosting for clients and local networking UDP calls. The middle-man computer only utilizes one CPU core, in no need of a GPU, and at intense utilization may need half a gigabyte of RAM. An ideal candidate for this application would be a raspberry pi considering that he has plenty of resources available for the docker container to utilize to both host the website for a client and also create TCP calls for the audio console.

Storage will depend on how many users/mixes the client is required to save but being that files to save are small, a 2gb will be the minimum amount storage needed for the application alone in order to save about 100 users worth of data. Please refer to our Appendix I for thorough instructions for installing the docker container on this middle-man computer.

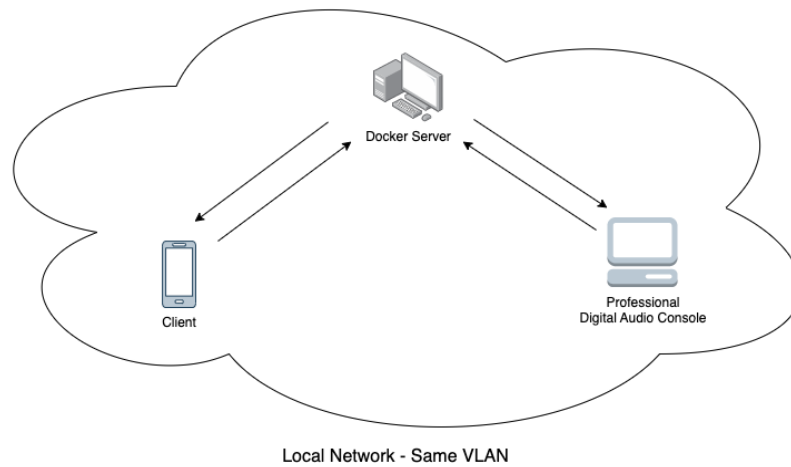


Fig 5. Client to Console Connection

Figure 5 shows the interaction between the client's device when connecting to the web browser, the web hosting computer that is named "Docker Server" and this server interacting with the audio console.

6. App Design and Architecture

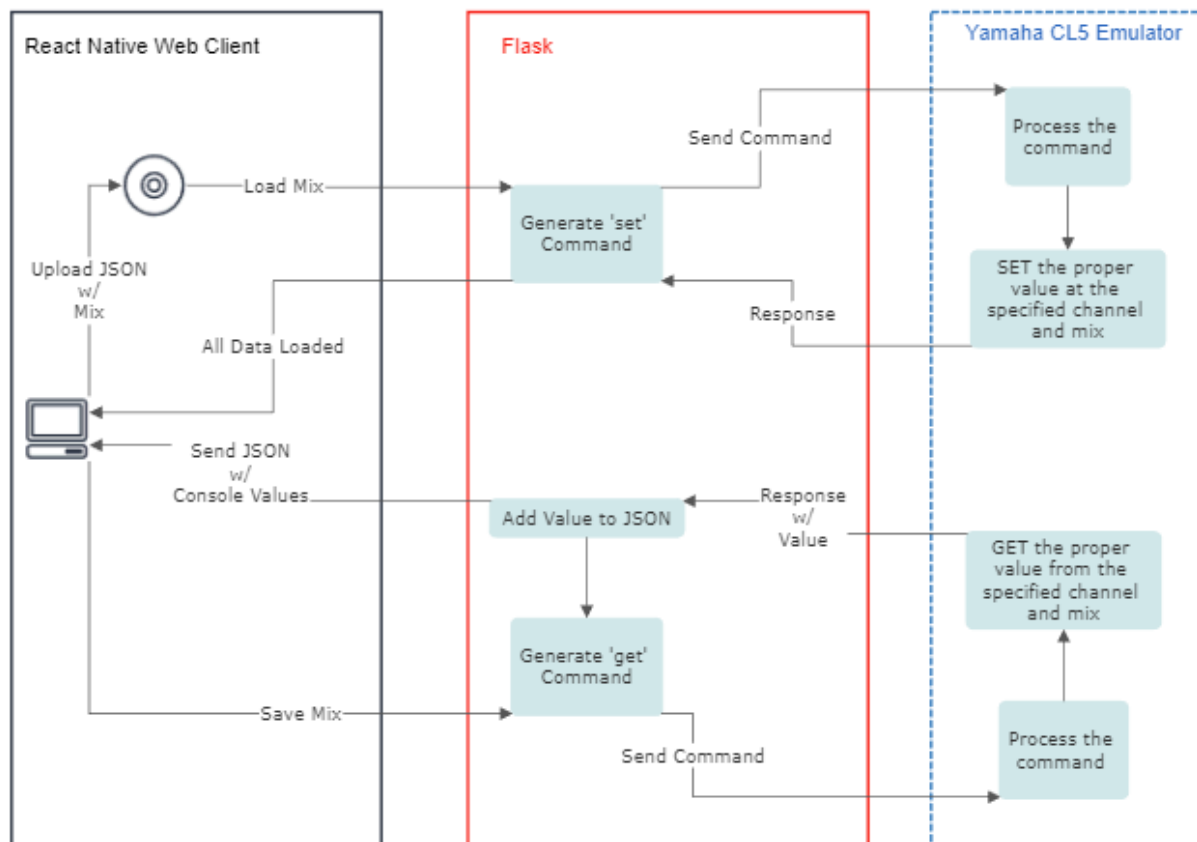


Fig 6. Basic flow design of our app

7 . Requirements

Emulator

Create a python script that allows a user to spin up a fake Yamaha CL5 console that can respond to the commands sent over TCP.

Assumptions

User already has access to a python interpreter. Users have knowledge on how to use the python scripting language to import the emulator into their code wherever needed.

Requirement	Notes
The emulator must be able to receive TCP messages from all IP addresses, and it must return the data request from the state of the emulator as a TCP message.	

API

Create API calls within Flask and Python that can communicate with a Yamaha CL5 to retrieve all information about its state or load new information to its state.

Assumptions

Users have access to Frontend and Backend containers of React Native applications and they are running.

Requirement	Notes
Users must be able to call a function that sends an API request to the backend, to gather and return the state of the Yamaha CL5 as a json file.	Preferably JSON data is saved/downloaded to local storage and works with all platforms

File should be saved in universal JSON file format defined in our data format section

8. Implementation

Yamaha CL5 Emulator:

Due to the team not having our own personal Yamaha CL5 to test with, there needed to be a way to test our code according to the response of the CL5. As a result we wrote a script, which can simulate the responses of a Yamaha CL5 so that our developers could test their code against the expected responses.

A module called `emulated.py` was made that would initialize a server that pretends to be an audio Console, based on the configuration of the given Host, Port, Maximum Input Channels, and Maximum Mixes. The function called `echoServer()` will take the parameters and host an emulator that responds based on the values of `dummy.json` in the same folder.

The emulator script returns profile data when asking for the state of Yamaha CL5. The emulator responds to the get and set commands in the way that a Yamaha CL5 console would respond. Currently there is no valid handling of incorrect inputs, as the emulator expects commands in the correct format within the bounds of each type of argument.

Save and Downloading Yamaha Profile Data:

In the React Native web app, a user must be able to download the JSON data format of a Yamaha console to their device with an API call. An API call was added to the Flask backend called `getConfigProfile` which will send messages to the Yamaha emulator or actual console depending on the arguments passed in the JSON detailed in the API documentation. The flask API requests the information from the device and returns the data formatted back to the API caller.

A matching file and function called `getConfigProfile` was added to the React native workspace to fetch the JSON data from the Flask API and then return it to the React Native frontend.

In the case of users needing to move their configuration file data manually, we have implemented a Javascript function that enables any JSON data to be downloaded to the local machine the web app is running on. This feature was implemented as the function `saveData()`, which takes JSON files and saves them locally.

While getConfigProfile worked for our requirements to fill in the UI, we had to make new edits where we separated getConfigProfile into two different API paths getYamahaProfile (works like getConfigProfile did) and getSingleYamahaMix(pulls a single mix instead) as well as implement to functions with the same names in React Native, which return the JSON file that was generated by the API.

Data Loading to Yamaha CL5

Loading data to a CL5 requires a specially formatted 'set' command. It goes something like 'set <attribute> <mix> <channel> <value>'. Basically what this is saying is that it will load a value into a certain attribute of a channel inside of a mix that is in the CL5. For example loading a 'pan' value of 35 to mix 1, channel 1 would look like this:

```
set MIXER:Current/InCh/ToMix/Pan 1 1 35
```

Our flask backend's function, setConfigProfile, is used to send these commands with the specified values from a user loaded JSON file to our emulated CL5. The function concatenates the specific profile data to their respective set commands and then sends them to the CL5 emulator. Nothing is returned to the user besides a confirmation from our backend that everything in their JSON file has been loaded to the console.

After further development the function was split into two functions called setSingleYamahaMix and setYamahaProfile, where either a whole console profile or a single mix can be loaded into the console. The function takes the parameters from the JSON file and returns a JSON response as detailed in the API Documentation.

9. Front-End UI

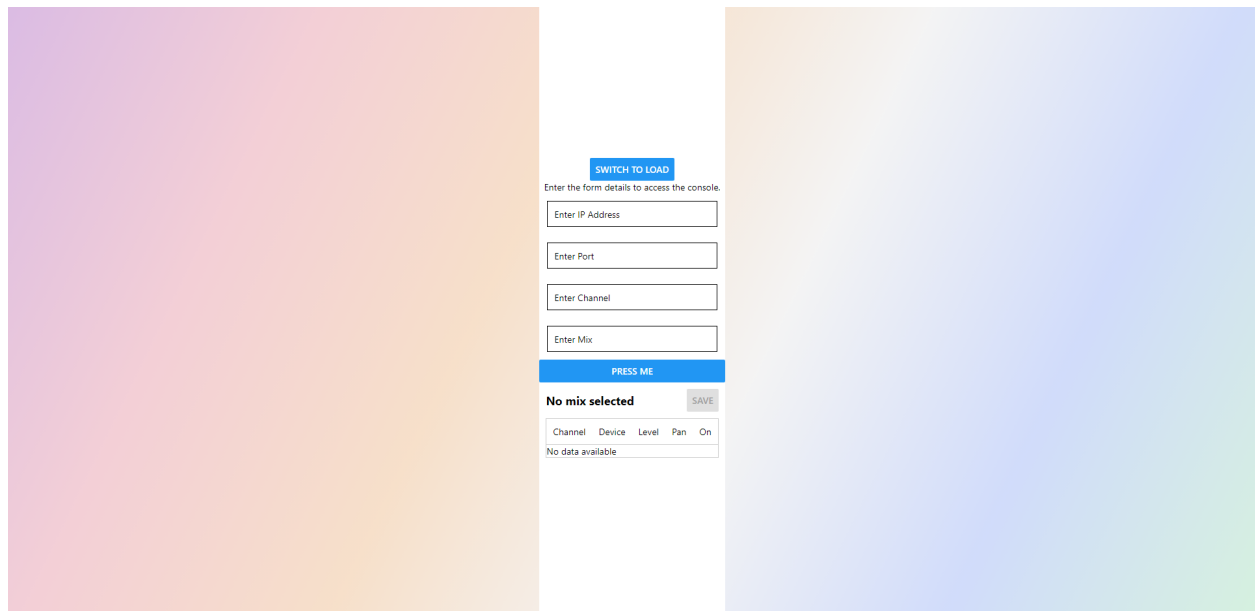


Fig 7. Web UI Splash Page

Input Field and Button

The input field contains four parameters that are used to connect to the console: IP address, Port Number, Number of Channels, and Number of Mixes. Once these fields are filled, the “Press Me” button sends the signal to connect to the console.

Table Component

The Table Component is a simple table that consists of five different column components: Channel, Device, Level, Pan, and On. The component uses DocumentPicker to read the JSON file and parse the data it needs to fill the cells.

Save/Load State Switch

The Save and Load states are swapped by using the button at the top of the page. The default state of the application is the Save State to accept the current state of the table. The Load State allows the user to select a pre existing JSON mix file that will be uploaded to the console.

SWITCH TO LOAD

Enter the form details to access the console.

127.0.0.1

5001

72

1

PRESS ME

CL5.json

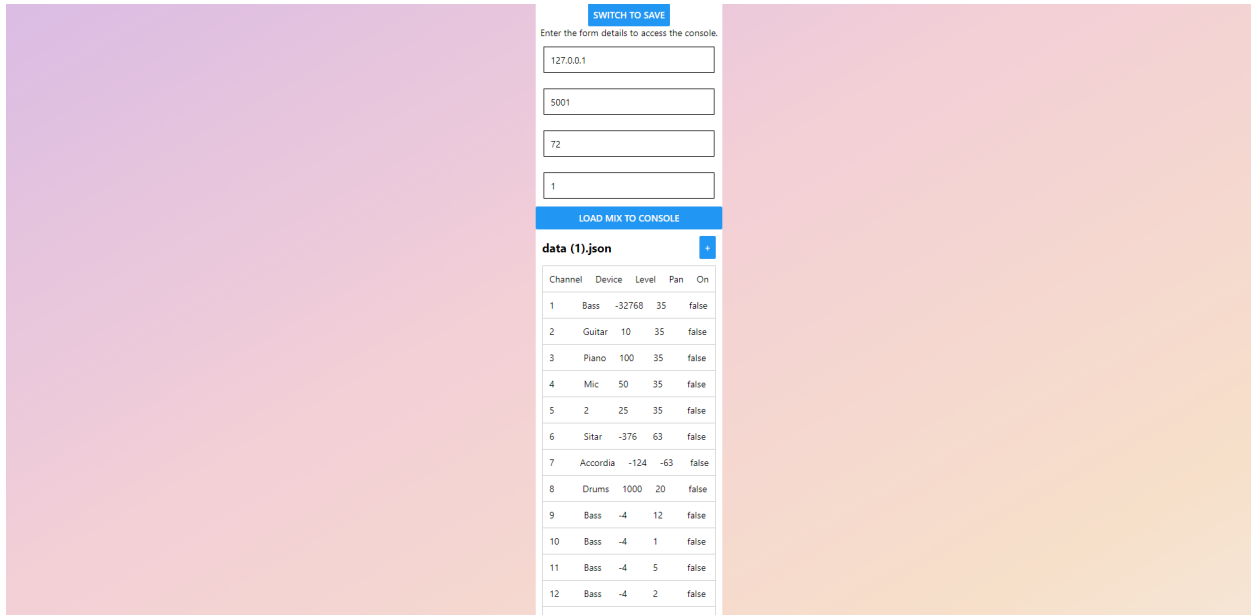
SAVE

Channel	Device	Level	Pan	On
1	Bass	-32768	35	false
2	Guitar	10	35	false
3	Piano	100	35	false
4	Mic	50	35	false
5	2	25	35	false
6	Sitar	-376	63	false
7	Accordia	-124	-63	false
8	Drums	1000	20	false
9	Bass	-4	12	false
10	Bass	-4	1	false
11	Bass	-4	5	false
12	Bass	-4	2	false

Fig 8. Web UI Save State After Connecting to Console

Save State Flow

1. Start Application and enter address of site ip.(ie localhost:19006)
2. Enter details for the console connection(ie. Connect to console at 127.0.0.1:5001)
3. Enter details for parameters of console(ie. Console has 72 channels, and a mix one should be pulled from the console.
4. Hit Press Me to send the POST request to Flask Backend for console data.
5. Application updates empty table with data from the requested mix from the console when POST completes.
6. Press Save to download a JSON file containing console information.



The image shows a web application interface for loading state after selecting a file. It features a central form with input fields for console connection details and a table of loaded data.

Form Fields:

- SWITCH TO SAVE** (button)
- Enter the form details to access the console.
- 127.0.0.1 (IP address)
- 5001 (Port)
- 72 (Channels)
- 1 (Mix)
- LOAD MIX TO CONSOLE** (button)

data (1).json (table)

Channel	Device	Level	Pan	On
1	Bass	-32768	35	false
2	Guitar	10	35	false
3	Piano	100	35	false
4	Mic	50	35	false
5	2	25	35	false
6	Sitar	-376	63	false
7	Accordia	-124	-63	false
8	Drums	1000	20	false
9	Bass	-4	12	false
10	Bass	-4	1	false
11	Bass	-4	5	false
12	Bass	-4	2	false

Fig 9. Web UI Load State After Selecting File

Save State Flow

1. Start Application and enter address of site ip.(ie localhost:19006)
2. Enter details for the console connection(ie. Connect to console at 127.0.0.1:5001)
3. Enter details for parameters of console(ie. Console has 72 channels, and a mix one should be pulled from the console.
4. Hit the + button to open the File Explorer
5. Select JSON file containing the desired mix
6. Hit Press Me to send the POST request to Flask Backend for console data.
7. Application shows a Toast notification telling the user that API successfully updated the data in the console.

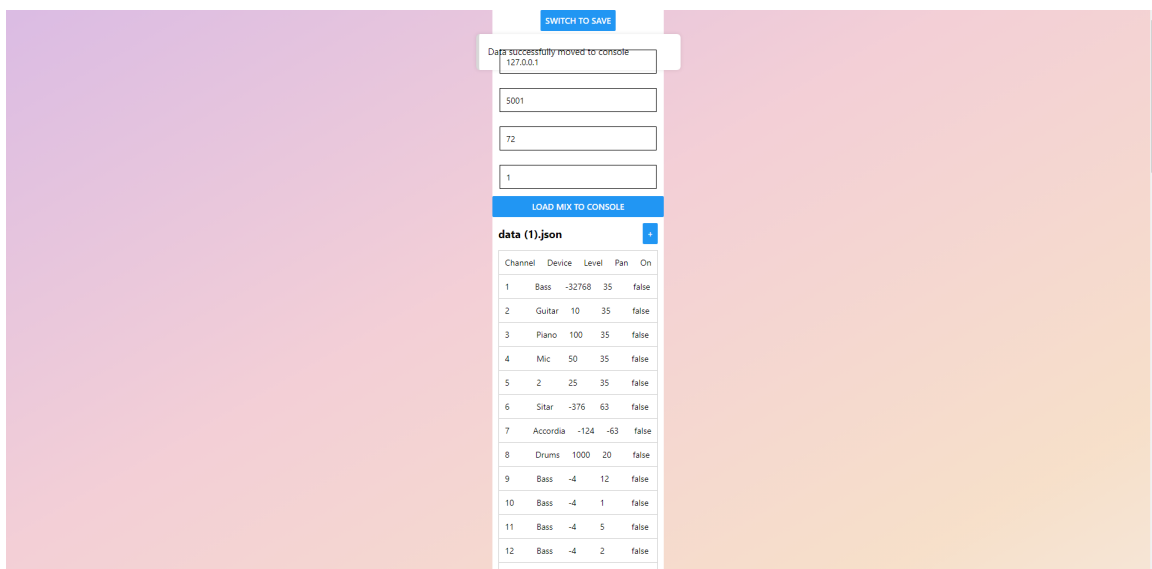


Fig 10. Web UI Load State After Loading File to Console Success

10. API Documentation

App.getYamahaProfile

Verb: POST

Path: /App/getYamahaProfile

JSON Parameter	type	Description
host	string	IP address of Audio Console
port	string number	Port that the console is listening on
channel	string number	Max channel size of console
mixes	string number	Max mixes on console
isDummy	bool	Controls whether emulator of audio console is created when API is called

Response

Content Type: application/json

Type: object

Object Properties

Response

Returns a JSON file containing information about Yamaha Console with respect to each channel and mix, and their values for the following paths.

```
validInfix = ['MIXER:Current/InCh/Label/Name',  
              'MIXER:Current/InCh/ToMix/Level',  
              'MIXER:Current/InCh/ToMix/Pan',  
              'MIXER:Current/InCh/ToMix/On']
```

JSON Response should appear in this format

```
jsonFormat = {  
  "filename": "CL5.json",  
  "version": "0.1",  
  "timestamp": 'temp',  
  "user": "",  
  "mixes": []}
```

App.getSingleYamahaMix

Verb: POST

Path: /App/getSingleYamahaMix

JSON Parameter	type	Description
host	string	IP address of Audio Console
port	string number	Port that the console is listening on
channel	string number	Max channel size of console
mixes	string number	Mix to pull from console
isDummy	bool	Controls whether emulator of audio console is created when API is called

Response

Content Type: application/json

Type: object

Object Properties

Response

Returns a JSON file containing information about Yamaha Console with respect to each channel for the selected mix, and their values for the following paths.

```
validInfix = ['MIXER:Current/InCh/Label/Name',  
             'MIXER:Current/InCh/ToMix/Level',
```

```
'MIXER:Current/InCh/ToMix/Pan',  
'MIXER:Current/InCh/ToMix/On']
```

JSON Response should appear in this format

```
jsonFormat = {  
  "filename": "CL5.json",  
  "version": "0.1",  
  "timestamp": 'temp',  
  "user": "",  
  "mixes": {<MIX JSON OBJECT HERE>}}
```

App.setYamahaProfile

Verb: POST

Path: /App/setYamahaProfile

JSON Parameter	type	Description
host	string	IP address of Audio Console
port	string number	Port that the console is listening on
channel	string number	Max channel size of console
mixes	string number	Max mix size of console.
isDummy	bool	Controls whether emulator of audio console is created when API is called
file	JSON Object	JSON File as an object containing state to load to Console

Response

Content Type: application/json

Type: object

Object Properties

Response

Returns a JSON file verifying that the Console with respect to each channel and mix was properly set according to the received JSON file from the POST request, based on these parameter paths for the console TCP commands.

```
validInfix = ['MIXER:Current/InCh/Label/Name',  
              'MIXER:Current/InCh/ToMix/Level',  
              'MIXER:Current/InCh/ToMix/Pan',  
              'MIXER:Current/InCh/ToMix/On']
```

JSON Response should appear in this format

```
{  
  "didLoad": true  
}
```

App.setSingleYamahaMix

Verb: POST

Path: /App/setSingleYamahaMix

JSON Parameter	type	Description
host	string	IP address of Audio Console
port	string number	Port that the console is listening on
channel	string number	Max channel size of console
mixes	string number	Mix to push to console
isDummy	bool	Controls whether emulator of audio console is created when API is called
file	JSON Object	JSON File as an object containing state to load to Console

Response

Content Type: application/json

Type: object

Object Properties

Response

Returns a JSON file verifying that the Console with respect to each channel for the selected mix was properly set according to the received JSON file from the POST request, based on these parameter paths for the console TCP commands.

```
validInfix = ['MIXER:Current/InCh/Label/Name',  
             'MIXER:Current/InCh/ToMix/Level',  
             'MIXER:Current/InCh/ToMix/Pan',  
             'MIXER:Current/InCh/ToMix/On']
```

JSON Response should appear in this format

```
{  
  "didLoad": true  
}
```


11. Testing

<u>Test Case #:</u> 1	<u>Test Case Name:</u> Getting Mixes API
<u>System:</u> Web	<u>Subsystem:</u> Flask API
<u>Test case designed by:</u> Kris Melgar Morales	<u>Design Date:</u> 4/12/2023
<u>Short description:</u>	Native app receives dummy data from getSingleYamahaMix and getYamahaPath API path, and displays and downloads data on device.

<u>Pre – conditions:</u>
Flask API is running on the container, and is accessible to the client either by IPV4 or localhost.

Step	Action	Expected system response	Comment
1	Open Rested extension for API testing	Form for entering API details appears	Using RESTED from firefox extension store - Kris

2	Enter details into form Details seen below in Relevant Images	N/A	
3	Hit Send request	Response 200 OK from API	JSON object {} in mixes when getSingleYamahaMix and JSON array [] when getYamahaProfile

Test Scripts

Scenario 1 getYamahaProfile:

Tester	Steps	Expected Results	Pass/Fail	Date Tested
Kris	1	N/A	PASS	4/22/2023
Kris	2	N/A	N/A	4/22/2023
Kris	3	Response 200, data appears in mixes in JSON array	N/A	4/22/2023

Scenario 2 getSingleYamahaMix:

Tester	Steps	Expected Results	Pass/Fail	Date Tested
Kris	1	N/A	N/A	4/22/2023
Kris	2	N/A	N/A	4/22/2023
Kris	3	Response 200, data appears in mixes in JSON object	PASS	4/22/2023

Relevant Images and Diagrams

The screenshot shows a REST client interface with the following configuration:

- Method:** POST
- URL:** http://localhost:5000/getYamahaProfile
- Headers:** (collapsed)
- Basic auth:** (collapsed)
- Request body:** (expanded)
 - Type:** JSON
 - Fields:**

Key	Value
channel	72
mix	1
PORT	5001
HOST	127.0.0.1
isDummy	True

Fig 11. Scenario 2: getYamahaProfile JSON form

The screenshot shows the response of the REST client:

- Status:** 200 OK
- Headers:** (collapsed)
- Preview:** (expanded)

```
{ "filename": "CL5.json", "version": "0.1", "timestamp": "2023-04-23 00:19:55.893253", "user": "", "mixes": [{"1":
```

Fig 12. JSON response with array object in “mixes” key value

Request

POST

[http://localhost:5000/getSingleYamahaMix](#)

[Headers >](#)

[Basic auth >](#)

[Request body v](#)

Type

<input type="text" value="channel"/>	<input type="text" value="72"/>
<input type="text" value="mix"/>	<input type="text" value="1"/>
<input type="text" value="PORT"/>	<input type="text" value="5001"/>
<input type="text" value="HOST"/>	<input type="text" value="127.0.0.1"/>
<input type="text" value="isDummy"/>	<input type="text" value="True"/>

[+Add parameter](#)

Fig 13. Scenario 2: getSingleYamahaMix JSON form

Response (1.053s) - [http://localhost:5000/getSingleYamahaMix](#)

200 OK

[Headers >](#)

[Preview >](#)

```
{"filename": "CL5.json", "version": "0.1", "timestamp": "2023-04-23 00:18:58.477166", "user": "", "mixes": {"1": {"Name": "Bass",
```

Fig 14. JSON response with object in “mixes” key value

<u>Test Case #:</u> 2	<u>Test Case Name:</u> Getting Mixes React
<u>System:</u> Web	<u>Subsystem:</u> React Native and Flask API
<u>Test case designed by:</u> Kris Melgar Morales	<u>Design Date:</u> 4/12/2023
<u>Short description:</u>	Native app receives dummy data from getSingleYamahaMix and getYamahaProfile API path, and displays and downloads data on device.

Pre – conditions:

React Native web server and Flask API are running on the containers, and are accessible to the client either by IPV4 or localhost.

Step	Action	Expected system response	Comment
1	Enter website from localhost:19006 or 192.168.##(based on ipv4 of host os hosting docker containers)	React Native Application opens in browser	

2	Enter details into the test form for saving. HOST:127.0.0.1 PORT:5001 MIX:1 CHANNEL:72	Form visually updates when typing parameters	
3	Open Developer Console at start recording Network traffic	No expected response	
4	Submit details using button	200 Response from Flask API, with JSON data format of Audio Console data. Data for a single mix is displayed on the table.	
5	Press save button	Json data is saved to the device.	Should only be able to save when JSON data is loaded from the console.

Test Scripts

Scenario 1:

Tester	Steps	Expected Results	Pass/Fail	Date Tested
Kris	1	Website loads	PASS	4/22/2023
Kris	2	Parameters are visible in form	PASS	4/22/2023
Kris	3	N/A	N/A	4/22/2023
Kris	4	Data received from API and displayed to table	PASS	4/22/2023
Kris	5	Data downloads to the device.	PASS	4/22/2023

Relevant Images and Diagrams

The screenshot shows a web form interface. At the top, there is a blue button labeled "SWITCH TO LOAD". Below it, a text prompt says "Enter the form details to access the console." followed by four input fields containing the values "127.0.0.1", "5001", "72", and "1". Below these fields is a blue button labeled "PRESS ME". Underneath the button, the text "No mix selected" is displayed next to a grey "SAVE" button. At the bottom, there is a table with five columns: "Channel", "Device", "Level", "Pan", and "On". The table currently contains the text "No data available".

Channel	Device	Level	Pan	On
No data available				

Fig 15. Scenario 1: Default Save Page with parameters filled in

SWITCH TO LOAD

Enter the form details to access the console.

127.0.0.1

5001

72

1

PRESS ME

CL5.json

SAVE

Channel	Device	Level	Pan	On
1	Bass	-32768	35	false
2	Guitar	10	35	false
3	Piano	100	35	false
4	Mic	50	35	false
5	2	25	35	false
6	Sitar	-376	63	false
7	Accordia	-124	-63	false
8	Drums	1000	20	false
9	Bass	-4	12	false
10	Bass	-4	1	false
11	Bass	-4	5	false
12	Bass	-4	2	false
13	Bass	-4	4	false

Fig 16. Scenario 1: Save page filled from console through API, save button enabled

<u>Test Case #:</u> 3	<u>Test Case Name:</u> Setting Mixes API
<u>System:</u> Web	<u>Subsystem:</u> Flask API
<u>Test case designed by:</u> Kris Melgar Morales	<u>Design Date:</u> 4/12/2023
<u>Short description:</u>	API receives data for setSingleYamahaMix and setYamahaProfile API path, and updates the device accordingly.

Pre – conditions:

Flask API is running on the container, and is accessible to the client either by IPV4 or localhost.

Step	Action	Expected system response	Comment
1	Open Rested extension for API testing	Form for entering API details appears	Using RESTED from firefox extension store - Kris
2	Enter details into form Details seen below in Relevant Images	N/A	

3	Hit Send request	Response 200 OK from API	JSON object {} in mixes when getSingleYamahaMix and JSON array [] when getYamahaProfile
---	------------------	--------------------------	---

Test Scripts

Scenario 1 getYamahaProfile:

Tester	Steps	Expected Results	Pass/Fail	Date Tested
Kris	1	N/A	PASS	4/22/2023
Kris	2	N/A	N/A	4/22/2023
Kris	3	Response 200, data appears in mixes in JSON array	N/A	4/22/2023

Scenario 2 getSingleYamahaMix:

Tester	Steps	Expected Results	Pass/Fail	Date Tested
Kris	1	N/A	N/A	4/22/2023
Kris	2	N/A	N/A	4/22/2023
Kris	3	Response 200, data appears in mixes in JSON object	PASS	4/22/2023

Relevant Images and Diagrams

The screenshot displays an API client interface for configuring a POST request. The 'Request' section shows the method 'POST' and the URL 'http://localhost:5000/setYamahaProfile'. The 'Headers' section has 'Content-Type' set to 'application/json'. The 'Request body' is set to 'Custom' and contains a JSON object with the following structure:

```
{
  "filename": "CL5.json",
  "version": "0.1",
  "user": "a5e135c",
  "mixes": [
    {
      "1": {
        "1": {
          "Name": "Bass",
          "Level": -32768,
          "Pan": 35,
```

The interface includes expandable sections for 'Headers', 'Basic auth', and 'Request body'. The 'Request body' section is currently expanded, showing the JSON payload.

Fig 17. Scenario 1: API form for setYamahaProfile, where mixes contains mix array of objects

Request

POST

http://localhost:5000/setSingleYamahaMix|

Headers

Content-Type

application/json

+Add header

Basic auth

Request body

Type

Custom

```
{
  "filename": "CL5.json",
  "version": "0.1",
  "user": "a5e135c",
  "mixes":
    {
      "1": {
        "Name": "Bass",
        "Level": -32768,
        "Pan": 35,
```

Fig 18. Scenario 1: API form for setSingleYamahaProfile, where mixes contains mix object

<u>Test Case #:</u> 4	<u>Test Case Name:</u> Setting Mixes React
<u>System:</u> Web	<u>Subsystem:</u> React Native and Flask API
<u>Test Case</u> <u>designed by:</u> Kris Melgar Morales	<u>Design Date:</u> 4/22/2023
<u>Short</u> <u>description:</u>	Native app sends data to API paths, and displays and downloads data on device.

Pre – conditions:

React Native web server and Flask API are running on the containers, and are accessible to the client either by IPV4 or localhost.

Step	Action	Expected system response
1	Enter website from localhost:19006 or 192.168.##(based on ipv4 of host os hosting docker containers)	React Native Application opens in browser
2	Press “Switch to Load”	Form Switches to load page.

3	Enter details into form for loading mix to console HOST:127.0.0.1 PORT:5001 MIX:1 CHANNEL:72	Form visually updates when typing parameters
4	Open Developer Console at start recording Network traffic	No expected response
5	Press the 'Select File' button to load a JSON file	Table fills with JSON data, and the submit button becomes pressable.
6	Submit details using button	200 Response from Flask API, with JSON data response to indicate load completed

Test Scripts

Scenario 1:

Tester	Steps	Expected Results	Pass/Fail
Kris	1	Website loads	PASS
Kris	2	Page switches	PASS
Kris	3	Form Updates with details	PASS
Kris	4	N/A	N/A
Kris	5	File is displayed to view, submit becomes pressable.	PASS
Kris	6	Data is loaded to the console. Response indicates data was loaded.	PASS

11. References

Burges, C. J. C. (1998). Tutorial on Support Vector Machines for Pattern Recognition. Kluwer Academic Publishers.

Chen, P., Fan, R., & Lin, C. (2006). A study on SMO-type decomposition methods for support vector machines. IEEE Transactions on Neural Networks.

React Native. (n.d.). In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/React_Native

About npm. (2022, November 22). Retrieved from <https://docs.npmjs.com/about-npm>

Schumann, David (n.d.). Mixing Station: Remote control digital mixers. dev-core. Retrieved from <https://mixingstation.app>

Yamaha Pro Audio (n.d.). RIVAGE PM Series OSC Specifications. Retrieved from https://cz.yamaha.com/files/download/other_assets/5/1407565/RIVAGE_PM_osc_specs_v102_en.pdf

btendrich (n.d.), Yamaha OSC Translator. Retrieved from https://github.com/btendrich/yamaha_osc_translator

12.Summary and Conclusion

The problem that the team was trying to solve was for musicians to be able save the current state of their mix (level adjustments, panning, on/off states) and be able to transfer that data from one console to another. With our current iteration of the project, the application is able to successfully store the values of any mix number and recall that data for later usage. Upon further investigation, we created tools that would enable us to line up data not by their channel number but by the console's labeling system. Additionally, the structure of our JSON data structure is not limited to the manufacturer but can be manipulated to work with the development of other emerging professional audio consoles.

In order to create an environment suitable for any unique solution required in a live production environment, utilizing a Docker container for scalability and flexibility was a must. Additionally, having this application available on any device was another must and was achieved by using React Native. Our application can easily be integrated with any unique scenario found in any production venue and is robust enough to handle as many clients as the venue may need.

We faced many challenges throughout the development of this application yet have learned many valuable things about primitive web socket programming, frontend & backend interaction, and merging branches between multiple developers parallel programming timelines. Every group member had, previous to this project, little knowledge of all of these challenges, however this application strengthened our skills to confidently be able to address any of these in the future with ease.

Appendix I: Installation Guide

Pre Build Instructions

To install our application first it must be pulled from the Github repository located at, <https://github.com/ChrisMelgarMorales/CapstoneProject>. It is recommended to save the repository to the local device beforehand, instead of attempting to pull it using the docker dev environment option to pull from an https link, as we have not tested this interaction.

Our project requires Docker Desktop and the CLI tools, which can be installed at <https://www.docker.com/>. Throughout the development process of this application, all members were using Docker Desktop version number 4. If the developer was to work on this while using macOS Ventura 13.2 or higher, they would need to disable “Airplay Receiver” from the OS settings in order for the backend and frontend to interact with each other.

A known bug occurs due to CRLF vs LF which causes the both Docker containers to fail to boot as the files are downloaded on windows with CRLF line endings. The bug can be mitigated by going into the root repository, and looking for `CapstoneProject/backend/flask-python/bootscrip.sh` and `CapstoneProject/frontend/react-native/bootscrip.sh` and manually switching the line endings, such as with an IDE like VSCode , to LF. These line endings should be changed before building the containers, as they will not execute after building if the line endings remain unchanged.

Building the Container Method A: buildscript.bat on Windows

To create a build of the containers the tools from Docker are needed, so that the file called `buildscript.bat` can be executed in windows. It contains only one command, `docker-compose -f compose-dev.yaml up`, which will build the containers based on the Dockerfiles in the frontend and backend folder of the repository, as well `compose-dev.yaml` in the root of the repository.

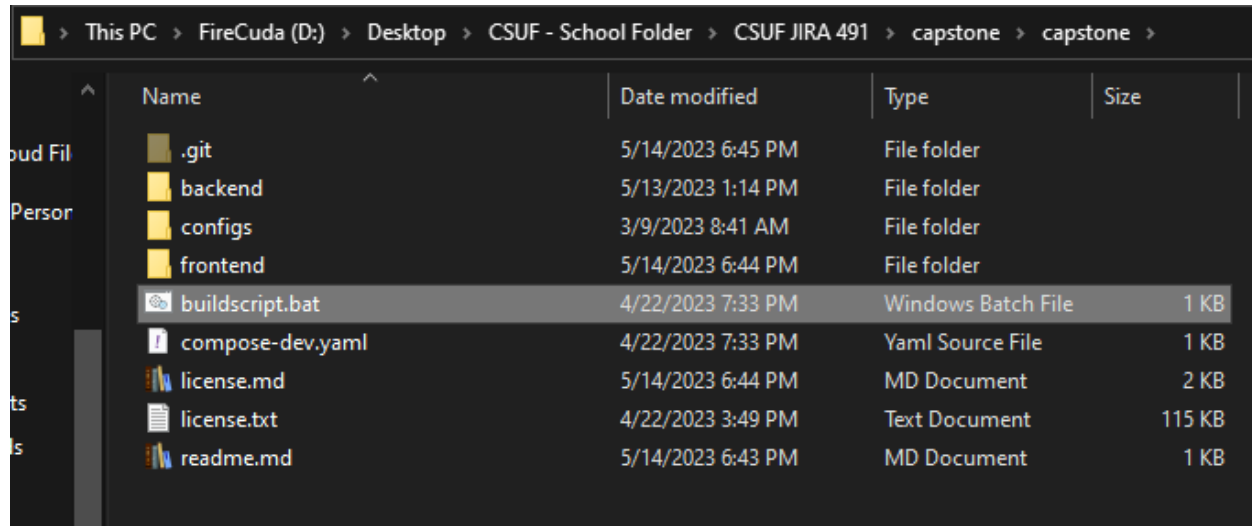


Fig 19. buildscript for windows located in root directory

After the containers are built they can be viewed in Docker Desktop and should be running after building.

Building the Container Method B: Docker Desktop Dev Environment

1. Open up docker desktop and enter the Dev Environment tab and create a new environment.

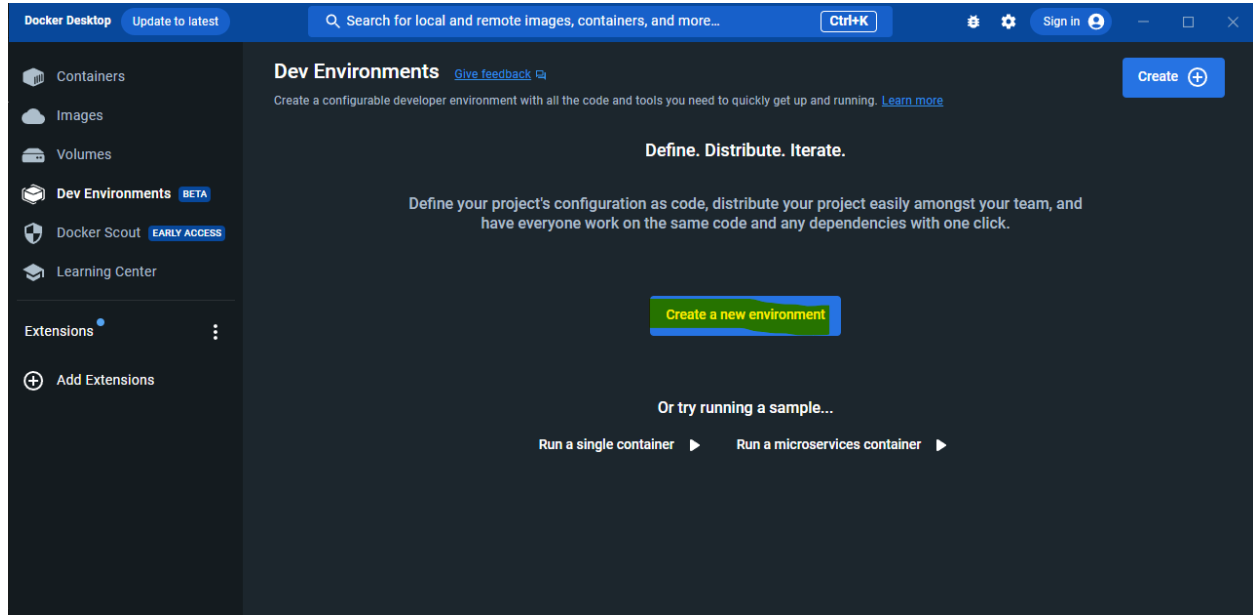


Fig 20. Docker Desktop view when first attempting to build environment

2. Press the Get Started and button to enter the Setup menu

3. Switch to a local directory and add the path of the capstone project repository on your local device and press Continue

Create a Dev Environment

Overview — 2 Setup — 3 Preparing — 4 Ready

Name

A random name is generated if you do not provide one.

Choose source

☐ Existing Git repo ☒ Local directory

Select

The Git repository is cloned to a local directory and attaches to your containers as a bind mount. This shares the directory from your computer to the container, and allows you to develop using any local editor or IDE.

Back **Close** **Continue**

Fig 21. Setup Menu for the Docker Dev Environment

4. Docker Desktop will now build the environment for the React Native site.

Appendix II: Operational Manual

Entering the Application

To enter the application, go into the browser and type in localhost:19006 or 127.0.0.1:19006

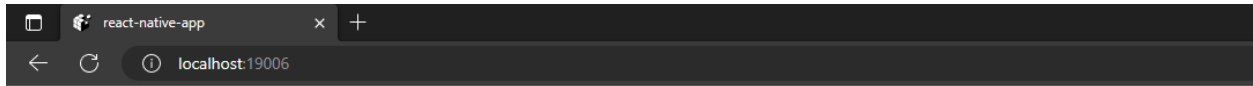


Fig 22. Web link for application

The web application should open up to a page like the figure ready for sending commands to the console.

The application was designed with the Yamaha CL5 which contains around 24 mixes and 72 channels. Unfortunately we weren't able to test our application with an actual console, so for now the application only interfaces with the emulator that we had set up to have data for 1 mix and 72 channels.

Saving Data

To view the data from the emulator from the React application, the following parameters should be entered

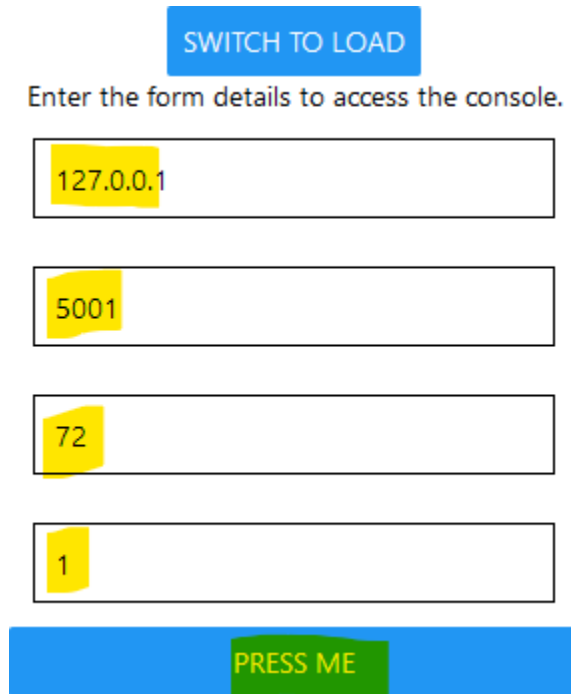
IP Address:127.0.0.1 (The emulator runs on localhost, so that is the ip that should be connected to)

Port:5001 or 5002 (This is the port of the console, when using the emulator, the api uses this field to host the Yamaha CL5 emulator at this port on localhost)

Channel:1 - 72 (This is the expected max amount of channels for the console communicating with, the Emulated CL5 has 72 channels and that is what we tested with, but should work with any number between 1 and 72)

Mix:1 (This is the mix to pull from the console, the emulator only has one mix in its data, so it should be set to 1 for now)

After entering the parameters, the Press Me button can be clicked and the respective data from the console should be downloaded, and downloadable by clicking the Save Button



A screenshot of a web form. At the top is a blue button labeled "SWITCH TO LOAD". Below it is the text "Enter the form details to access the console." followed by four input fields. The first field contains "127.0.0.1", the second "5001", the third "72", and the fourth "1". Each input field has a yellow highlight on its left side. At the bottom is a blue bar containing a green button labeled "PRESS ME".

Parameter	Value
IP Address	127.0.0.1
Port	5001
Timeout	72
Count	1

Fig 23. Parameters entered into save prompt for console data

SWITCH TO LOAD

Enter the form details to access the console.

127.0.0.1

5001

72


1

PRESS ME

CL5.json

SAVE

Channel	Device	Level	Pan	On
1	Bass	-32768	35	false
2	Guitar	10	35	true
3	Piano	100	35	false
4	Mic	50	35	false
5	2	25	35	false

 data(59).json
Completed — 6.8 KB

Show all downloads

Fig 24. Data downloaded from console, shown to screen, and downloadable using save button

Loading Data

To load data to the emulator from the React application, first switch to the load page by pressing the ‘Switch To Load’ button,

SWITCH TO LOAD

Enter the form details to access the console.

Enter IP Address

Enter Port

Enter Channel

Enter Mix

PRESS ME

No mix selected

SAVE

Channel	Device	Level	Pan	On
No data available				

Fig 25. Button to switch page to load on React Native application

After entering the load page, the following parameters should be entered into the React Native Application,

IP Address:127.0.0.1 (The emulator runs on localhost, so that is the ip that should be connected to).

Port:5001 or 5002 (This is the port of the console, when using the emulator, the api uses this field to host the Yamaha CL5 emulator at this port on localhost).

Channel:1 - 72 (This is the expected max amount of channels for the console communicating with, the Emulated CL5 has 72 channels and that is what we tested with, but should work with any number between 1 and 72).

Mix:1 (This is the mix to pull from the console, the emulator only has one mix in its data, so it should be set to 1 for now).

After entering the parameters, the Plus button should be clicked so that the data can be loaded to the React Native Application from the local operating systems file explorer.

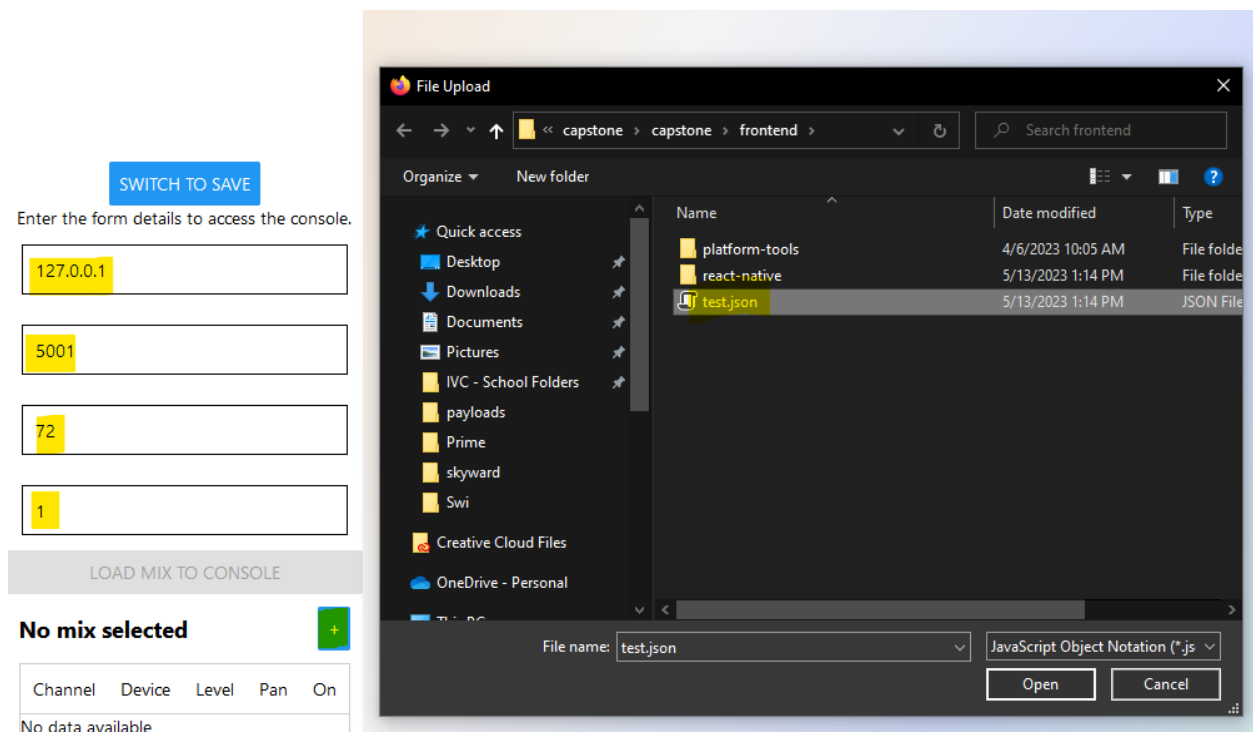


Fig 26. Parameters Entered and File Selected in File Explorer to Load

SWITCH TO SAVE

Enter the form details to access the console.

127.0.0.1

5001

72

1

LOAD MIX TO CONSOLE

test.json

+

Channel	Device	Level	Pan	On
1	zero	0	0	false
2	Guitar	10	35	true
3	Piano	100	35	false

Fig 27. Data loaded to React Application, ready to load mix to console on button press

Then, the data can be loaded properly to the console. To see the changes to the console data, the data can be pulled from the console using the Save Page, or by looking for `/CapstoneProject/backend/flask-python/testserver/dummy.json`, since that is the file the emulator console uses as its profile when its spun up by the API.

Appendix II: Licensing

This project was licensed under the open source MIT License as described in `license.md` in the github repository of our code at `CapstoneProject/license.md` and below,

Copyright 2023 WeDev Capstone Project Team

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Additional License from the dependencies can be found at `CapstoneProject/license.txt`

Appendix IV: Jira, Confluence and Github

Jira: <https://491-wedev.atlassian.net/jira/software/projects/AUDIO/boards/1/roadmap>

Confluence: <https://491-wedev.atlassian.net/wiki/spaces/SD>

Github: <https://github.com/ChrisMelgarMorales/CapstoneProject>