Contents

Abstract

The purpose of “bandCloud” is to allow bands that are touring on the road, to record on the road. The motivation for developing the application came from bands like Bob Seger with Turn the Page, Megadeth with Killing Road and various other artists who use touring as a source of inspiration. While applications like “Band Camp” or “Sound Cloud” are good solutions for sharing audio, they are limited in their audio signal processing capabilities. Inversely, applications like “Audacity” or “Riffworks” are valuable tools for audio signal processing, but they do not offer data sharing capabilities or manage end-user data for them. With “bandCloud” however, users are offered both audio signal processing capabilities via an angular web application, along with persistent and reliable access to their data through hosting the service in a custom Amazon Web Services cloud environment using Terraform. Wherein database solutions were implemented for end-user account management, and storage solutions were implemented for end-user audio data. From a security perspective, a backend application gateway was deployed into a private subnet, so that end-user requests for backend resources stored in database, storage can be authenticated. To work towards achieving high availability of the service, the docker images of the frontend angular app, and backend spring REST API applications were deployed separately behind load balancers in replicate subnets. In registering the public “bandcloudapp.com” domain and acquiring a TLS/SSL certificate, the “bandCloud” service offers it end-users a secure context to run the application over the internet on different device sizes.

Acknowledgements

I would like to extend an enormous thank you to my lectures & classmates for providing a course that in hindsight, I would gladly pay more for & got far more out of than a simple acknowledgements section can convey. On a more personal note, I want to acknowledge my friends and family (near & far) because without them I would be lost.

Chapter 1: Introduction

General Aims of bandCloud

The rationale of “bandCloud” is that bands touring can record their next album/concept piece “on the road” with minimal technological effort. Quite often hit records like “Bob Seger - Turn the Page” or “Megadeth - Killing Road” compare aspects of life to touring. Thereby highlighting touring as a source of inspiration for a variety of artists. However, there is currently an awkwardness between their ability to produce and capture ideas in their raw form within this setting. While free-to-use technologies like “Audacity” are quite verbose in the audio signal processing capabilities they provide their users, “getting started” is quite a procedure. Users of desktop applications like Audacity will need an electrical instrument, an audio interface to pass data from their instrument into the app, turn on their computer, launch the app, etc, outside of the data management considerations. On the other hand, there are apps like SoundCloud/BandCamp that exist for sharing audio, however, they don’t provide their users the capacity to both record & mix “on the road”. With “bandCloud” users are offered both, and need only to launch an app on their phone or other device and tap record after registering. Users are provided the ability to share/playback this audio feed to fellow band members and add a track from their “axe” to this next hit record. These separate audio tracks could then be mixed into a demo that they could share on their social media account(s). Additionally, such an application could be developed with subscriptions in mind, which include features such as expanding audio processing capabilities & storage limits that can be provisioned based on a changeable account type.

Project Planning

For “bandCloud” to offer the services highlighted in the introduction, developing the application was seen as a “full-stack” project. Where end-users will interact with a web application, which requests user-specific data from a backend REST API service that interacts with the cloud services where their data is stored. The rationale of partitioning the project into the three zones of “*Frontend*”, “*Backend*” & “*Deployment*”, was to allow each part to be developed in a focused & independent manner. Owing to this and how it was a solo project, an iterative & incremental development approach was adopted where the output from each phase of the project informs the planning and development of the next within their timeline (Fig 1-4). So that the developed prototype achieved the essential features presented discussed in “*Chapter 3: System Analysis and Design*” and “*Chapter 4: Implementation of the System*”.

Project Scope

As per the overview and introduction, each zone of the application was viewed as essential and is expanded in the “*Chapter 3: System Analysis and Design*” section. While minimalistically, this means a web application where users can register & access their data (i.e read/write audio). The project emphasised frontend features that will enable end-users to record tracks, mix tracks, and save their projects. Backend features that facilitate these data exchange processes, such as persistence of end-user data or client authentication. In combination with a cloud-based infrastructure that delivers the “bandCloud” service to end-users. The level at which these three zones were designed was carefully considered for the project deadline of 13/08/2022, and as such, the project was planned based on the phases discussed in the chapter 4 which were limited to the time blocks of Fig 2. In short, each phase had its own high- and low-level deliverables wherein the requirements of chapter 3 were distributed. Where high-level deliverables were deemed priorities for the corresponding phase, and low-level deliverables were either noted for the next phase or “worked around” for that phase/period of focus where possible. Such an approach allowed development hurdles like migrating the REST API to a private network to be considered in more detail in the next phase/iteration.

In a similar light, the project was grounded in what it won’t deliver, where some example areas of development that are viewed “out-of-scope” for the project included; Deploying the application using microservice architecture aided by services like AWS’s Lambda Functions, deploying applications into manageable container clusters like AWS’s Fargate, background jobs that moderate the platform (ie analyse audio data). However,

discussing the potential value that they could bring may not be, as they can facilitate the planning of future “bandCloud” versions beyond this project. The decisions here was primarily made based on the focus on time management, but also because it is expected that the project will offer an opportunity to acquire more knowledge in the broad areas of networking, DevOps & audio signal processing. It should also allow sufficient time to engage with the technologies to be used and the context for their development (i.e the value of single-page websites etc) which is the primary personal aim of the project.

Iterative & Incremental Development Phases

As per the “*Project Planning*” section in combination with the scoping considerations of the project, an iterative and incremental development approach was adopted for the project, whose results are discussed in “*Chapter 4: Implementation of the System*”. The development methodology was adopted because it emphasizes an informed development life-cycle, wherein the results from each phase/iteration guide the development of the next organically (Larman and Basili, 2003). Specifically for this project, the development of the submitted artifact followed three phases. Where the goal of Phase-I was to research each of the three zones of the project, i.e basic audio signal processing, cloud computing, and REST API development in isolation. The goal of Phase-II was to tie these elements together into a functional application that includes the core three zones of the project, which can be built upon where necessary for the final prototype in Phase-III. Which saw the implementation of a public cloud deployment of the system, development of classes which aided in managing the audio data on the workbench, and delivering the service in a secure context (ie “*https://bandcloudapp.com*”).

Chapter 2: Literature Review

Audio Signal Processing

While the research of this field for the project was intentionally geared to minimalistic, it did offer space to explore the field. In particular, the lectures & labs from the Coursera course (https://www.coursera.org/learn/audio-signal-processing) offered a general overview of how adopting & better utilizing the Web Audio API could be accomplished given broad knowledge about how audio data is represented on computers (Docs, 2022b). In particular, how programming skills acquired from the higher diploma course could allow me to see that on a basic level an audio signal is an array of numbers, and be confident at manipulating & creating them. But also acknowledge why standard objects for the field (i.e Pulse-Code Modulation, PCM) exist. In that, they encapsulate these data into entities with standardized methods & attributes that support their use (Fig-5). More broadly, lab demonstrations from the course were useful for gaining insight into why sinusoidal modelling is valuable for the field. Where a one-dimensional sine-wave can be represented as a three-dimensional wave prorogation in a simpler form. Such a strategy has allowed ASP engineers to develop modelling strategies for identifying fundamental frequencies, which would practically offer methods to drop background noise from an audio signal. Additionally, sinusoidal modelling strategies in the course also touched on advanced audio effects like instrument isolation and chorus (common “progressive rock” effect). Both of which could be valid features for future versions of the “bandCloud” system.

Technologies Used

The core technologies used were “*Web Audio API*”, the “*Angular Audio Context*” module that adds types to this API so that it can be used with the Angular framework, and the “*Recording-RTC*” module which does the same for “navigation get user media”. Initially, “*Pizzicato.js*” was reviewed because of its grouping features & additional audio effects. But was not adopted because the personal aim of the project was to explore how such methods could be developed. Additionally adding effects like gain & distortion to the “bandCloud” prototype was seen as valuable, not essential.

Modern Web Development

Owing to the growth in device capabilities, there is an emphasis on responsive web design & applications that can run on multiple devices and are simple to use. The frameworks that support their development also represent a natural progression in the field. For instance, the jQuery & Boostrap-CSS frameworks introduced a simplified way for users to design responsive webpages and utilize commonly used data structures in those pages such as accordions, carousels, that are now common place within the Angular framework (Hajian, 2019). Frameworks like Angular also built on these developments by offering an interface for modularizing the design plans of single-page web applications, ex “ng generate component | class | enum | service”, introducing so called “Backend for a Frontend” design patterns. In single page web application frameworks like Angular, components of an application can be navigated without requiring requests for rendering additional components. The field has also evolved with progressive web application (PWA) solutions that also enable the development of cross-platform applications with the "look and feel" of native apps. While not fully explored in this project, such systems are usually designed with separate applications for mobile & desktop traffic. Allowing systems to offer not only offline features, but their respective UIs to be developed separately and optimized for user experience with the device size. Which for this project became quite apparent as the workbench of “bandCloud” could benefit from the desktop application looking one way & the mobile another.

Technologies Used

The core technologies used for the frontend application were Angular for developing a single-page web application, Node.js for serving the deployment and application, then “*Express.js*” in conjunction with “*Axios.js*” so that application can proxy requests to backend resources. Angular was chosen because web applications can be built as single pages from the angle of efficiency as the static content need only be fetched once. In addition to modules like “Angular Material Design” which offer simplified way of incorporating user-interface components into the application. Alongside, Angular design scheme which provides an interface to aid in planning web applications through a more modularized manner (ex “ng generate component | class | enum | service”). The Node.js, Express.js, and Axios.js technologies were chosen for the web server because collectively simplified designing & implementing a webserver. Additionally, Express.js and Axios.js together simplified developing the logic for endpoints on the webserver listening for end-user requests that need to be proxied to the backend RESTful API service.

REST API Development

APIs are a useful technology because they simplify client-server message exchange allowing dynamic web pages to be developed and are cloud services are delivered (James F. Kurose, 2021c). They simplify this because the message exchange is a representation of either a server or client-side object, wherein either side can be developed to specialize logic for handling that representation. Where a client sending a request to a server-side API isn't aware of the specifics of how the response is generated. Meaning that they can be altered independently, allowing frontend & backend apps to be designed in a reasonably isolated manner. They also offer a means of designing this client-server model with a regulatory layer specifically designed for validating client requests to ultimately backend resources (storage, DB, web content, etc). For this project, both points were utilized in developing the current “bandCloud” system.

Technologies Used

Java & Spring were chosen because of the experience accumulated during my coursework at CCT (Baeldung, no date; Gonalez, 2017; Sciore, 2019). Where was chosen Spring because it allowed the development arc to focus on designing the backend as a resource that supports frontend and not focus on the logic of socket programming. The Gradle build tool was chosen because of its simplicity & intuitive dependency management which was particularly useful as Spring and the AWS-SDK were the main dependencies, and the AWS-SDK required installing additional packages.

Cloud Computing

With roots in high performance cluster computing and grid computing, Cloud has had an enormous impact on SMEs (Small and Medium Enterprises), large companies, and research communities (Fernández-Del-Castillo, Scardaci and García, 2015; NHLBI, 2016; Kamal Kant Hiran, Ruchi Doshi, Dr. Temitayo Fagbola, 2019b, 2019a). Where for Ireland, the Central Statistics Office estimates that roughly 6 in 10 companies adopt cloud computing services, where, as shown in figure 6, emails, office software, and storage represent 40-47% of use cases (CSO, 2021). The impact that cloud vendors such as IBM, AWS, Google, Alibaba, SURF, etc have had is by offering IT infrastructure as a service, instead of each entity having to invest in and maintain their own infrastructure (ie storage, compute servers, networking devices, etc). Where the cloud vendors themselves develop and maintain their own data centres to meet the demands of their clients (businesses), hardware solutions (to facilitate bigger and bigger data), or software solutions (to better expose using their platform ex AWS Batch / Aurora RDS). So that diverse businesses such as Zoom, and EA Games can host their applications on virtual machines running on these systems, enabling thousands of meetings to be simultaneously hosted each with their set participants, or thousands of servers hosting online games that gamers can join (Google, no date).

Technologies Used

AWS was the chosen cloud vendor because of prior experience in their database, storage & compute services, which made exploring the concepts of load balancing & IaaC more feasible. Terraform was chosen as the IaaC based on personal preference because it offered the simplest IaaC interface (Brikman, 2019). Which allowed me to get started quicker than with Ansible because there appeared to be less involved. The Linux OS was chosen for the Elastic Cloud Compute (EC2) Virual Machines (VM) because scripting is simpler than in Windows. Docker was chosen because it helped to automate application deployment. DynamoDB was the chosen database technology because its provisioning strategy provides a convenient database solution for user data lookups & scalable persistence. A NoSQL (No Structured Query Language) solution was preferred an SQL solution because object attributes can be stored in a single item that is queried on a user-specific level, and those data can be used to reconstruct the same/tailored version of the same object. Meaning that the access pattern from the client-server is one request, one item, without the need to join operations. While AWS offers relation database solutions, their deployment has more overhead in terms of provisioning, and currently, the requirement for join operations in “bandCloud” is non-existent.

Simple Storage Service (S3) was the chosen storage solution because while it's an object store and not a file store, bigger objects like audio data, and compressed archives of deployment apps, can be stored under prefixes delimited by "/" as if it was a file system. Additionally, very little is required to configure limitless storage and additional relevant functionality like object metadata for setting object content types or methods for generating pre-signed URLs allowing objects within a private bucket to share as if they were public with appropriate read/write permissions.

Chapter 3: System Analysis & Design

Overview

The following chapter discusses how the system will be designed to the deliver aims of the project brief. The chapter explores this by discussing the functional & data requirements and their design separately. Where the "*Functionality*" section discusses the desired and essential requirements of the system, to convey how adopting the "*Iterative and Implement*" approach enabled prioritizing the tasks for each phase of the project for each of the three system components: *Frontend Web Application. Backend REST API* and *Deployment*. The focus of the system data section is on how the system will store the relevant data.

Functionality: Requirements & Design

Overview

As per "*Chapter 1*", the purpose of "bandCloud" is to deliver end-users audio signal processing capabilities over the internet in the form of a web application. With a view that the persistence of these data is vital, the project was divided into three zones each with their own functional requirements and design. A frontend was designed as the user interface where users can (de)register, login, view/edit their account data, and be presented with a workbench to manage their audio data (ie recording, mixing, playing audio). A RESTful API acting as an application gateway was designed to enable & regulate end-user requests to their data such as system credentials & audio projects (James F. Kurose, 2021b). Finally, the system was hosted and deployed on the AWS cloud so that the application can be accessed over the internet and end-user data can be stored in scalable database & storage solutions.

User Interface: Frontend Web Application

The frontend is responsible for communicating with the backend RESTful API service for retrieving/updating the end-users' data. With this application, users can (de)register/login to their account and also makes use of cookies to provide a session-like authentication because HTTP requests are independent. While they could be given options to also view & edit their account information (username, email, password, etc), it is not essential for the application, but it does realistically follow that a user should be able to edit these data as needed. The application will be composed of components that communicate with their respective controller on the backed and will make use of classes to construct backend responses as objects (list of project metadata, project metadata, account details). This is essential because it is a full-stack

project, with a design approach towards single responsibility. Essential audio tasks have been viewed as providing the user with the capability to read, write & edit audio data. Reading will primarily take the form of accessing the audio data from all of their “BandCloud” projects, while being able to import audio data from the user’s device would be useful it is not central because the user’s microphone can be used to generate audio data. Since the process of “mixing” edits audio data (ie add one audio signal to another), it is essential that “bandCloud” provides this feature because it is in the project's name. Additional edits of audio signals such as gain and distortion, while valuable, are not essential for the application.

Backend REST API

The backend RESTful API will be responsible for the logic of servicing end-user requests for authorizing users to the system, authenticating the requests made to it, and the associated CRUD operations on end-user data if valid (Fig 7-9, Table 2-3). The application will be composed of at least two controllers: Project and Accounts, which will store user-specific project data in their directory on an S3 bucket and account information in DynamoDB. These actions will be supported by appropriate models which can be displayed in the client’s view. The accounts controller is responsible for the persistent storage of end-users login details in a database (Table-2). The database used is AWS’s NoSQL DynamoDB, which uses a “Users” table for storing the end-users: user ID, username, email address, session expiration date & key, hashed password, and salt for a password so that the user authentication process is more secure (“*System Data: Requirements & Design*”, Fig 7-9). While a “View” table could be used to store additional information about the user such as the number of projects, and how much storage space they are using, these data can be queried from S3 for the project controller. The “Projects” controller is responsible for sending & fetching the user's project data which uses an “s3://bandcloud/data/{userID}/{projectName}” hierarchy (Table 3). Within a project, the raw and mixed audio data are stored in separate directories. The controller makes use of S3 object tags to simplify how the end-user interacts with the projects through what is effectively a file explorer. All endpoints parse the user's cookie to authenticate end-user requests without having to explicitly maintain a session.

Application Deployment

The following considers the deployment of the application AWS cloud. Since the above sections discussed relevant tables & storage hierarchy, the focus of this section is the essential requirements for the hybrid cloud deployment. Where, as per Fig 3-4, each application component is hosted behind a load balancer that is responsible for distributing traffic across their respective availability zone subnets. A virtual private cloud is essential for hosting the application on the AWS cloud because it is where resources such as web servers can be deployed into. Creating a VPC has two major components a region such as Ireland/eu-west-1 which is a datacentre, and an IP address which determines the size of the VPC. A class B IPv4 address was chosen because it will limit subnet sizes to <254 hosts while the system is being designed with 192.168.0.0./16. For simplicity, IPv4 was used and separate subnets were created for the frontend & backend application, to separate API and web application traffic. Each application was deployed in a replicate /24 subnet that resides in separate availability zones so that high availability of the "bandCloud" application can be achieved and automatically applied using a terraform object variable (192.168.1.0/24, 192.168.2.0/24, 192.168.3.0/24, 192.168.4.0/24). The compute configuration will make use of launch configurations and auto-scaling groups that work with their elastic load balancer, so that resource provisioning is automated which is intuitively essential. Both the frontend and backend applications will be deployed into EC2 instances which run the appropriate web server in docker containers that are fetched from their ECR on instance start-up through a user-data script. While container-based deployment is not essential per se, it better automates application deployment and avoids lengthy user-data scripts which is a preferred development approach personally. It is essential that both the VPC configuration & the application deployment behind load balancers work towards a modularized approach for this project because this code is used to deploy the application that could in theory require a high degree of "*scaling out*", so updating this needs to be simple. Identity access management roles are essential for deployment and will be created using the AWS console where appropriate policies will be applied so that the backend instances can read/write to DynamoDB & S3, Read ECR, and the frontend can read ECR, but cannot read/write to DynamoDB, etc. While initially, this would be debugging, the long-term view is one towards "*least privileges*" for security reasons but not essential for this project as it will not yield a production ready system.

System Data: Requirements & Design

As discussed in the previous section *“Backend REST API: Design & Requirements”* the essential data to be stored are end-users account data, and audio projects. The persistent storage of end-user accounts is essential and the system should also be designed to validate the authorization of the related processes. End-users accounts are stored in a database so that the data for a given user can be queried fast because of the implementation of indexes on the userID and username fields (Fig 8-9). Owing to the need for the system to only look up a given end-user by their userID from a session or query a username (validate login data, availability on registration), and no immediate join requirements a NoSQL solution was adopted. DynamoDB was the specific technology adopted here because of the ease of configuring reading & writing auto-scaling of its indexes. The persistent storage of end-users audio data is also essential for the system because it is the purpose of the application. End-users audio data are stored in a ”bandCloud” S3 bucket, under the hierarchy of “data {userID} {projectName}”, which separates audio projects by userIDs. S3 was chosen as the storage solution because it is an auto-scaling storage solution managed by AWS, and also provides the

Chapter 4: Implementation of the System

Overview

The following chapter describes how each of the project's three phases contributed to the development of the system based on both "*Chapter 3: System Analysis & Design*". The focus is on how the submitted artifact grew over the course of the project and how some of the main obstacles encountered during the project were overcome.

Phase-I

Phase-I was the heaviest research-orientated phase and considered the zones of the project in isolation. The aims of this phase are geared towards identifying feasible deliverables that can be built upon. For deployment, this took the form of a simple login app that was deployed behind a load balancer on AWS and storing login details in a database. For the frontend, only the basics of web-based audio signal processing basic were considered such as reading/writing, editing & mixing which were aided by the documentation of "MDN Web Audio". Finally, REST API development was explored during coursework, with an eye for features such as securing login processes & persistent authentication throughout an end-user's use of the application. The specific technologies used for this project were decided following the completion of Phase-I and are discussed in "*Chapter 2: Literature Review*".

Phase-II: Primitive Artefact

Phase-II developed the output from Phase-I for all three zones of the project. Where a frontend with functionality allowing users to register, login, edit account data & deregister was developed. Additionally, the frontend also set a skeleton for how the project selector & project workbench could be designed via an initial skeleton so that end-users are offered simple solutions for choosing a project. Central to this phase was the incorporation of core audio functionality for fetching, playing single & multi-track, saving, mixing & recording audio which was adopted from Phase-I. While the logic for encapsulating the audio functionality in a specialized class was not determined in this phase, patterns of repetitive tasks were observed and the exposure to asynchronous programming set a starting point for Phase-III to tackle this problem (ie the Track and Track collection classes).

The backend in this phase was designed to support the persistence of data generated from the frontend application. While initially viewing its development in this phase as skeletal, before refactoring AWS-SDK specific methods, their refactoring was completed during this

phase (except for fetching audio projects). The most useful of these “workarounds” was using a singleton array list of users with methods for login validation, and endpoints that retrieve test data using a hierarchy that emulates the final design so that the frontend could progress towards audio functionality, without the backend being a complete bottleneck. Where such packages were updated with methods that interact with DynamoDB, or a user’s specific project data after the audio functionality was mapped because the core backend response objects do not change. The rationale here was that the progress made by the frontend, backend, and deployment can be as close to independent as possible. Finally, the deliverable from the deployment zone was Terraform modules that create the infrastructure required for deploying the applications and then deploying them. These modules were then further developed in Phase-III to achieve the hybrid deployment model, where the frontend application is accessible over HTTP(s) via the “bandcloudapp.com” domain, and the backend API is only accessible to internal resources via the “resource.bandcloud.com” domain.

Phase-III: Submitted Artefact

Frontend Track and Track Collection Classes

The primary focus of Phase III was simplifying how the frontend audio workbench can not only carry a project's metadata but also manage to translate these metadata into audio signals and provide end-users functionality to interact with them. Solving these problems was benefited by developing a Track class that can be instantiated from an audio metadata object. The object can then make use of methods for fetching the related audio file, setting its audio buffer property, and updating its state (ie summary statistics of the audio signal, whether or not the audio is playing), so that the development of the workbench can focus on providing buttons, etc to the end-user for using these methods. However, since a project is realistically a list of audio tracks, a Tracks class was developed for managing a collection of Track objects. Such methods include adding/dropping a reference to a track, mixing each track of the collection into a new “mixed” track object, sorting tracks by ascending size, as well as playing tracks in/out of sync. The appreciation of this logic enabled the development of generic methods for the workbench that can handle adding, dropping, playing and mixing track objects from a track collection objection. Additionally, it also allows the workbench to be developed with multiple instances of track collections. The feature is useful for “bandCloud” because it allows end-users to record new audio signals into an object, that is distinct from a desired form. Long-term this could enable the workbench to reload an active project, record new audio signals, and then present the end-user a means to select their “best” recordings to add into the project’s master mix.

Deploying Backend Application into a Private Network

The secondary focus of Phase-II was exploring how feasible the deployment of “bandCloud” as per Fig-4 could be for the project (James F. Kurose, 2021a, 2021b). Though its priorities compared to other elements of the could be argued against frontend stylings and more work on the backend REST API. Its exploration was a personal preference and one that more immediately helps my career while still being a valuable contribution to the project as a whole. The key components here was removing the route to the VPCs internet gateway, and not assigning the VMs with a public IP address. Meaning, that there is no route for internet traffic to reach the associated subnets. The second component was editing the firewall rules on the network level via network access control list (NACL), and the EC2 service level via security groups. The updates were essentially to block any non-internal traffic to the private subnets and the VMs. However, the opening of ephemeral ports (#1024-65535) was required because the subnets will be associated with a network address translation gateway that AWS manages and the range also covers the ports required for Amazon’s Linux image that the EC2 instances use (https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html#nacl-ephemeral-ports). The final component was configuring the backend servers with access to internet, which was achieved by setting up a NAT gateway in a public subnet, applying a route to this gateway on the private subnets in addition, applying the aforementioned NACL rule but also additional rules allowing outbound HTTPs traffic (ie TCP, port 443) to respective firewalls. With the exception of the backend EC2 VM’s security group because they are stateful, meaning that if inbound traffic is allowed then the response is assumed.

Configuring bandCloud Domain & HTTPs Connections

Domain management using AWS's Route-53 service (Domain Name System uses port 53 was explored for "bandCloud" because both public & private networks exist within the VPC (James F. Kurose, 2021a, 2021b, 2021c). Additionally, the process of proxying requests could benefit from using a set value "resource.bandcloud.com" that can be updated. Instead of the domain name of the related load balancer being queried before either the fronted/backend servers are launched. In doing so it was noted that AWS is not only a registrant but they offer TSL/SSL certificates, meaning that the "bandCloud" app could be registered under "bandcloudapp.com" with minimal effort. Additionally, since both the frontend & backend applications are hosted behind load balancing domains canonical name records were added into the registered domain which points to these services, where "web.bandcloudapp.com" resolves to the internet-facing load balancer which distributes traffic over the public subnets into which the frontend application is administered. Then the "*api.bandcloudapp.com*" canonical name record was added which resolves to the internal application load balancer domain which distributes traffic over the private subnets into which the backend RESTful API service is administered. Following registration of the "bandcloudapp.com" domain, an SSL/TLS certificate was requested and is currently used by the service so that end-users can establish secure connections to the "bandCloud" system that is hosted on AWS cloud.

Proxying Client-Side Request for Backed Resources

While the primary aims of this phase were those discussed in the above sections, their implementation meant that developing the web server with this capability could be revisited (James F. Kurose, 2021d). The core problem areas to revisit were how the process handles bandCloud-specific objects represented as JSON and how this process handles cookies set by the backend server. For the former, this meant configuring the webserver requests with appropriate headers where necessary in addition to converting “bandCloud” objects to JSON strings. For example, the registration endpoint expects a “User” object, so the request that the web server sends to the backend on behalf of a client is as the below screenshot. Where the “*apiForwardingUrl*” is the internal domain of “resource.bandcloud.com”.

```
let data = instance.post(
    `${apiForwardingUrl}/account/manage/register`,
    JSON.stringify(req.body),
    {
        headers: {
            'Content-Type': 'application/json',
            'Proxy-Connection': 'keep-alive'
        }
    }
);
```

The handling of cookies was managed in three ways, one for the “*/account/display/view*”, a separate process for privileged endpoints that expect a client-side cookie, and a final process for returning the session to the client (ie registration and login). For the endpoint responsible for providing end-users with their account information, the request was simply forwarded as is to the appropriate backend endpoint. The commonality between other endpoints is shown in the below screenshot, which required embedding the end-users session into the request to be proxied.

```
let data = instance.post(
    `${apiForwardingUrl + req.url}`,
    JSON.stringify(req.body),
    {
        headers: {
            'Content-Type': 'application/json',
            'Proxy-Connection': 'keep-alive',
            'Cookie': req.headers.cookie
        }
    }
);
```

Finally, for the login & registration endpoints, the session cookie defined by the backend server needed to be parsed and then defined in the response of the proxy request process. The below screenshot shows how this was implemented. In short, the cookie data set by the backend server is parsed via the “cookie-parser.js” module and is iteratively embedded into the response of the proxied request (“resp”). An outstanding issue for the proxying of client-side requests is that webserver is exposed to any user agent (ie cURL, Postman, etc) as opposed to just web browsers. Although not implemented, a solution here is to parse this field from the client-side request and reject such user agents, as the relevant data for this is available under “*res.headers[‘user-agent’]*”.

```
data.then( function(res) {

    // Set response cookie
    console.dir(res.headers['set-cookie']);
    for(let cookieData of res.headers['set-cookie']) {

        // Get cookie key & value
        let cookieKey = cookieData.split(";")[0].split("=")[0];
        let cookieVal = cookieData.split(";")[0].split("=")[1];

        // Create cookie
        resp.cookie(cookieKey, cookieVal, {
            maxAge: (3 * 24 * 60 * 60),
            httpOnly: true,
            path: "/"
        });
    }

    // Send response
    resp.writeHead(200, {'Content-Type': 'application/json'});
    resp.end(JSON.stringify(res.data));
})
```

Avenues for Development

Owing to Phase-III delivering the core logic and flow of “bandCloud” puts the focus of future onto refinement over exploration. The following highlights these areas for each zone of the project.

Fronted Webserver

The main areas for development on the frontend web server related to how projects are handled & the end-users audio is posted to S3. For the submitted artifact these were only touched so that the logic of the workbench could be built. Next steps should at the functionality for end-users to select a project with which to load a workbench, or create a new project with an empty workbench. While the skeleton of how these data are to be presented was developed for the submitted artifact, it can be grown upon using an accordion as per the “Account Management” component. Similarly for the workbench, the end-users should be given a form through which they can name their audio tracks, and also rename their project if they so choose. Additionally, the end-users should also be presented with an audio playback tickbox which dictates whether or not previously recorded tracks should be played back during the recording of a new audio track. Once these areas have being developed, the workbench should then pursue how end-users can update their project on S3. Which should take the form of adding a single raw audio track, update entire project via tickboxes which provides a list of track names from a tracks collection object to push to storage (builds on the current method). Following this a user-guide/FAQ should be considered, so that end-users are informed about how to get the most out of service. For instance, recording tracks from the microphone of their phone with headphones for uncorrupted playback during recording is not intuitive.

Optimizing the posting of audio can be handled by reducing the number of “bandCloud” sever hops it takes for an audio file to be posted to S3, and by compressing the related data. Currently it takes 3 hops for a file to reach S3, one post from the end-user’s device to the frontend web server, the second from this server to the API, and the last from the API to S3. Addressing this requires developing the backend API to return a pre-signed URL for the frontend with a time restriction, ex 20mins (discussed in the below). So that an audio object has 1 direct “bandCloud“ server hop to reach S3. Its development depends on the progress discussed in following “REST API” section. A second optimization can be applied by compressing the related audio blob before posting, and decompressing with the same format for get requests (iex gzip). Both suggestions will require their own specific planning & research in order to implement the features, a starting point for compression of the audio is exploring the “Compression Stream” API or using an MP3 format instead of WAV (Docs, 2022a).

REST API

During testing of the most recent version of the deployed website some bugs were identified when a user is only updating one field from the form on the page. Where cause & solution will merit reviewing the associated endpoint. Currently the logic for the endpoint is that, any query parameter that is true marks that attribute as needed to be updated. Once done, the post & get methods from the project controller should be revisited in a generic way. The current methods provided a means for developing & debugging the process using mock data, which should be lifted over to be based on project names under a userID. While a lower priority than generic methods for posting/getting audio, the validation methods should be updated to return a list of enum values that the frontend can use so that the end-user is more informed about the issue (ex password rule etc).

Once these three areas have been resolved, backend development should focus on supporting the logic of end-users being a band member. While template classes have been written for this, along with a custom list of user's class to aid in the process, the logic for delivering this feature should be given its own time. The reason for this is because of its utility for "bandCloud" as a product but also because if it is inadequately developed, it might as well not be implemented. Once in place, the related file meta data classes can then be developed to make use of the coded but not implemented, pre-signed URL methods. The rational for this is to reduce the number "bandCloud" server hops it takes for an audio file to reach S3 and end-users, but also lay the groundwork for adding a feature to allow bands to share selected audio files.

Deployment

Currently there are two priority areas of development for the deployment of bandCloud. One is debugging the proxy request issues discussed in this chapter's "*Proxying Client-Side Requests for Backend Resource*" section. However, when "bandCloud" is viewed as a product this could instead explore using a more common approach such as those provided the NGIX server technology. The second priority area of development for deployment, is to update the backend REST API to be authenticated on the service level instead of the currently implemented credential-based authentication. The reason is security based, because these credential keys should be updated periodically & service-based authentication also allows the associated identity access management (IAM) role of the servers to be updated using AWS's IAM simpler than on a user/group level.

Chapter 5: Testing & Evaluation

The following discusses the testing & evaluation that was performed on the system. The focus is on how the Postman client was used to test the development of the backend REST API. How deployment was tested through verifiying the built containers before deployment, API calls to the backend resources, and monitoring traffic to the “bandcloudapp.com” domain. Before closing with how testing and evaluation of the user interface were performed.

REST API Tests with Postman REST Client

API calls to the backend service were tested using the Postman REST client, a project was created with branches for each controller and tabs for each respective endpoint. For this project, the focus of testing & evaluation was functionality. To this each endpoint an entry for each endpoint was created in both branches, and requests were created with appropriate parameters and request bodies. The aim here was to evaluate whether or not the system performed as expected, for example, the registration endpoint responding with a session cookie & storing end-user data in the Users DynamoDB table. Tests were set up with genuine user input, input that reaches the account registration rules, and invalid login input. Fig 10-16 show example results from these tests from the file submitted with this project where it was found that the system performed as expected in terms of HTTP response codes, object responses & enum values for the context of error codes. The second layer of tests implemented was viewed as “system algorithm” based, where the aim was to assess whether or not users can access privileged endpoints without a session cookie. To conduct this test each endpoint were sent requests using “*correct*” input, without a cookie. An example of the output is shown in Fig-13, where the system responds with an unauthorized HTTP status code of 401 and context to this error for the web app from the “*Login Validation Types*” enum which shows that the HTTP status was raised because the user did not login to receive “bandCloud” session cookie.

Deployment Tests

The main deployment tests carried out were confirming that both the frontend and the backend applications work as intended in docker containers and were accessible behind their load balancing domains. Since both applications were initially on public networks, both applications could undergo the pre-deployment test as per Fig-16. However, as the project progressed the backend API was pushed into a private network, meaning that the HTTP requests for testing had to be performed within the internal network of the “bandCloud” VPC. One challenge that was encountered in achieving the hybrid deployment, was that while HTTPs traffic was allowed out of the NAT gateway residing on a public subnet, instances in the private network could not install docker (i.e no internet access). The cause was determined to be on the network access control list level, as inbound traffic on ephemeral ports was only allowed internally and not from anywhere (https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html#nacl-ephemeral-ports). Altering the source traffic IP address, enabled the instances to install docker and thus solved the deployment issue.

Confirmation that the backend resources were indeed private was provided by the tests carried out in Fig 17, where ICMP-IPv4 requests from internal & external networks were sent to the private subdomain of the application under “api.bandcloudapp.com”. Where replies could only be received from within the “bandCloud” VPC network. Finally, Wireshark was used to test the states of requests to the web server. Where contrasting the results from Fig 19 in to Fig 18, show that requests to “bandCloudapp.com” are indeed encrypted. Establishing this for the project, required utilizing AWS’s DNS service for the creation of a public domain, requesting an SSL/TLS certificate for this domain, and attaching a HTTPS listener rule to the frontend application load balancer which forwards this traffic to load balancers target group. The Route-53 service was also further explored for adding records for the REST-API’s load balancing domain, creating a private domain called “bandcloud.com” and creating a subdomain “resource.bandcloud.com” that the web server currently uses for the proxying of client-side requests to backend resources (implemented before the test in Fig-17 was performed).

User-Interface Tests

The aim of the user-interface tests was functionality driven because more work is needed to push the submitted artifact from its current form to a state that will benefit most from usability tests. The main tests conducted were to see whether or not when a user registers that they can login to the system, edit their account data, remove their account, that the project workbench works as expected and that the web applications components are indeed lazy-loaded. Excluding the audio workbench & lazy loading, the tests were conducted by using the application in its intended way and then verifying data changes on DynamoDB using the AWS console, or the refresh button on the user account page after submitting the form. In all the cases, the web application performed as expected. However, error handling such as raising alerts to end-users on invalid registration is not currently implemented and should be a priority for the next phases given that the direction that the audio functionality has been addressed with the Audio-Service and Track, Tracks classes.

The audio workbench and lazy loading were tested by viewing the appropriate logs to the console, either message or objects (ie console.dir(an instance of Track)). For lazy loading on pages that do not initiate themselves with a HTTP request, a simple “hello I am Component-X” string was logged to the console during component initialization (results not shown). Determining if lazy loading of application components is indeed functional, means that only one “hello” message should appear on the console and that that message should relate to the active component. For pages that are initialized with a HTTP request, simply loading the application is enough, because these components will send HTTP requests that are destined to fail as the end-user has not selected valid input. This means, their lazy-loading can be observed by seeing no HTTP status code errors when the end-user lands on the registration component.

Since the audio workbench is in its development phase, the current layout is test orientated for the essential audio functions. Owing to this, the workbench component currently relies heavily on the logging messages and objects to ensure that the page is indeed functional. The results of these tests are shown in the video demonstration of the submitted artifact and an example with Fig-20. The recording quality & range was tested with the microphone from my personal laptop & phone, and via USB with an audio interface using an electric bass guitar. The test results were successful in that the instrument's audio can be captured with sufficient quality for both devices & USB via a Rocksmith cable given the collective output from Phase-III. Finally, during testing of the recording tracks interface with an instrument via USB, it was noted that synchronizing separate instrument tracks for mixing was very difficult from an end-user perspective. To solve this issue, an auto-playback of the pre-

recorded audio feature was developed. The auto-playback feature allows the audio data of the newly recorded track to be naturally in sync with the other instrument tracks because the end-user can play along to the recorded tracks. However, this feature does not function with microphone-based recording because all audio data would be captured in the active recording track. The cause is that the recording device is receiving input from multiple sources, being the end-users instrument and the speakers from the recording device that are outputting previous recording. The solution for utilising auto playback on microphone recordings is to use headphones, this way the output device for the speakers are the headphones (Fig-5). Regardless, the issue merits that future versions of "bandCloud" offer the end-user with a check box they can toggle as to whether or not they want auto-playback of pre-recorded audio to be done.

Finally, during the local & deployed functional tests of posting of audio data two issues were encountered. First issue was that it was slow, because it takes 3 "bandCloud" related server hops to reach S3 (as discussed in the chapter 4 under avenues for development of the user interface). The second, which has solidified audio posts using pre-signed URLs, was the configurations needed to allow the frontend webserver to receive the currently implemented & non-compressed WAV format. Where the webserver needed to be configured to allow incoming request bodies with sizes in mega byte range, and also send requests with bodies in the same range (currently set to 50MB). Collectively the issues highlight the need for one, compressing the audio data (gzip related blob or use MP3 format) so that the related payloads are smaller. Then secondly, there is no need for multiple hops to S3 when this AWS service offers pre-signed URLs, which can be configured for set time and method a method (https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html) . An issue with implementing it however, is that two requests from the end-user's device will be sent. One request for the pre-signed URL and the second for the post/get of the audio data. However, two requests here is still better than the currently implemented solution.

Project Summary

In its current form "bandCloud" delivers the core aims of a cloud-based web service where potential users can (de)register, manage their account data and are offered core audio features (ie recording & mixing tracks). While each of three zones of the project, "*Frontend*", "*Backend REST API*" and "*Deployment*", have their own further development arc. The artifact that was submitted following completion of Phase-III, offers the core logic & organization required for delivering the avenues discussed in chapter 4.

References

Baeldung (no date) *Building a REST API with Spring*, *baeldung.com*. Available at: https://www.baeldung.com/rest-api-spring-guide (Accessed: 6 June 2022).

Brikman, Y. (2019) *Terraform: Up & Running, Writing Infrastructure as Code*. Second Edi. Edited by N. B. John Devins, Virgina Willson. O'Reilly.

CSO, C. S. O. (2021) *CSO Cloud Computing*, *Information Society Statistics Enterprises 2021*. Available at: https://www.cso.ie/en/releasesandpublications/ep/p-isse/informationsocietystatisticsenterprises2021/ (Accessed: 20 November 2021).

Docs, M. W. (2022a) *Compression Stream API*. Available at: https://developer.mozilla.org/en-US/docs/Web/API/CompressionStream/CompressionStream (Accessed: 12 August 2022).

Docs, M. W. (2022b) *Web Audio*. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API (Accessed: 6 August 2022).

Fernández-Del-Castillo, E., Scardaci, D. and García, Á. L. (2015) 'The EGI Federated Cloud e-Infrastructure', in *Procedia Computer Science*. doi: 10.1016/j.procs.2015.09.235.

Gonalez, D. (2017) *Implementing Modern DevOps: Enabling IT Organizations to Deliver Faster & Smarter*. Packt Publishing.

Google, C. (no date) *Google, MMOG Cloud Infrastructure*, *Overview of Cloud Game Infrastructure*. Available at: https://cloud.google.com/architecture/cloud-game-infrastructure (Accessed: 20 November 2021).

Hajian, M. (2019) 'Progressive Web Fundamentals', in *Progressive Web Apps with Angular: Create Responsive, Fast and Reliable PWAs Using Angular*. Apress, pp. 1–2. doi: https://doi.org/10.1007/978-1-4842-4448-7.

James F. Kurose, K. W. R. (2021a) 'Internet Protocol IPv4, Addressing, IPv6 and More', in *Computer Networking: A Top-Down Approach*. 8th edn. Pearson Education Ltd, pp. 360–377.

James F. Kurose, K. W. R. (2021b) 'Operation Security: Firewalls and Intrusion Detection Systems', in *Computer Networking: A Top-Down Approach*. 8th edn. Pearson Education Ltd, pp. 697–705.

James F. Kurose, K. W. R. (2021c) 'Principles of Network Applications', in *Computer Networking: A Top-Down Approach*. 8th edn. Pearson Education Ltd, pp. 114–118.

James F. Kurose, K. W. R. (2021d) ‘The Web and HTTP’, in *Computer Networking: A Top-Down Approach*. 8th edn. Pearson Education Ltd, pp. 135–142.

Kamal Kant Hiran, Ruchi Doshi, Dr. Temitayo Fagbola, M. M. (2019a) ‘Cloud Computing Concepts; Advantages & Disadvantages of Cloud Computing’, in *Cloud Computing: Master Cloud Computing Concepts, Architecture and Applications with Real-world Examples and Case Studies*. BPB Publications, pp. 7–11.

Kamal Kant Hiran, Ruchi Doshi, Dr. Temitayo Fagbola, M. M. (2019b) ‘Comparison of Traditional and Cloud Computing Paradigms’, in *Cloud Computing: Master Cloud Computing Concepts, Architecture and Applications with Real-world Examples and Case Studies*. First. BPB Publications, pp. 11–17.

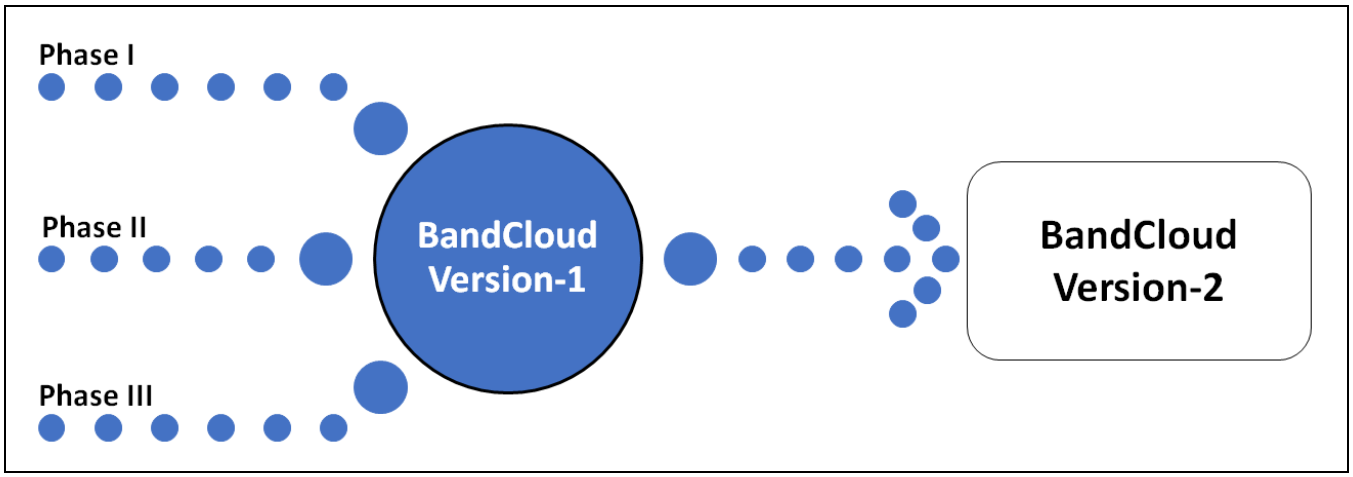
Larman, C. and Basili, V. R. (2003) ‘Iterative and incremental development: A brief history’, *Computer*. doi: 10.1109/MC.2003.1204375.

NHLBI (2016) *NHLBI Trans-Omics for Precision Medicine Whole Genome Sequencing Program. TOPMed*, *https://www.nhlbiwgs.org/*.

Sciore, E. (2019) *Java Program Design: Principles, Polymorphism, and Patterns*. Edited by A. Jecan. Apress. doi: https://doi.org/10.1007/978-1-4842-4143-1.

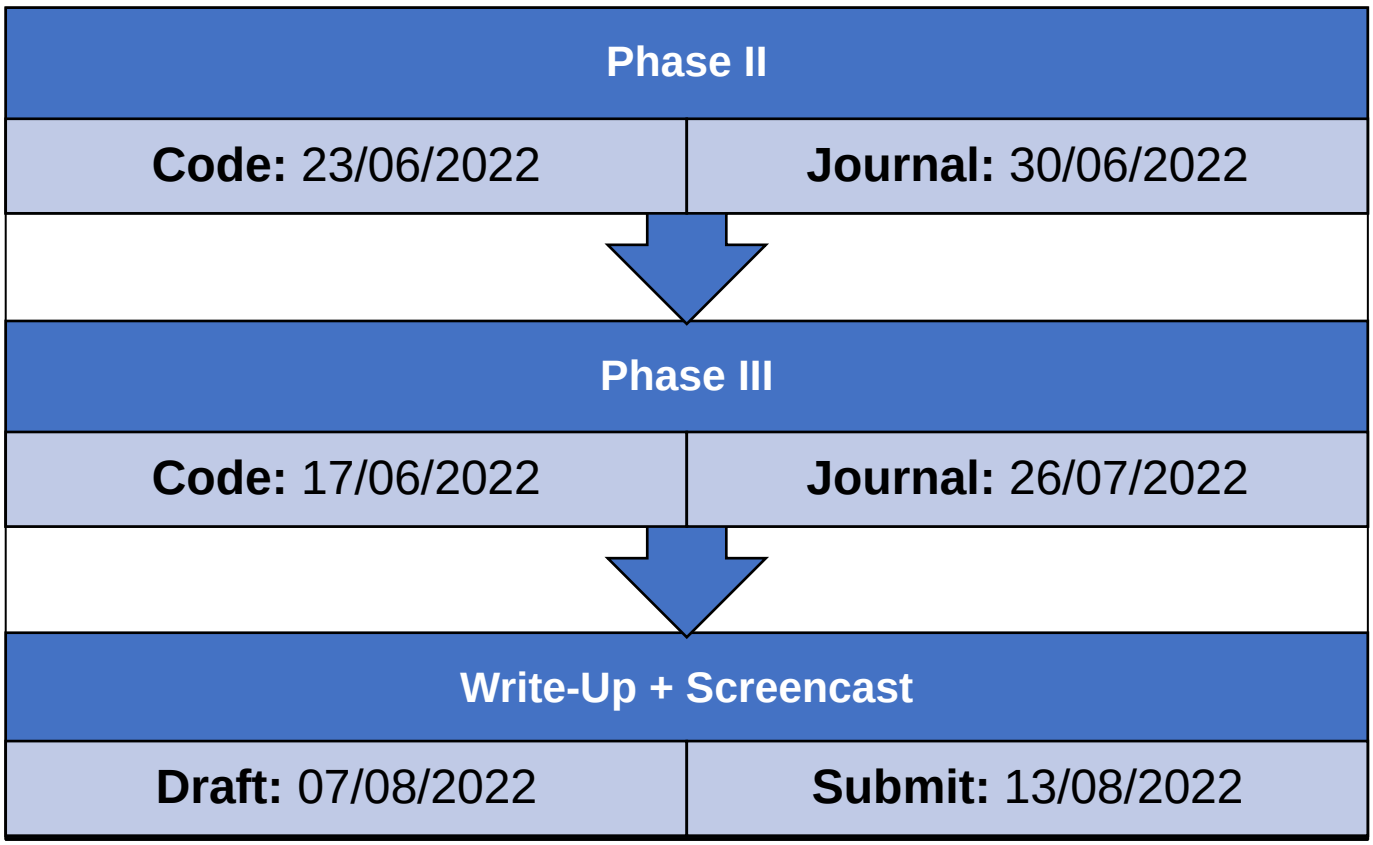
Figure Appendices

Figure 1: Project Phases



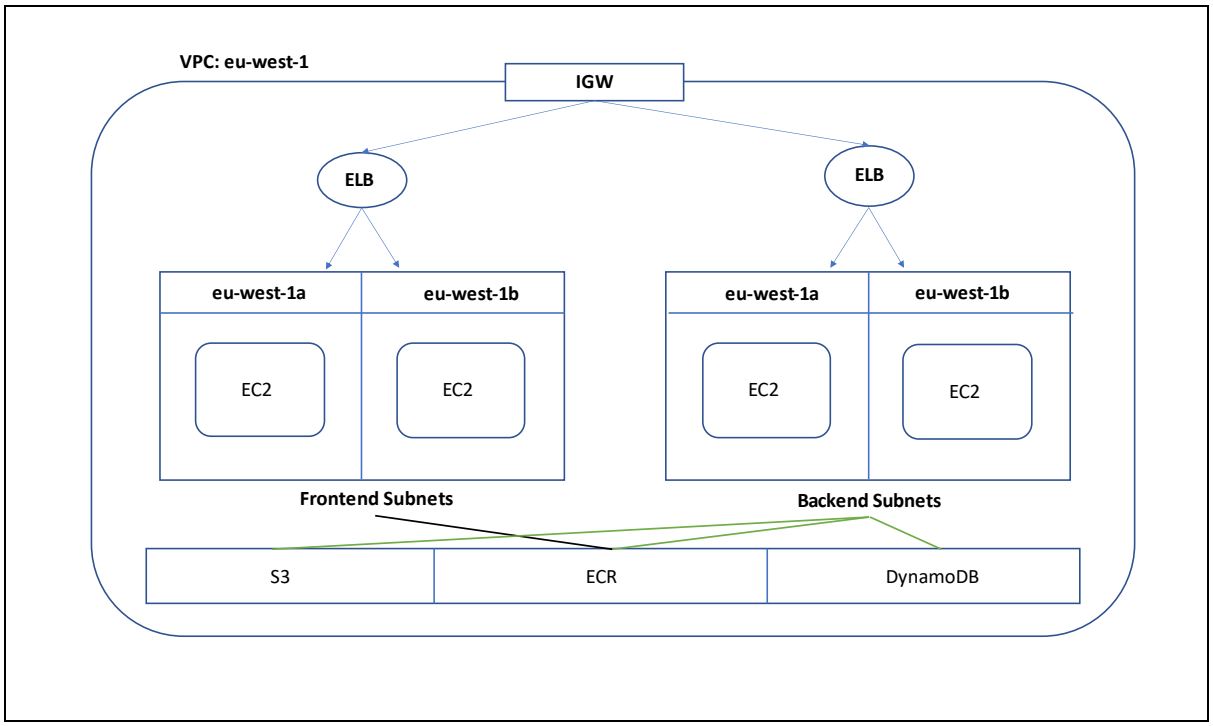
Displays the development phases of bandCloud. Where the priorities of Phase-I are towards broader research, feasibility, and exposure to the new technologies. Phase-II is concerned with exploring how these new technologies can be specifically used for bandCloud. Phase-III is concerned with refining their implementation, applying any priority development notes, and considering viable avenues of development for the next version.

Figure 2: Project Timeline



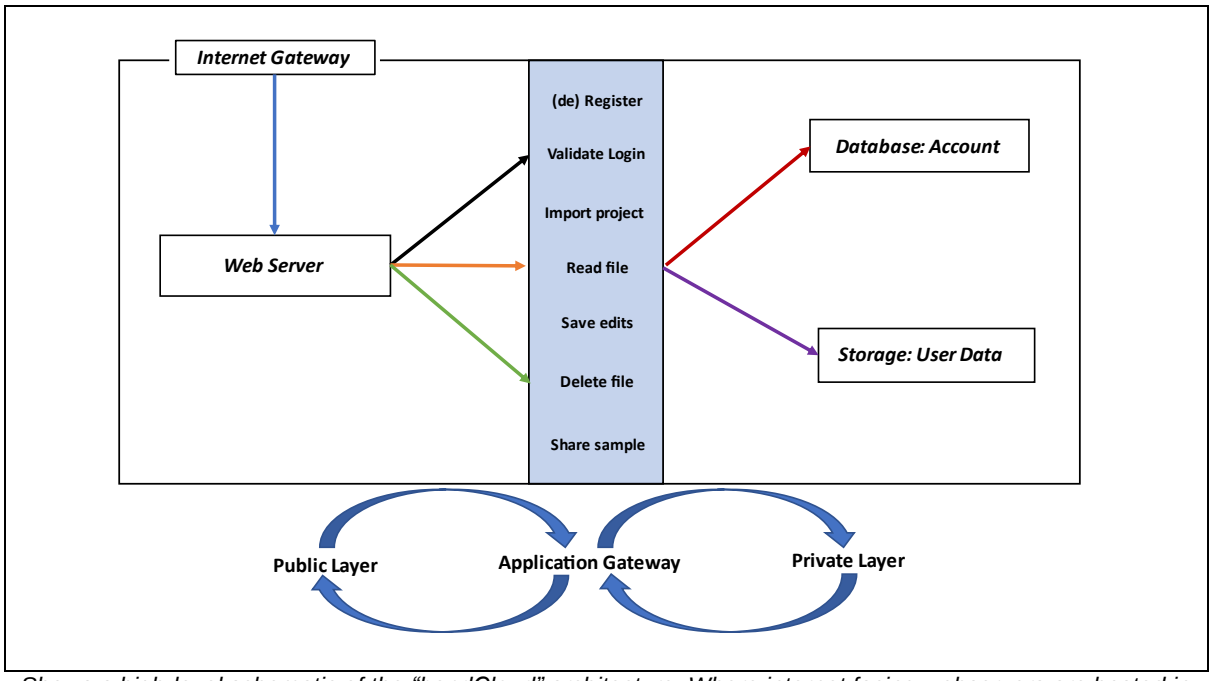
Displays project time following completion of Phase-I (ie semester 2 of the course). The core deliverable here is the continual growth in deployment of both front & backend applications, and how they provide the essential functionality. The level of which can be built upon in Phase-III which acknowledges the ambitious sprints of Phase-II.

Figure 3: Phase-II bandCloud Architecture



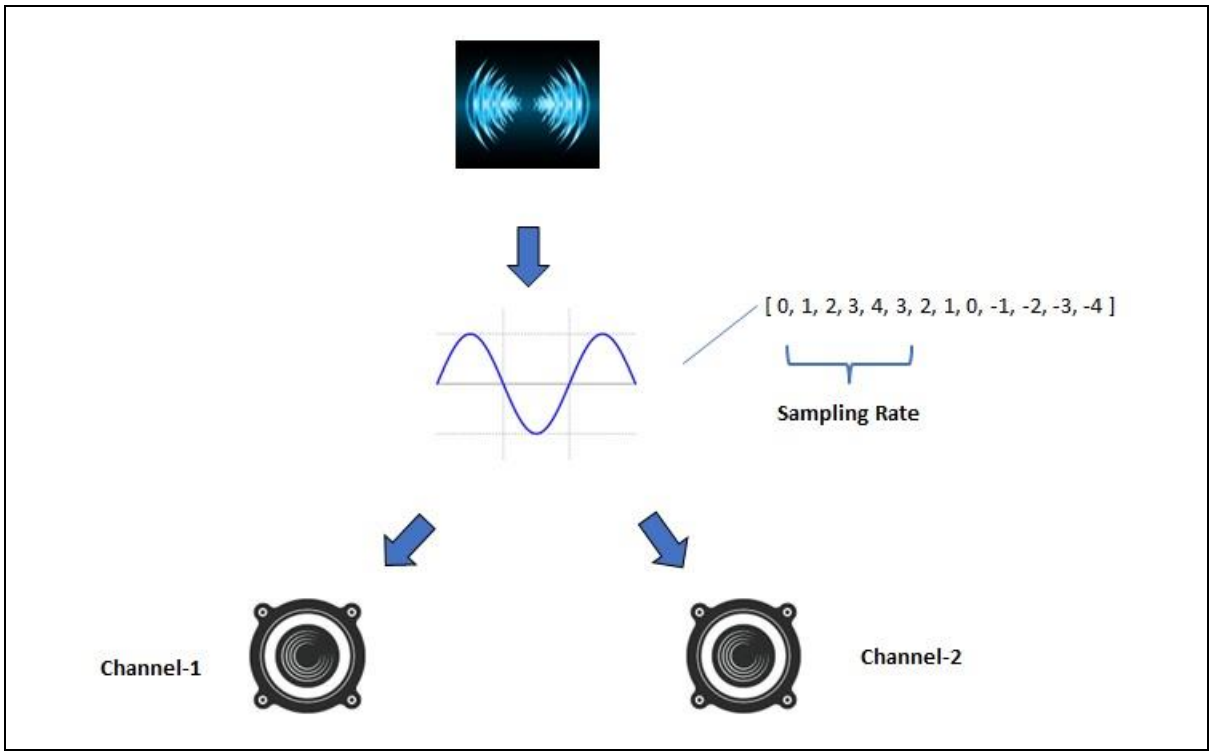
Shows the developmental architecture for hosting the "bandCloud" application on AWS from Phase-II. Both the front & backends are deployed within a single VPC and sit behind separate application elastic load balancers and both were initially accessible via a single internet gateways. Both applications are deployed into two subnets within different availability zones for high-availability. EC2 instances are spun-up in these AZs depending on ELB status 200 health checks, which pull the appropriate docker container from ECR. Finally, only the backend EC2 instances can interact with S3 & DynamoDB.

Figure 4: Phase-III bandCloud Architecture



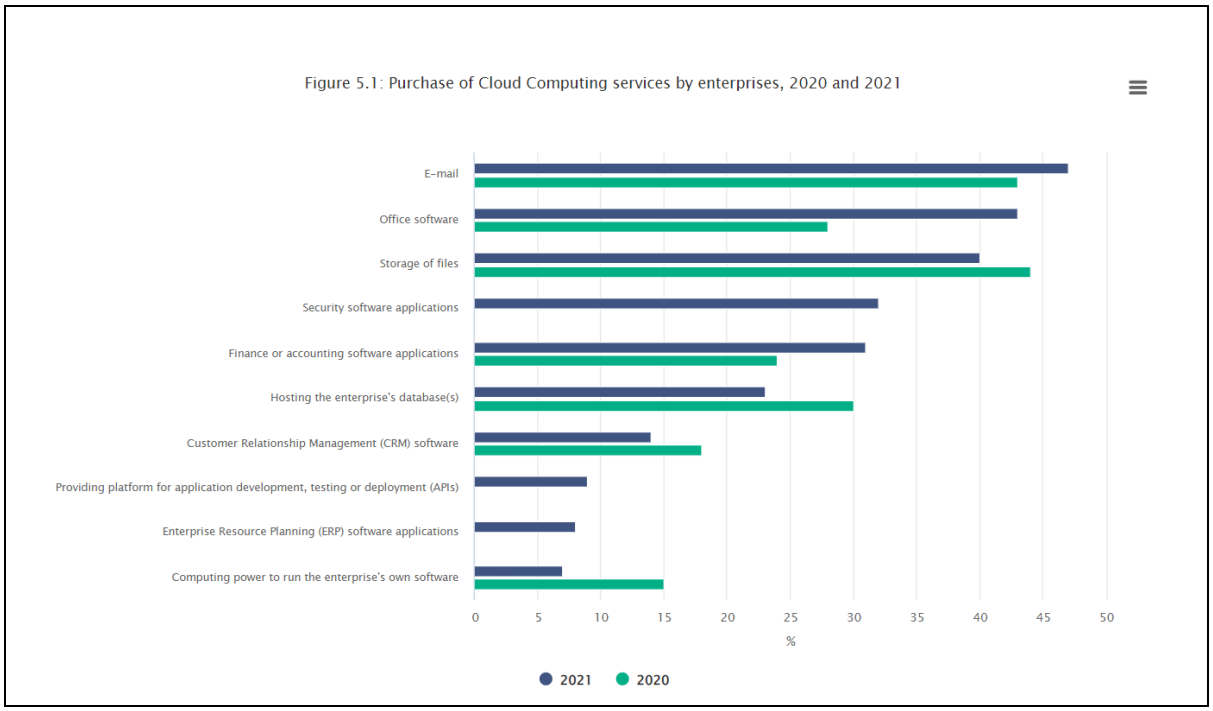
Shows a high-level schematic of the "bandCloud" architecture. Where internet facing webservers are hosted in public subnets, which proxy client-side requests to an application gateway hosted in private subnets. The application gateway / RESTful API service is permanently authorized to send API calls to the private database & storage services on behalf of the end-user.

Figure 5: Digital Representation of Audio Signals



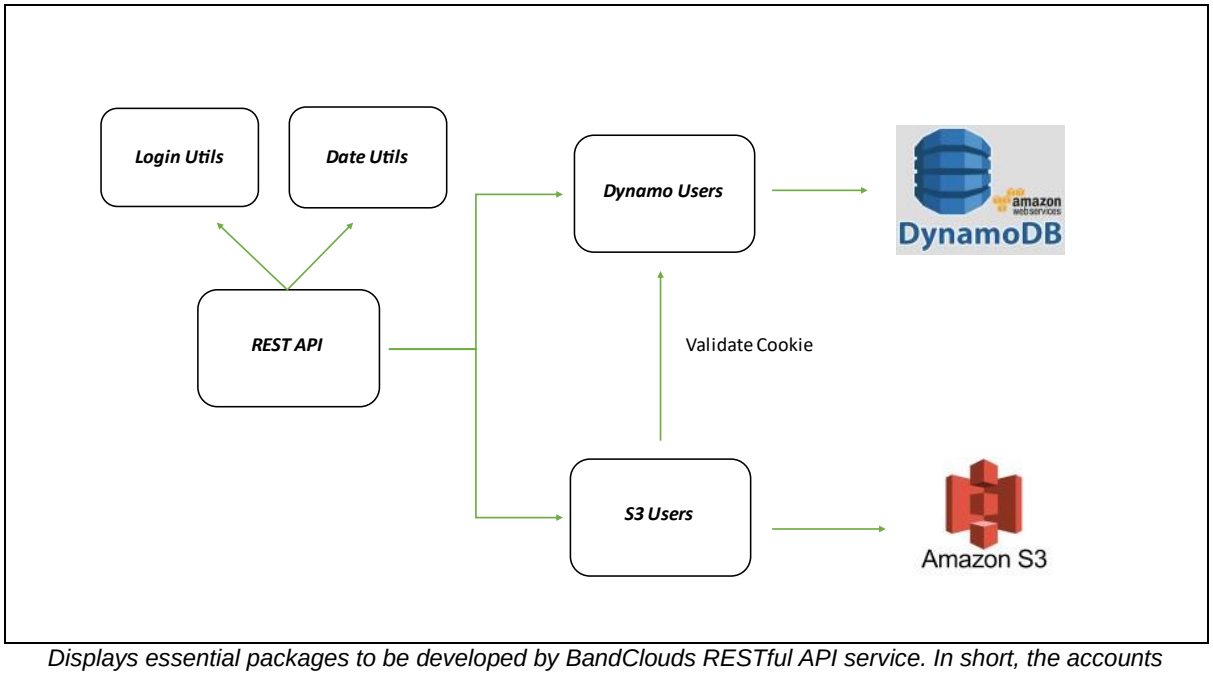
To convey a high-level overview of how audio is "represented" by computers. In brief, sound is a wave that propagates in three dimensions. Assuming this prorogation to be uniform we can represent it as a 1-Dimensional wave with properties that describe how to map frames (elements in an array) to seconds (sampling rate is the number of frames per second). These attributes make up the core Pulse-Code Modulation, PCM, object used in the field of audio signal processing to represent audio signals. Representing audio signals in this way allows logic to be built on top of the object to interact with the data (play, pause, mix, edit). Where playing the same PCM object can be done by sending it to the available speakers of the device called channels

Figure 6: Cloud Computing Adoption by Enterprises in Ireland



Screenshot was taken Central Statistics Office survey Information Society Statistics Enterprises survey, Cloud Computing section: https://www.cso.ie/en/releasesandpublications/ep/p-isse/informationsocietystatisticsenterprises2021/cloudcomputing/

Figure 7: Backend Design Schematic



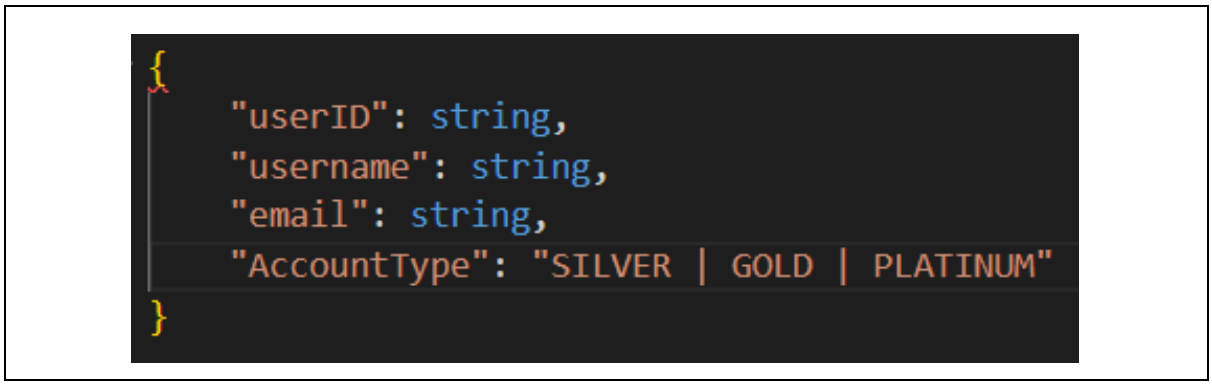
Displays essential packages to be developed by BandClouds RESTful API service. In short, the accounts controller will make use of packages designed to simplify login & date validation. Which will make use of AWS-orientated packages for managing model collections, such as Users, Projects, etc. Since cookies are to be used, all controllers will need to make use of a cookie validation method in the DynamoUsers package.

Figure 8: User Model

```
{
    "userID": "from backend server (BE)",
    "username": "letters only, length of 20",
    "email": "username@domain.com",
    "password": "hashed",
    "salt": "from BE",

    "session": {
        "authKey": "random hex string",
        "expireDate": "dd/MM/yyyy, 1 week",
        "Is cookie": true
    },

    "accountType": {
        "Enum Value": "SILVER | GOLD | PLATNIUM",
        "Storage Limit": "5 | 15 | 40 MB",
        "Allowed Distortion": "GOLD | PLATNIUM",
        "Allowed All Effects": "PLATNIUM",
        "Share Raw": "GOLD | PLATNIUM"
    }
}
```

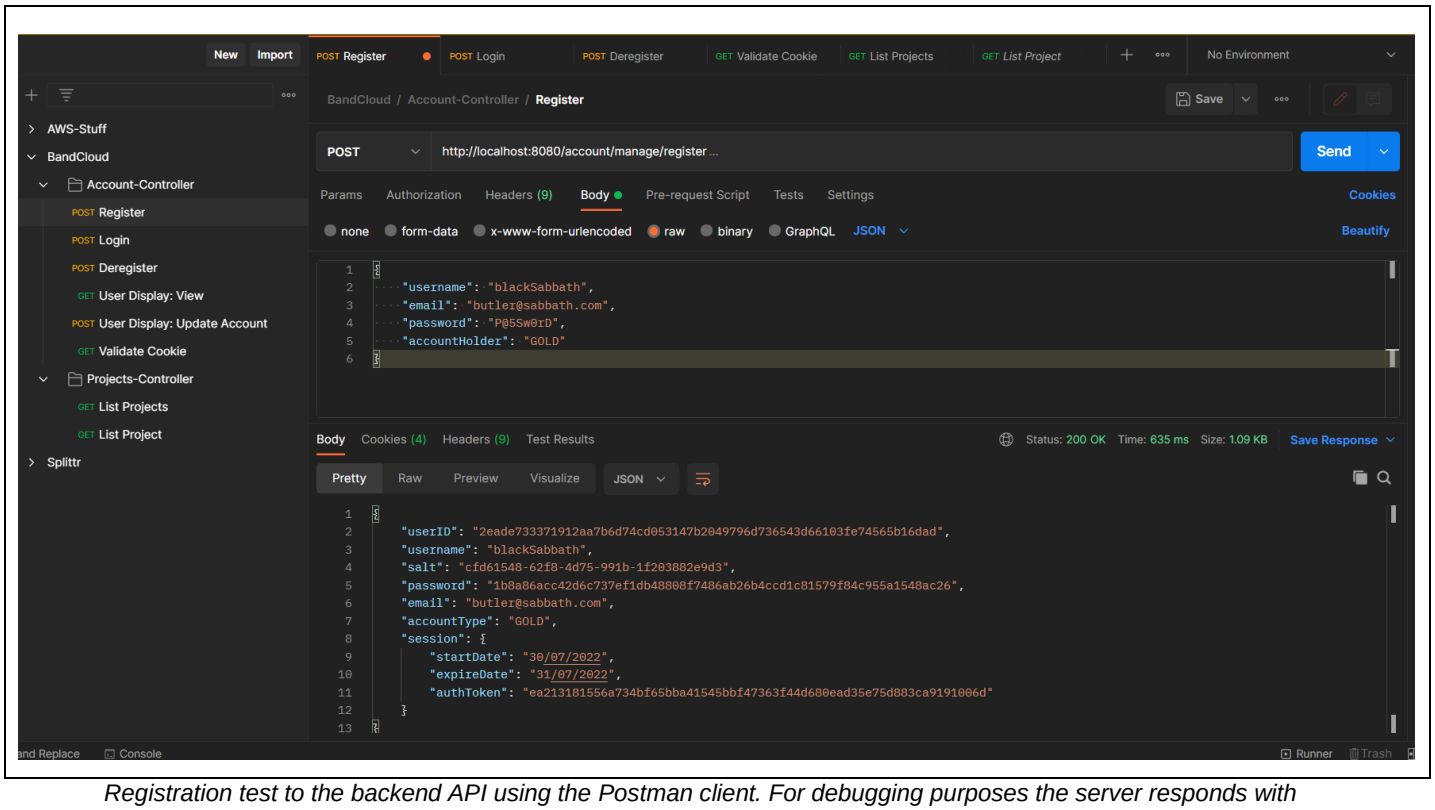
Displays the core attributes of the user model that is created by the system and is stored in a "Users" DynamoDB table. Input from the end-user (ie username, email, password, and Account Type Enum Value) is used to register users. All controllers make use of a "BandCloudSession" object for creating & validating user sessions which are stored in the user's table. Passwords are stored in the user's table as a hexadecimal hash of the user's password and a user-specific randomly generated string (UUID) called salt. When end-users update their password, the system also updates the salt value.

Figure 9: UserDisplay Model



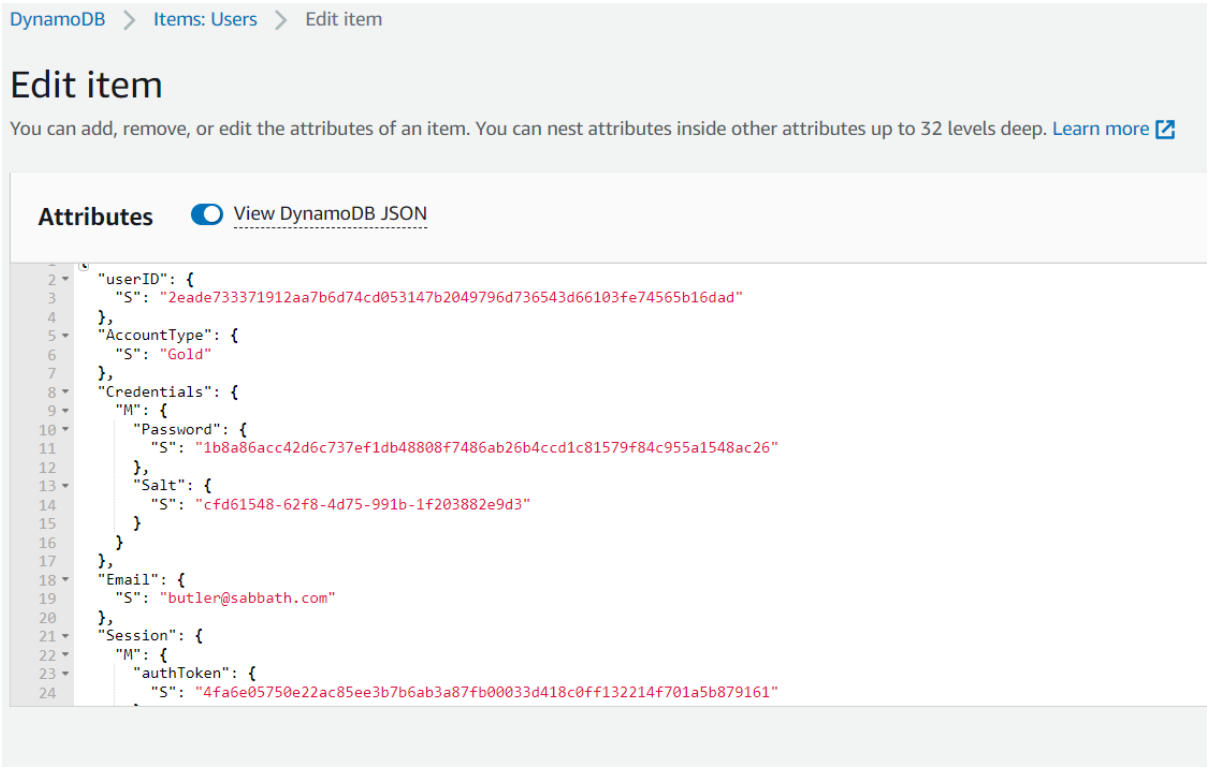
Displays the attributes of the UserDisplay object used to represent & update an end-users account data. For updates to account data, a user display object was used with an attribute for password. The userID is the only immutable attribute.

Figure 10: Registration Endpoint Test



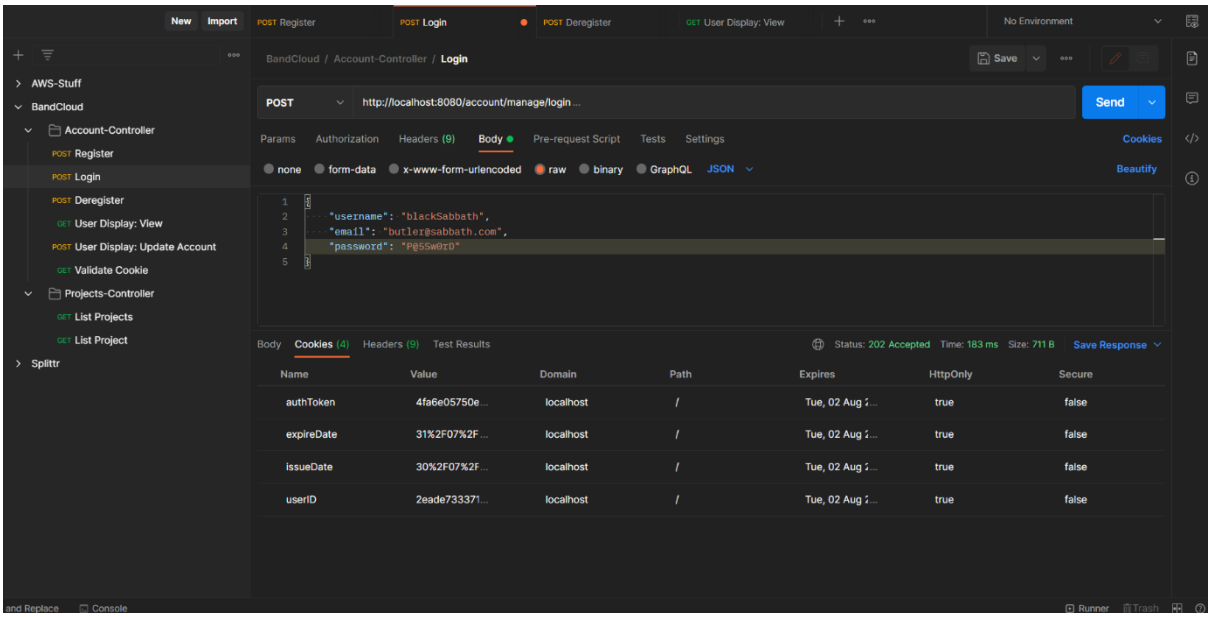
Registration test to the backend API using the Postman client. For debugging purposes the server responds with the new user created. The important response is the session cookie which authenticates the end-user to the system, as each previlaged endpoint validates these data. Invalid input causes system to respond with an enum value indicating first erroneous field, which should be further developed into a list of erroroneous fields.

Figure 11: Registration Data in DynamoDB



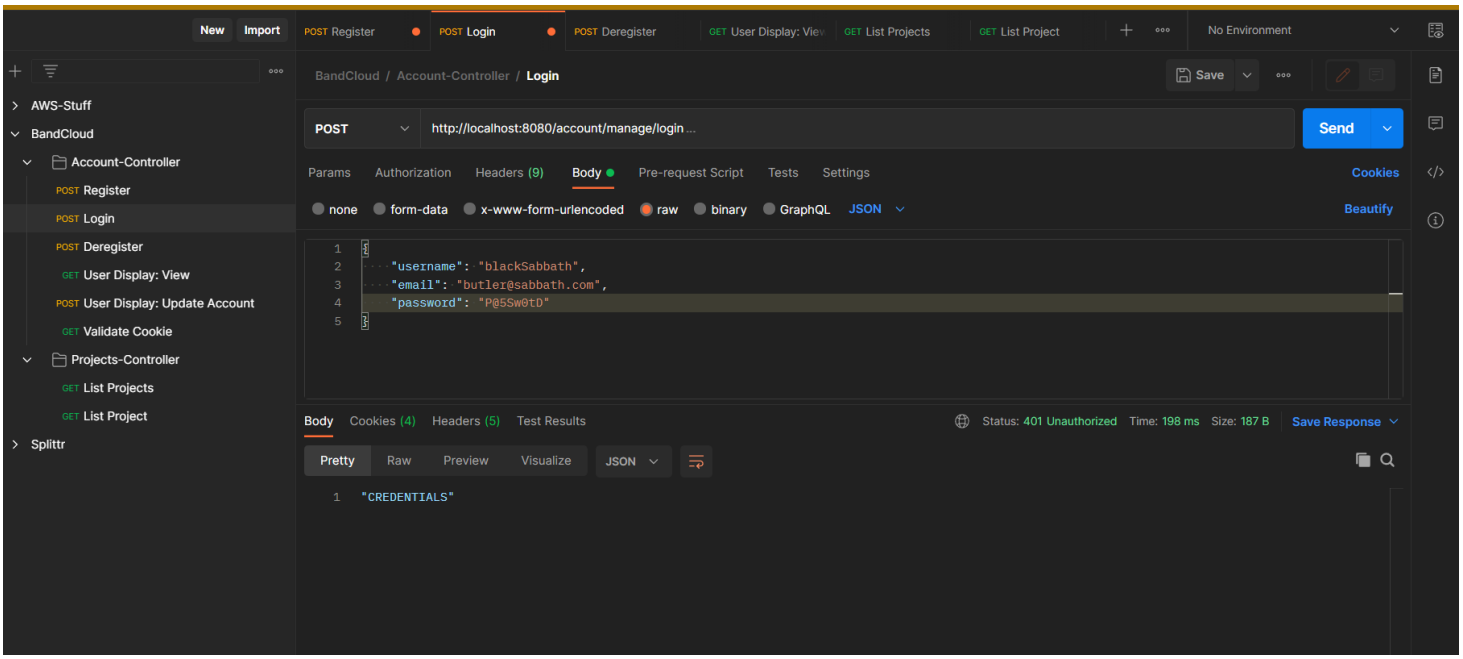
Shows the successful storage of new user registration from Fig-11

Figure 12: Login Endpoint Test



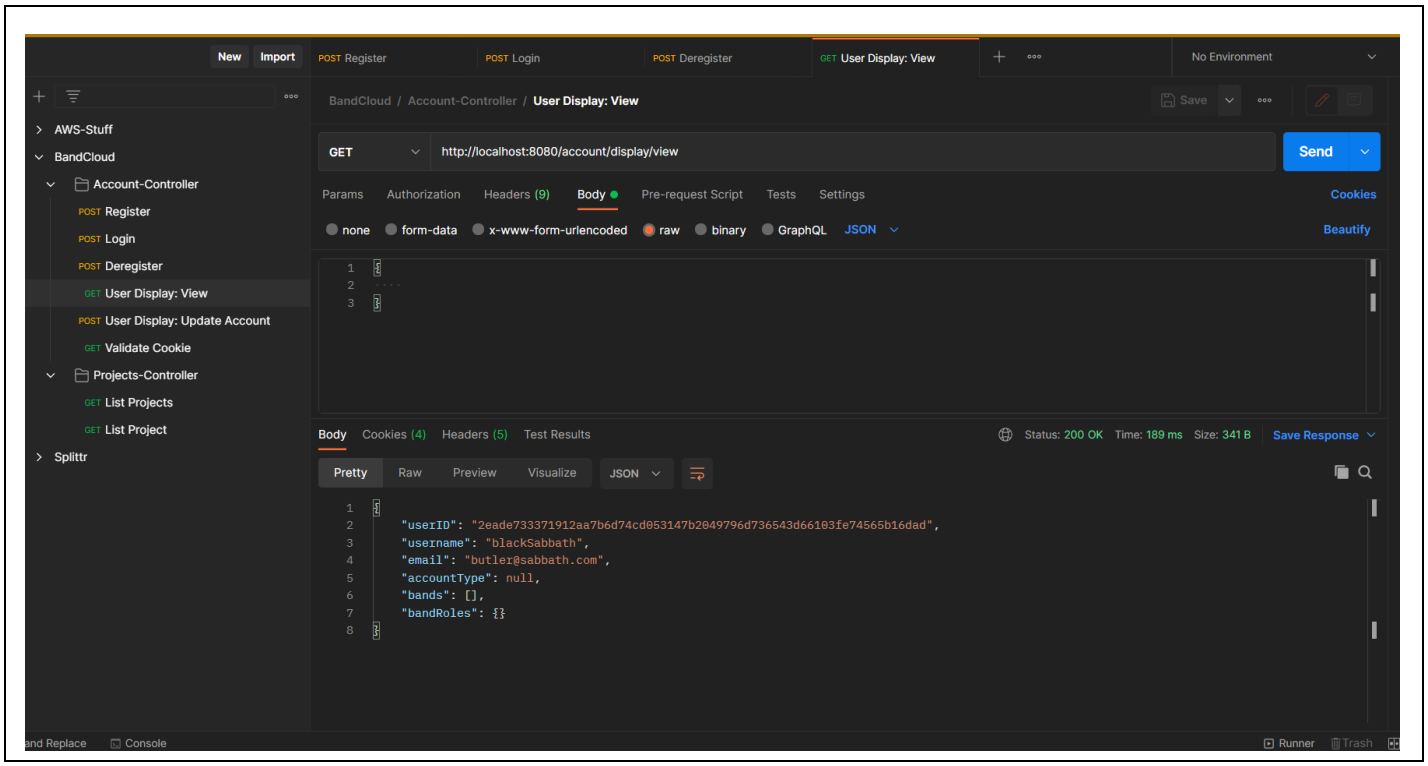
Shows the output from the login endpoint test which responds a validation enum value, and a session cookie if successful. Shown here is the session cookie with the same fields as Fig-11.

Figure 13: Invalid Login Test



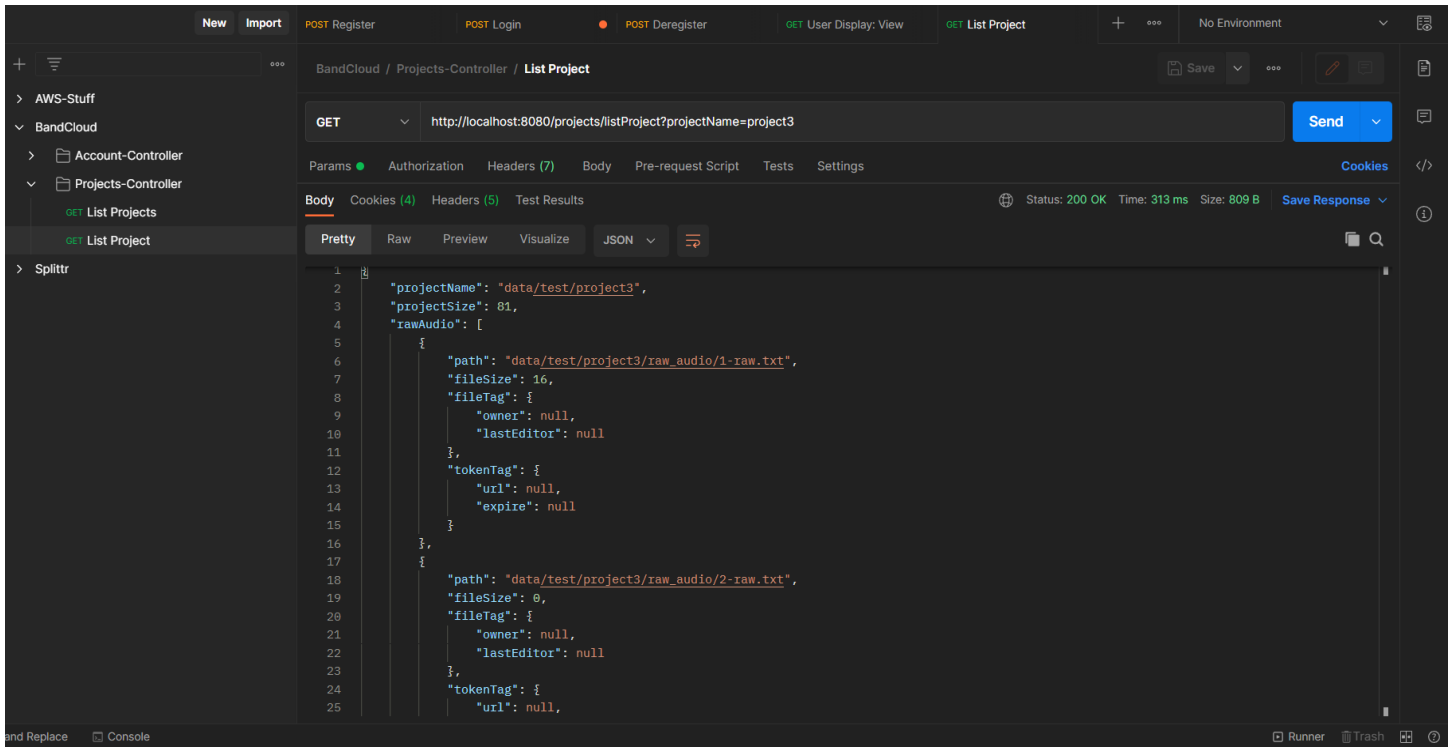
Shows the results from an invalid login attempt for a registerd user. The server responds with an unauthorized 401 error, and a value from its validation enum to provide context for the error code to the clent.

Figure 14: User Display Endpoint Test



Shows the output from testing the previlaged user display endpoint. Where all fields excluding the userID are changeable by the system based on the data that is posted to the "/account/display/update" endpoint.

Figure 15: Project Endpoint Test



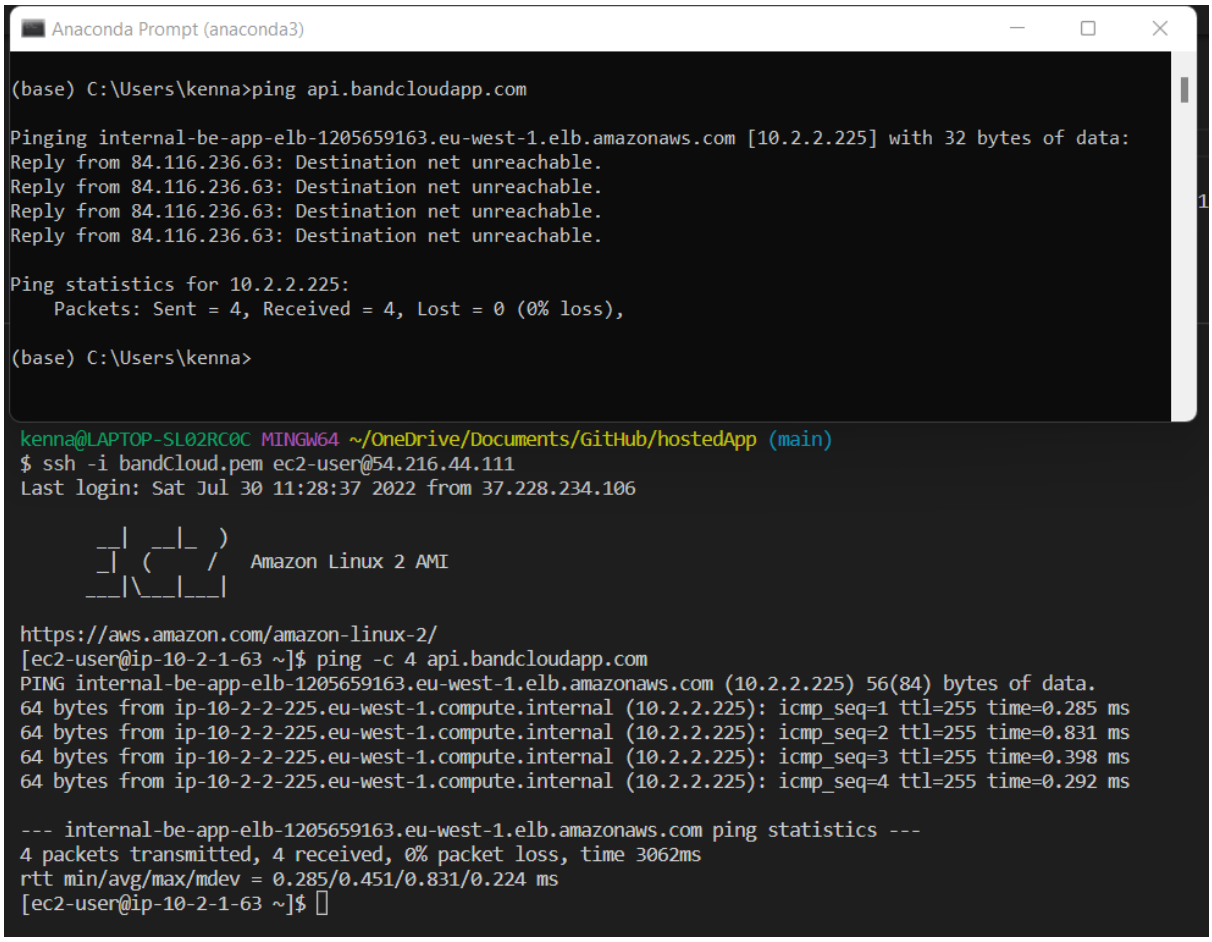
Shows the output from the ilst project endpoint test. The endpoint is responsible for providing the client with meta-data about the user's active project, the client then translates these meta-data into audio data. Currently, the project endpoints use mock data so that the client-side development could progress more fruitfully.

Figure 16: Deployment Test

```
# Build repo      You, now • Uncommitted changes
cd ~/bandCloud-Angular
aws ecr get-login-password | docker login --username AWS --password-stdin XXX.dkr.ecr.eu-west-1.amazonaws.com/bandcloud-frontend
docker build --no-cache -t bandcloud-frontend .


# Sanity check
docker run -it -p 8080:8080 bandcloud-frontend bash launchServer.sh


# Push if good
docker tag bandcloud-frontend XXXX.dkr.ecr.eu-west-1.amazonaws.com/bandcloud-frontend:latest
docker push XXX.dkr.ecr.eu-west-1.amazonaws.com/bandcloud-frontend:latest


# Sanity check: Pull image and run
docker pull XXX.dkr.ecr.eu-west-1.amazonaws.com/bandcloud-frontend
docker run -it -p 8080:8080 XXX.dkr.ecr.eu-west-1.amazonaws.com/bandcloud-frontend bash launchServer.sh
```

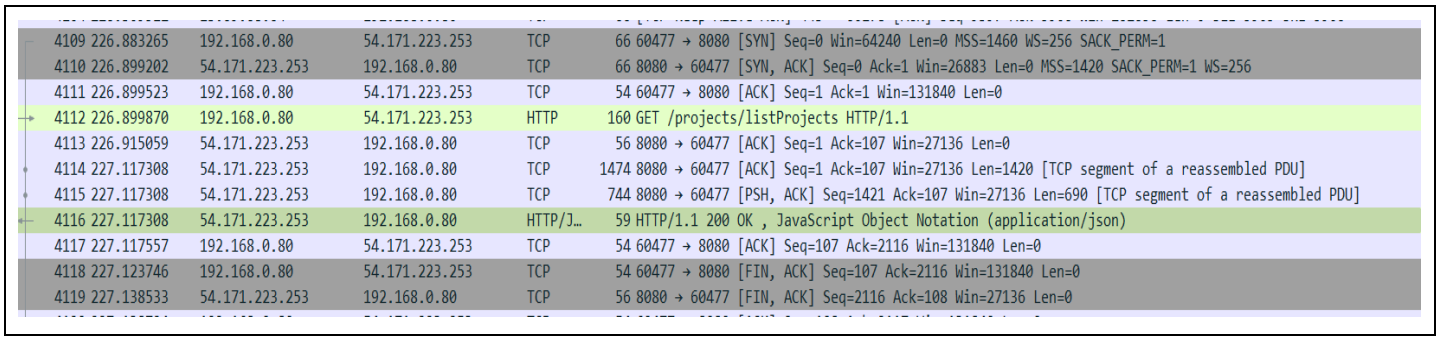
Shows an example deployment test for docker containers pre-deployment. Where before commiting an updated version of the image, the built container is verified in this case sending a HTTP request to the hosting EC2 instance. Once verified, the image is pushed to projects private ECR repository which is then sanity checked by pulling the image and running the associated application.

Figure 17: Private Network Test



Shows the results from ICMP-IPv4 tests to assess whether or not the backend load balancers are on a private network. Where the same sub-domain was sent ICMP packets from my home network, and the "bandCloud" VPC network. In brief, my local network can query canonical name record "api.bandcloudapp.com" which points to the internal application load balancer, but does not have a route to that domain. However, since the resource is deployed within the "bandCloud" VPC there is a route to this resource for packets that originate within the internal network.

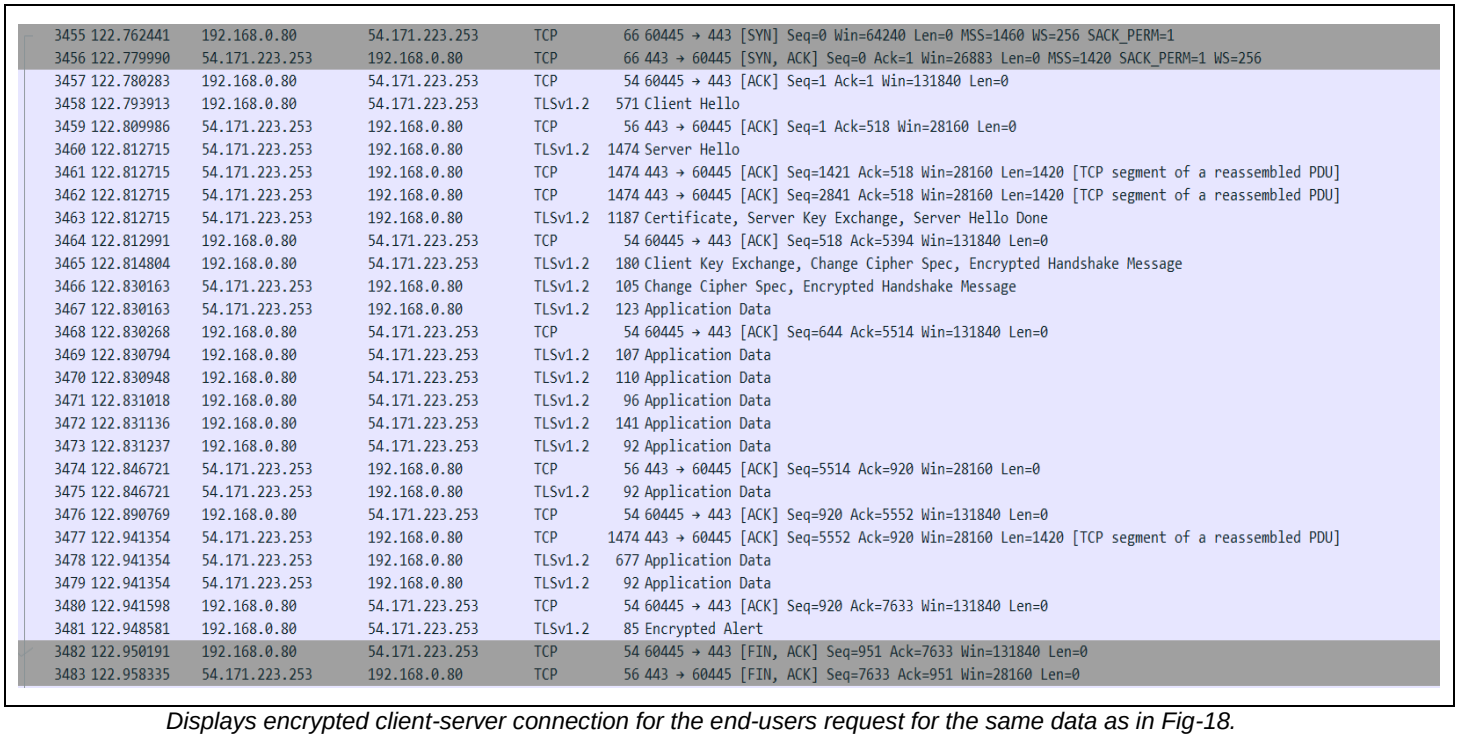
Figure 18: HTTP Packet Capture



4109	226.883265	192.168.0.80	54.171.223.253	TCP	66	60477 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
4110	226.899202	54.171.223.253	192.168.0.80	TCP	66	8080 → 60477 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1420 SACK_PERM=1 WS=256
4111	226.899523	192.168.0.80	54.171.223.253	TCP	54	60477 → 8080 [ACK] Seq=1 Ack=1 Win=131840 Len=0
4112	226.899870	192.168.0.80	54.171.223.253	HTTP	160	GET /projects/listProjects HTTP/1.1
4113	226.915059	54.171.223.253	192.168.0.80	TCP	56	8080 → 60477 [ACK] Seq=1 Ack=107 Win=27136 Len=0
4114	227.117308	54.171.223.253	192.168.0.80	TCP	1474	8080 → 60477 [ACK] Seq=1 Ack=107 Win=27136 Len=1420 [TCP segment of a reassembled PDU]
4115	227.117308	54.171.223.253	192.168.0.80	TCP	744	8080 → 60477 [PSH, ACK] Seq=1421 Ack=107 Win=27136 Len=690 [TCP segment of a reassembled PDU]
4116	227.117308	54.171.223.253	192.168.0.80	HTTP/J…	59	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
4117	227.117557	192.168.0.80	54.171.223.253	TCP	54	60477 → 8080 [ACK] Seq=107 Ack=2116 Win=131840 Len=0
4118	227.123746	192.168.0.80	54.171.223.253	TCP	54	60477 → 8080 [FIN, ACK] Seq=107 Ack=2116 Win=131840 Len=0
4119	227.138533	54.171.223.253	192.168.0.80	TCP	56	8080 → 60477 [FIN, ACK] Seq=2116 Ack=108 Win=27136 Len=0

Displays unencrypted client-server connection for an end-user request to the "/project/listProjects" endpoint, which queries data stored in a private AWS S3 bucket.

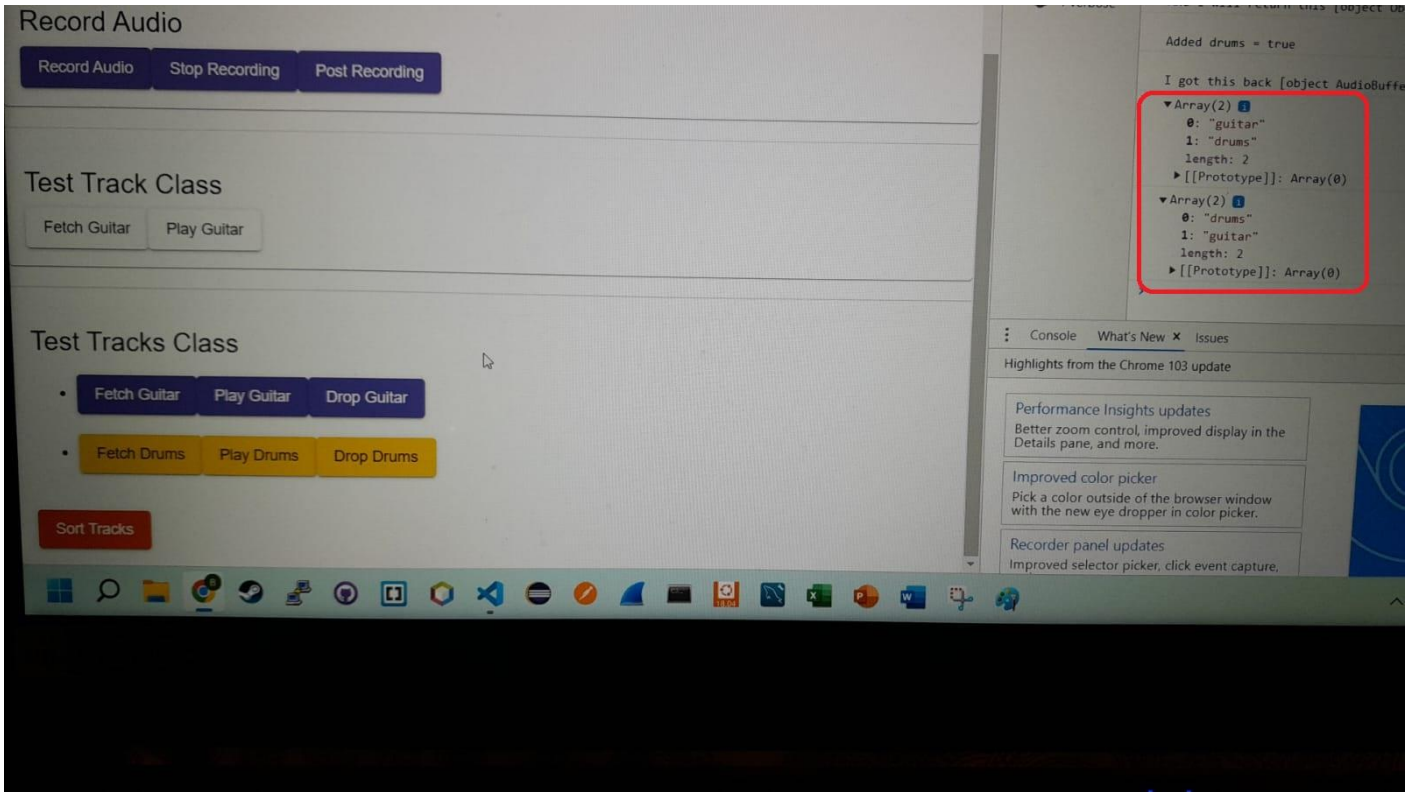
Figure 19: HTTPs Packet Capture



3455	122.762441	192.168.0.80	54.171.223.253	TCP	66	60445 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
3456	122.779990	54.171.223.253	192.168.0.80	TCP	66	443 → 60445 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1420 SACK_PERM=1 WS=256
3457	122.780283	192.168.0.80	54.171.223.253	TCP	54	60445 → 443 [ACK] Seq=1 Ack=1 Win=131840 Len=0
3458	122.793913	192.168.0.80	54.171.223.253	TLSv1.2	571	Client Hello
3459	122.809986	54.171.223.253	192.168.0.80	TCP	56	443 → 60445 [ACK] Seq=1 Ack=518 Win=28160 Len=0
3460	122.812715	54.171.223.253	192.168.0.80	TLSv1.2	1474	Server Hello
3461	122.812715	54.171.223.253	192.168.0.80	TCP	1474	443 → 60445 [ACK] Seq=1421 Ack=518 Win=28160 Len=1420 [TCP segment of a reassembled PDU]
3462	122.812715	54.171.223.253	192.168.0.80	TCP	1474	443 → 60445 [ACK] Seq=2841 Ack=518 Win=28160 Len=1420 [TCP segment of a reassembled PDU]
3463	122.812715	54.171.223.253	192.168.0.80	TLSv1.2	1187	Certificate, Server Key Exchange, Server Hello Done
3464	122.812991	192.168.0.80	54.171.223.253	TCP	54	60445 → 443 [ACK] Seq=518 Ack=5394 Win=131840 Len=0
3465	122.814804	192.168.0.80	54.171.223.253	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
3466	122.830163	54.171.223.253	192.168.0.80	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
3467	122.830163	54.171.223.253	192.168.0.80	TLSv1.2	123	Application Data
3468	122.830268	192.168.0.80	54.171.223.253	TCP	54	60445 → 443 [ACK] Seq=644 Ack=5514 Win=131840 Len=0
3469	122.830794	192.168.0.80	54.171.223.253	TLSv1.2	107	Application Data
3470	122.830948	192.168.0.80	54.171.223.253	TLSv1.2	110	Application Data
3471	122.831018	192.168.0.80	54.171.223.253	TLSv1.2	96	Application Data
3472	122.831136	192.168.0.80	54.171.223.253	TLSv1.2	141	Application Data
3473	122.831237	192.168.0.80	54.171.223.253	TLSv1.2	92	Application Data
3474	122.846721	54.171.223.253	192.168.0.80	TCP	56	443 → 60445 [ACK] Seq=5514 Ack=920 Win=28160 Len=0
3475	122.846721	54.171.223.253	192.168.0.80	TLSv1.2	92	Application Data
3476	122.890769	192.168.0.80	54.171.223.253	TCP	54	60445 → 443 [ACK] Seq=920 Ack=5552 Win=131840 Len=0
3477	122.941354	54.171.223.253	192.168.0.80	TCP	1474	443 → 60445 [ACK] Seq=5552 Ack=920 Win=28160 Len=1420 [TCP segment of a reassembled PDU]
3478	122.941354	54.171.223.253	192.168.0.80	TLSv1.2	677	Application Data
3479	122.941354	54.171.223.253	192.168.0.80	TLSv1.2	92	Application Data
3480	122.941598	192.168.0.80	54.171.223.253	TCP	54	60445 → 443 [ACK] Seq=920 Ack=7633 Win=131840 Len=0
3481	122.948581	192.168.0.80	54.171.223.253	TLSv1.2	85	Encrypted Alert
3482	122.950191	192.168.0.80	54.171.223.253	TCP	54	60445 → 443 [FIN, ACK] Seq=951 Ack=7633 Win=131840 Len=0
3483	122.958335	54.171.223.253	192.168.0.80	TCP	56	443 → 60445 [FIN, ACK] Seq=7633 Ack=951 Win=28160 Len=0

Displays encrypted client-server connection for the end-users request for the same data as in Fig-18.

Figure 20: Testing Tracks Audio Sorting Method



Displays results from testing the bubble sorting method of the Tracks class, where the 'guitar' track of 7s is a larger audio signal than the 'drums' track of 4s. In brief, each instance of a Track within the Tracks collection is sorted ascendingly by the length of their audio signal array. The method allows tracks of different lengths to be mixed by the system because one recorded track could be larger than the other based on how the end-user uses the recording buttons. Such as when they start playing their instrument once the record button is pressed, or stop the recording once finished.

Table Appendices

Table 1: Essential Frontend Requirements

Essential Frontend Requirements	
Section	**Component**
Account	Register, login, deregister
	View & edit account
Project	Create, drop & list projects
	Open workbench on project
Workbench	Record audio
	Add/drop audio track(s)
	Mix audio
	Save audio

Lists the essential requirements of the frontend web application. The development of the application was divided into three sections; an accounts section is responsible for delivering end-users with account administration and authentication to the system. A project section to present end-users with CRUD operations to manage their "bandCloud" projects and resume work on a specific project. Finally, a workbench to present end-users an interface to work on a project, in terms of being able to add/drop tracks to mixable track, record new tracks for the project, and save these data to the systems storage solution.

Table 2: Backend Accounts Controller Requirements

Backend Accounts Controller Requirements				
Root	**Endpoint**	**Method**	**Request**	**Response**
/account/	manage/registration	POST	Body: username, email and password	Cookie, enum value
	manage/login	POST	Body: username, email and password	Cookie, enum value
	manage/deregister	POST	Body: username. Email & password	Enum value
	display/view	GET	-	User display object
	display/update	POST	Body: username, email, password & account type	Enum value

Displays the essential requirements and design plan for the endpoints developed to allow end-users to manage their account data. The essential functionality of the account controller is to allow end-user to (de)register and login into their accounts, so that can be authenticated to use the system for their data and update it as needs be. The controller makes use of session cookies so that user experience is simplified as HTTP requests are independent and uses an enum to support frontend error handling of invalid login credentials, or registration data.

Table 3: Backend Projects Controller Requirements

Backend Projects Controller Requirements				
Root	**Endpoint**	**Method**	**Request**	**Response**
/projects	-	GET	-	Projects Object
	{projectName}	GET	-	Project Object
	upload/{projectName}	POST	Body: Audio Blob as String	-
	drop/{projectName}	DELETE	-	-
	listMockProjects	GET	-	Projects Object
	listMockProject	GET	-	Project Object
	post-recording	POST	Body: Audio Blob as String	

Displays essential requirements of the Projects controller. The controller utilizes the end-users session for userID, username, and authentication provided by the login endpoint. Allowing the Projects endpoints to accept user-provided project names and maintain a standard hierarchy for prefixes/paths on S3. Audio data is posted to the API as base64 encoded string and retrieved via pre-signed URLs (limit the number of "hops" to storage). The "listMockProject(s)" and "post-recording" endpoints were developed to encapsulate functional requirements in a standard way and inform the developmental requirements of their related deployment endpoints & not pose developmental bottlenecks (see "Implementation of System" for their status).