Carleton University
COMP4905 Honours Project
JamSpace
Nick Bachalo
Professor Louis Nel
Abstract

It’s hard enough as it is for musicians to come together and play or practice when they live in the same area, but if they are remote, then it can be impossible. JamSpace provides a way for musicians to play together online with friends and network with other musicians. It is a web application built on the MEAN Stack (Mongo, Express, Angular, Node) utilizing PeerJS and the web browser built-in user media APIs for peer to peer client audio communication and local audio track recording. It was designed to be an exploration of web development and VOIP technology, and succeeded as both a learning experience and an implementation.
Acknowledgements

I would like to express my humblest thanks for Professor Louis Nel of the Computer Science Department at Carleton University for allowing me to take this project on and explore web development and internet audio communication for the first time.

I would also like to express sincere thanks to my workplace manager, Jan Smith of Juniper Networks for being patient and understanding of my situation while I balanced work, school, and life responsibilities.

This project would not have been possible without the many available software libraries, frameworks, and APIs available to me for these purposes. NodeJS, ExpressJS, AngularJS, MongoDB, PassportJS, Mongoose, Bower, WebRTC, JSSip, SIPML5, and PeerJS.

Online services also had a large impact on the success of this project, MLab’s Mongo database cloud hosting, and PeerJS’ PeerJS server cloud hosting.

This project was something that has been on my mind for a long time, and to be given the opportunity to finally realize it has been a very interesting and enlightening experience. I learned a great deal about self-driven project development, full stack web development, and about VOIP communication protocols and development. I plan to continue working on this project even after this as there are many more features that I now feel equipped with sufficient knowledge to add.
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is JamSpace?</td>
<td>5</td>
</tr>
<tr>
<td>The Foundation of JamSpace</td>
<td>7</td>
</tr>
<tr>
<td>JamSpace User Authentication</td>
<td>8</td>
</tr>
<tr>
<td>JamSpace Rooms</td>
<td>10</td>
</tr>
<tr>
<td>VOIP Communication</td>
<td>12</td>
</tr>
<tr>
<td>WebRTC and JamSpace Considerations</td>
<td>12</td>
</tr>
<tr>
<td>PeerJS in JamSpace</td>
<td>14</td>
</tr>
<tr>
<td>User Media in JamSpace</td>
<td>17</td>
</tr>
<tr>
<td>JamSpace Results</td>
<td>19</td>
</tr>
<tr>
<td>Using JamSpace</td>
<td>22</td>
</tr>
<tr>
<td>Future Work</td>
<td>28</td>
</tr>
<tr>
<td>Resources and Citations</td>
<td>29</td>
</tr>
<tr>
<td>Appendix</td>
<td>31</td>
</tr>
</tbody>
</table>
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 1 - MEAN Stack Diagram</td>
<td>6</td>
</tr>
<tr>
<td>FIGURE 2 - User Authentication Flowchart</td>
<td>9</td>
</tr>
<tr>
<td>FIGURE 3 - AngularJS Data Bindings</td>
<td>11</td>
</tr>
<tr>
<td>FIGURE 4 - PeerJS Signalling Diagram</td>
<td>15</td>
</tr>
<tr>
<td>FIGURE 5 - Peer Connection Diagram</td>
<td>16</td>
</tr>
<tr>
<td>FIGURE 6 - JamSpace Operation Flow Chart</td>
<td>20</td>
</tr>
<tr>
<td>FIGURE 7 - JamSpace Homepage</td>
<td>22</td>
</tr>
<tr>
<td>FIGURE 8 - JamSpace Sign Up Page</td>
<td>23</td>
</tr>
<tr>
<td>FIGURE 9 - JamSpace Login Page</td>
<td>23</td>
</tr>
<tr>
<td>FIGURE 10 - JamSpace User Profile Page</td>
<td>24</td>
</tr>
<tr>
<td>FIGURE 11 - JamSpace Room List Page</td>
<td>25</td>
</tr>
<tr>
<td>FIGURE 12 - JamSpace Room Page</td>
<td>26</td>
</tr>
<tr>
<td>FIGURE 13 - JamSpace Recording</td>
<td>27</td>
</tr>
<tr>
<td>FIGURE 14 - JamSpace Track Recorded</td>
<td>27</td>
</tr>
</tbody>
</table>
What is JamSpace?

As a budding musician, it is hard to find time to get together with friends in order to play music together. Living remotely, this makes it nigh-impossible to get a “jam” session planned out and seen through. In order to help find a solution to this, JamSpace was created. JamSpace is a web application designed to allow musicians to connect to each other remotely in “jam spaces” and play music together through the internet. Users can create accounts, and join or create rooms for their friends or others. In these rooms, all present users can stream audio to each other and record local tracks for playback while practicing a song. The goal of this project was to explore full stack web application development and voice over IP (VOIP) communication protocols. It was also meant as a way to see if such a project would be possible and could be built on in the future for even more functionality and interest.

JamSpace is built on the MEAN web application stack. MEAN is an acronym for the JavaScript frameworks and tools that it primarily uses for operation. This is a full-stack enabled web application base, which means that it includes the front end client side logic framework AngularJS, and the back end server side logic frameworks NodeJS and ExpressJS as well as MongoDB for persistent server storage of user documents and other website data.

NodeJS

NodeJS is the backbone of the MEAN stack, providing full server functionality and routing capabilities, as well as including the Node Package Manager, which is used for installing all of the necessary dependencies for any web application. It is an open-source, cross platform JavaScript runtime environment built for running server side JavaScript code. [NodeJS, n.d.]

ExpressJS

ExpressJS is the standard framework for NodeJS which extends its capabilities and provides simplified server functionality and routing, as well as allowing for easier inclusion of modules and other libraries and middleware. [Express, n.d.]

MongoDB

MongoDB is the persistent storage. It is an open-source, cross-platform document based database that allows for saving of documents and data, such as user profiles, settings, and other documents. The Mongoose middleware is most often paired with the MEAN Stack.
because it serves as a direct interface between NodeJS servers and MongoDB databases. [Mongo, n.d.a]

AngularJS

AngularJS is a frontend library that is used across Google applications. It makes use of distinct views, controllers, services, and directives to power the front end of any web application. The controllers handle the main logic of the frontend, while the services act as middlemen to communicate between NodeJS servers and AngularJS controllers. The directives are used for directly modifying the views. [Angular, n.d.]

FIGURE 1 - MEAN Stack Diagram
The Foundation of JamSpace

The MEAN stack offers a lot of modern and simple functionality for web development. Being based entirely on JavaScript, it allows for a single unified development environment and cuts down on the need to be familiar with many languages. Using ExpressJS and NodeJS, setting up a server and programming how to handle routes is quick and easy, allowing for very nice separation and abstraction of code.

AngularJS is a declarative style frontend framework. This means most of the code for it is 'behind-the-scenes'. For example, to display a data binding, AngularJS' way of updating the data in the HTML view of the web browser, one simply needs to enter an “ng-” tag that corresponds with the type of data binding that one wants to use, and AngularJS handles the rest. All of these factors, when combined, make the MEAN stack a very quick and easy framework to develop web applications on. It allows for elegant coding organization and keeps a unified workflow through exclusive use of JavaScript. [Wayner, 2017]

When compared to the other industry standard, the LAMP (Linux, Apache, MySQL, PHP) stack, MEAN offers much more robust and dynamic front end control through AngularJS, and much simpler backend control through use of NodeJS and the included Node Package Manager. [Rajput, 2015] The Node Package Manager, or NPM for short, is a tool included with NodeJS that simplifies the installation of any NodeJS compatible library, framework, middleware, or other software by making use of one simple command line argument. Due to the vast amount of these libraries, frameworks, etc that are developed and maintained for NodeJS, there will be something useful for any developer. This is because of another benefit of using the MEAN stack. Each of the components included is open source and cross platform, which means that any developer can extend, improve, modify, or otherwise develop for, any piece of the MEAN stack. As such, there is extensive support and community involvement with the MEAN stack. Additionally, seeing as it is entirely built on JavaScript, it is compatible with any other previously standing JavaScript library. For example, jQuery, another common front end JavaScript library, will still work alongside AngularJS.

When taking into account the versatility, modernity, and vast support for the MEAN stack, it was an easy decision to make for using it as the basis for JamSpace. Another small dilemma was the decision for the version of Angular to use, as Angular2 was recently released
and, while offering many new features over the traditional AngularJS, it wasn’t quite as well established, and still had the potential for incompatibilities and unforeseen bugs. [Muller, 2015] Thus, AngularJS was used for the front end of JamSpace due to it’s better establishment, user support, and stability.

Due to the dynamic nature of the application, another option for the front end was ReactJS, a JavaScript frontend that is used in Facebook. [ReactJS, n.d.] It is an open source and cross platform solution that is similar to AngularJS. However, it is also quite new comparatively, and for the same reasons as not using Angular 2, AngularJS was the preferred framework.

**JamSpace User Authentication**

Middleware is the software that extends or enhances the capabilities of another software. Due to the nature of the MEAN stack, there exists plenty of useful pieces of middleware for developers of all types.

The most essential functionality for any website is user authentication. There are many methods for implementing this, and because there is such a widespread need for websites to have user authentication, there are endless middleware libraries compatible with NodeJS available. PassportJS is a user authentication middleware that implements something it calls a strategy, which is used by the server to verify a user when they login, sign up, or even perform an action that only a specific user is allowed to do. [PassportJS, n.d.]

Another consideration at this stage was the database being used. With MongoDB, there were two options: host a local database, or use a cloud storage solution. In order to make it easier to deploy JamSpace and move it around while maintaining the persistence of the data, a cloud solution was used. MLab’s cloud hosted Mongo databases allow for free hosting of any database containing up to 0.5 GB of information, which for testing purposes of JamSpace is more than enough. It works by providing a user with a remote database key that Mongoose then uses to connect to the remote database and make it available to NodeJS as a database object. [mLab, n.d.]
The first design consideration was the information about the user to be stored on the server, and how. A Mongoose schema for a JamSpace user contains the user’s username and password. For security reasons, the password is hashed so that it is not stored in plaintext on the database. PassportJS makes this easier by providing methods for doing so.

In JamSpace, PassportJS is used to streamline the user authentication process for the server. It is used in conjunction with the Mongoose middleware which is used to interface between NodeJS and a MongoDB database. Users create accounts or log in to pre-existing accounts, get verified through PassportJS, and are then compared or added to the MongoDB database with Mongoose. The schema based approach that mongoose takes makes working with a MongoDB database much easier as it uses a similar syntax to JSON objects, another standard for MEAN stack’s JavaScript based approach. It also has built in functionality that makes it easy to operate on information stored in the database. CRUD (Create, Read, Update, Delete) operations are the basic operations performed when using a database, and Mongoose functions allow for the basics, as well as extended functionality.

![User Authentication Flowchart](image)
If a user does not currently have an account, then they can sign up to create a new account, creating a new user schema in the database. If they already have an account, then they can login, which securely compares the given information with what is stored in the database. Once successfully logged in, the user can view their profile information.

**JamSpace Rooms**

After user functionality was implemented, the next step was to implement the rooms in which users would connect to each other and be able to communicate together. The main concern was with having a clean and responsive interface for users. This is where using AngularJS was handy, as it builds on basic HTML and adds many useful features for DOM (document object model, a term used for the interactive front end of a web application) manipulation. AngularJS can be included either by including the scripts hosted by Google, or by installing a local library to use.

Through Bower, it is easy to install any kind of frontend framework or middleware. Bower is a similar program to the Node Package Manager, and is installed with NPM as well. It is used as a means of managing frontend components, such as HTML, CSS, JavaScript, fonts, and image files. [Bower, n.d. ] In JamSpace, Bower was used to install the frontend framework AngularJS, as well as the included BootStrap CSS library which allows for neater formatting of HTML views.

Users must have a way to view the list of rooms available to join, and if there are no rooms to their liking, they need the ability to create a room on their own. Firstly, the room schema needed to be built. This is the model that the server and database will use to keep track of the traits for each room in the database. The traits that a room has are its name and a set of user names present in the room. There does not need to be any checking for if users within rooms are unique, as not only is each individual user in JamSpace unique, but also by the nature of a set, each element stored within is unique.

In order to view the rooms that are available, the rooms.html view makes use of an AngularJS data binding to show the list of rooms that are currently stored on the server. How this works is first, the Angular controller for the room list makes a request to the server along an
API route to get the currently available rooms. It then populates a locally saved list of rooms and uses this list in conjunction with a repeater method that iterates through the list and shows the rooms to the user. A user must also be able to create a room, done by using another data binding to get the input from a user, then POSTing to the server the new room information, and then moving the user into the room.

![Diagram of AngularJS Data Bindings]

**FIGURE 3 - AngularJS Data Bindings**

The room view is divided into three different view panes. The first and largest one is the track list pane, which shows the list of recorded tracks. Right below that is the track recording pane, where the user is shown the recording controls. On the right hand side is the bar for showing the list of users that are currently in the room.

This is how the room view should be divided in order to keep functionalities abstracted from each other.
**VOIP Communication**

The next step was to research solutions for allowing users to communicate with each other while in a room. When dealing with VOIP in a web browser, there are only so many solutions. These are: self-programmed JavaScript encoding/decoding functions used in conjunction with browser built-in media streaming tools, Flash, WebRTC (Web Real Time Communication), and SIP (Session Initiation Protocol). Of course, the first option would have been incredibly time consuming and impractical, as it is not based on any pre-existing technologies and is not guaranteed to be cross platform or accessible in any way. Flash is slow, buggy, and also no longer fully supported in most browsers with the advent of HTML5. [Anderson, 2013a] This leaves WebRTC or SIP as the only options to consider for this task.

Both WebRTC and SIP are communication protocols that use sockets to send audio, video, and text data between client and server endpoints. They enable full audio sharing capability for use across multiple platforms. The key difference is that while SIP can be used for anything, it is WebRTC that outshines it for web browser performance. “SIP works best when used simply: telephone calls, instant messaging and some video and audio are the main territories of SIP. But with WebRTC, not only do those same technologies come into play—file transfers, audio and video—but they come in on Web browsers, meaning that the intermediary step of softphones is no longer required.” [Anderson, 2013b] The issue with WebRTC is that it requires help in establishing connections with both other callers, or servers.

In essence, SIP can be used without WebRTC, but WebRTC can (but not always) not be used without SIP or some other protocol. [Ventimiglia, 2013] Being designed for web browsers in mind however WebRTC is the only real option available to be used for JamSpace.

**WebRTC and JamSpace Considerations**

JamSpace implements WebRTC compatible libraries because it is the standard for web communication, and because it is highly scalable, with it’s high range of compatibility allowing for easy extension of JamSpace to other platforms.

Seeing as WebRTC is the new current standard for web browser communication, it was then necessary to implement a library that allowed for JamSpace users to connect and share
audio with each other. There were a few solutions that were looked into in order to implement web audio calling in JamSpace. These were: sipML5 and webrtc2sip, JsSIP, and PeerJS.

SipML5 and webrtc2sip are combined because sipML5 implements SIP functionality with JavaScript that can be implemented on multiple systems. The webrtc2sip library in conjunction is used to make it easier for web browsers to implement sipML5 by providing a WebRTC to SIP proxy. [DuoBango, n.d.] Essentially this allows for users on web browsers to make calls to any other SIP enabled device. JsSIP was similar however it worked as JavaScript library that already combined the abilities of sipML5 and webrtc2sip libraries. [JsSIP, n.d.]

They are all client side implementations of WebRTC communication, however PeerJS and the other two had some distinct differences. The implementations of JsSIP and sipML5 would also require the setup of a private server called an Asterisk server, based on a telephone technology called PBX (Private Branch Exchange). [Wikipedia, n.d.a] This is something that is used to connect SIP clients and transfer any data such as audio, video, or text to other SIP client endpoints. There would have been two options for setting up an Asterisk server, hosting one on my own, or using an online provider. Installing a local server involved minimal setup, however it was not as versatile as using an online service. The issue with using an online service is that most Asterisk or SIP server providers all required data plans or money subscriptions. On top of these limitations, each user would need separate log in information for the server, on top of the information already in use for JamSpace. This all served to create a bottleneck in JamSpace.

This made PeerJS the final consideration for JamSpace. PeerJS implements a peer to peer WebRTC connection interface in JavaScript. It does require a server, but only for brokering connections between peers, which means that all data transfer is done between peers. [PeerJS, n.d.]
PeerJS in JamSpace

PeerJS was the ideal choice for JamSpace. Due to its peer to peer nature, it allowed for the lowest latencies for connected users, and while it didn’t entirely do away with the requirement of another server, the server for PeerJS was much simpler, as it did not require separate user authentications and was not responsible for data streaming through it. A server can be implemented directly into NodeJS with the PeerJS server package. It simply runs with, or beside, a NodeJS server deployment. To connect to it, peers just need to know the address and port of the server to connect to. Sadly, due to some unknown issue, the local installation of the server did not work. It is possible that this was due to some internal networking issue on the testing computer, but trying some changes did not seem to help. When researched online, this issue seemed to not be an isolated case, as there were several other people with the same issue. Due to the PeerJS user base’s small size, there weren’t any solutions to be found. When experimenting with the server, using different STUN and TURN servers to connect peers together did not solve the issue, nor did creating a secondary PeerJS server to run alongside the main JamSpace app server.

Thus, if PeerJS was to work, it was required to use the PeerJS’ free cloud hosted service. All this required was an API key that is plugged in to each client trying to connect to the corresponding server. Then, once a client has registered with the server, it is then able to know about other peers to connect to based on a unique ID that peers have. Peers can assign unique IDs for themselves, or have the server assign these IDs. Since JamSpace usernames are all unique, it seemed the most consistent to use these usernames as the peer IDs registered with the server so that when both users had PeerJS sessions opened, they would be able to connect to each other easily by just referring to each other’s usernames/IDs.

A PeerJS server works by learning about a peer’s ID and registering it, so that when another peer connects and wants to call this ID, the server can then broker the connections between these peers and allow them to communicate. In PeerJS, when a peer wants to call another peer, it sends a ‘call’ event and the ID of the peer it wants to call to the server. The server then recognizes this peer ID, and passes the call event on to the corresponding peer, where the incoming call is then handled.
The PeerJS server makes use of two other intermediary servers to help connect users via peer to peer connections. STUN (Session Traversal Utilities for NAT) servers are used to discover the IP addresses of peers requesting connections by giving the peers IP addresses that are allowed to be discovered by others in public domains. [Wikipedia, n.d.b] This is what makes it possible for the connections to be made between PeerJS peers. As a fallback, TURN (Traversal Using Relays around NAT) servers are used to relay stream information between peers if UDP or TCP connections fail. [Wikipedia, n.d.c] [Dutton, 2013]

Implementing PeerJS in JamSpace was surprisingly easy to do. All of the required code is client side, so all that was needed was to program all of the functionality in the Angular controller for the Room view. Since there was no server logic to implement, this was all the
work that was needed to be done to get PeerJS working. However, it was also necessary to get every single user connected in a room to enter a call in PeerJS together.

To ensure this was happening, every next user that joined a room would initiate a PeerJS call with every other user connected in the same room. So for example, Peer 1 joins an empty room. There is nobody to call, so it just gets registered with the PeerJS server and awaits other peer calls. When Peer 2 joins, the client sees that Peer 1 is already in the room, so Peer 2 connects with PeerJS and calls Peer 1’s ID. Now Peer 1 and Peer 2 are in a call. Peer 3 then joins the room, sees Peer 1 and Peer 2’s IDs, connects to PeerJS, and then calls both Peer 1 and Peer 2, which both answer the call in return. A fourth peer joins and repeats the same process as Peer 3. Now all 4 peers are connected to each other in an audio call. If we look at the connected peers as a node and edge graph, this would form a complete graph like so:

![Peer Connection Diagram](FIGURE_5.png)
User Media in JamSpace

Of course, PeerJS can only share audio data between peers if there is an available source of audio data to share. By using the browser built in Get User Media API, it is possible to accept input from microphones or other system audio input devices. At first, JamSpace used the browser navigator object to directly get user media, however this was found to be deprecated, and was then updated to use the newer functionality that was accessible with the newer Media Devices API. This API has better support for modern browsers and has more features than the old API. [Mozilla, n.d.]

The user media is seen as a stream of audio in the case of JamSpace. This audio stream is then used in PeerJS to stream it to the other connected peers. Another benefit to the new user media API was the ability for it to interface with the HTML5 canvas, allowing for an audio visualizer to be drawn on screen.

It also offers functions for recording the user’s stream. The way this is implemented in JamSpace, the media stream is saved in chunks as it is recorded in real time. When recording is finished, the chunks are all saved into one final audio blob file, that is then kept in memory of the local browser session. Once recorded, the audio recording is available in the room view under the tracks list. Due to time constraints, these recorded tracks are only available locally, and not streamable by other connected peers.

To make tracks available to all peers the PeerJS connections could also double as data connections, where each newly recorded track is sent to all currently connected peers and added to their track lists, and any previously existing tracks are requested from the previous peer by newly connected peers to update their own listing of tracks.

However, if the tracks were to be made editable by all connected peers, sort of like a communal digital audio workstation, then this might be cumbersome to constantly send track data across all connected peers. As such it might be better to have a system in which the tracks are stored on the server and streamed to users in the corresponding room. Edits are then made by the client, sent to the server, and reflected in the streams of audio to clients. Saving the streams on the server would require use of libraries such as GridFS which allows files larger than 16 MB (the default limit in a MongoDB database) to be saved. It accomplishes
this by dividing the audio blob into chunks, and so when this file is operated on, it is reassembled by the GridFS driver and returned. It can also “skip” to arbitrary segments of the audio file in storage, thus allowing for full file streaming from the database. [MongoDB, n.d.b]

Due to the nature of JamSpace, user recorded audio files can vary wildly in size, so in order to be able to store them properly, it would be necessary to use GridFS to stream audio files from the server to the clients. This would also prevent client side memory bloat from having too many tracks loaded in at once if the tracks were to all be held on the client’s side. This would also allow for future proofing of JamSpace as GridFS is conducive to the client’s ability to edit tracks and have these edits be available to all users.

When a recorded track is ready to be shown to a user, it is added to the track list in the room view as an HTML audio element. Currently, it is simply the basic web browser media player, so it doesn’t fit very well into the UI flow. However, given more time, it would have been possible to use JavaScript libraries to change the appearance of the player and to even add controls.
**JamSpace Results**

JamSpace was successfully built on the MEAN stack, which uses NodeJS and ExpressJS at its core to handle routing and serving of files, MongoDB to store user and room data, and AngularJS to extend client side functionality. It makes use of middlewares such as PassportJS for user authentication, and Mongoose for MongoDB operations. Finally, it successfully implements PeerJS and the web browser’s built in Media Device API to share audio streams in a peer to peer connection and to record user streams into audio clips that are able to be played back to the user. The only feature that could have been built on to make JamSpace really stand out would be the track recording ability, as in the current state, the audio is only recorded and saved locally. As discussed in previous chapters, it would be ideal for JamSpace to have these audio tracks recorded and saved onto the server, to then be streamed to clients so that edits and changes can be preserved across all users more easily.

The front end code is divided into respective views and controllers, and the entire application is injected as one single page Angular app. Please refer to appendix 2 in the Appendix section included at the end of the report to view the full listing of the files in the source code.

The code for JamSpace can be divided into two parts. The server side code, and the code that is served to the client for front end views and functionality. The server side code is all contained in the “server” folder, and the files within are responsible for all of the logic pertaining to file serving, user authentication, PeerJS communication, and database functionality. The “public” folder contains all of the client side code, which is served to web browsers and displayed for the front end. It holds all of the HTML views, CSS stylesheets, AngularJS controllers/services, and images.
FIGURE 6 - JamSpace Operation Flow Chart
As shown in figure 6 above, the code for JamSpace is organized into several pieces. The control flow between all of these pieces is segregated into different parts for easy organization and modularity. The views are the HTML files served to the client, they are the final GUI of the application and provide the user with simple controls. The Angular controllers extend this functionality and also implement the PeerJS connectability in the Room controller. The services are used as an abstraction of server functionality and allow the controllers to interface with the server without directly exposing server logic to end users. The route handler on the server side is responsible for then receiving requests from the Angular services and dealing with them appropriately. The routes code is where all of the database operations are handled, as most of the routes handled within deal with data being POSTed from the client, or with data being asked for in a GET request from the client. If data is sent back in a server response, the Angular service will then handle that as well by passing along relevant data to its corresponding controller.

The db and models code are responsible for local MongoDB and Mongoose information. The db contains the link to the MongoDB database, and the models folder contains the Mongoose Schemas for the pieces of data that are being stored in the database. JamSpace makes use of two custom schemas, users and rooms. The user schemas contain fields for a user's username and password, and the room schemas contain fields for the room’s name and list of usernames in it.

The server file is what includes all of the middleware and is the main file that is executed when creating an instance of the JamSpace application.
Using JamSpace

Refer to Appendix entry 1 for installation instructions.

When a user first enters JamSpace, they will be greeted with the home page, where they are presented with the options to Log in or Sign up. Clicking on either of these buttons will take the user to the respective page in which they enter their information. Shown in figure 7 below.

FIGURE 7 - JamSpace Homepage

The signup and login screens are both similar in appearance. Shown in figures 8 and 9 below respectively. If a user decides to sign up, they must use a unique username or else their request will not be submitted. If a user decides to log in, they must enter a valid username and password combination in order to be allowed to log in.
FIGURE 8 - JamSpace Sign Up Page

FIGURE 9 - JamSpace Login Page
Once a user has successfully logged in or signed up, they will be presented with their user profile page, pictured in figure 10 below. From here they can click the View Rooms button to be taken to the Rooms List page. They can also click the Logout button to be returned to the homepage and end their session.

**FIGURE 10 - JamSpace User Profile Page**
The rooms list page pictured in figure 11 below, shows a list of currently available rooms that the user can join. To join a room, click the Join button on the desired room listing. If there are no rooms to be joined, then the user can create a room by entering a room name in the text input box and clicking the Create Room button. Both joining and creating a room will take the user into the room view for that particular room.

FIGURE 11 - JamSpace Room List Page
The room view is the main part of the JamSpace application, pictured below in figure 12. It is made up of three panes. The user list pane, where the connected users are all shown. The recording controls pane at the bottom, where the recording controls are shown, and the track list pane, where recorded tracks are shown.

To mute a user, the user can click on the blue mute icon button beside a user’s name to stop the audio from their stream.

To begin recording a track, click the Start Recording button in the track control pane. Once clicked, any audio being played into the microphone will be recorded to a track. Recording state is pictured in figure 13 below.

To stop recording a track, click the Stop Recording button in the track control pane. Once clicked, the audio stream will stop being recorded, and will be shown as a web player audio clip in the track view, pictured in figure 14 below.

To leave a room, click the Leave button at the bottom right. This will take the user back to the list of rooms.

FIGURE 12 - JamSpace Room Page
FIGURE 13 - JamSpace Recording

FIGURE 14 - JamSpace Track Recorded
Future Work

Overall, JamSpace stands as a proof of concept to show that having a place where users can connect to each other over the internet to share audio with one another. Given the time period in which it was built, it is feature complete in regards to what was proposed, but there is a lot of work to be done in order to make it fully featured and to make it stand out from other internet voice chat services.

The following features should be added in future JamSpace iterations:

More robust audio options, in order to make JamSpace closer to a simple Digital Audio Workstation where musicians can communally edit their tracks. This would include the ability to change simple things like a track’s volume, panning, compression, and other traits in the overall mix. It should also include options for users to upload their own tracks, for example: a drum beat or a song to play along to.

More control over a user’s profile, so that users can represent themselves more accurately. The ability to link accounts with other websites such as Facebook or Twitter, to add avatars, and to set a favorite instrument. Additionally, to make these profiles visible to others, users can set their profiles to be visible to other users if they desire.

The favorite instrument feature would play nicely into another interesting feature that would really make JamSpace unique. A ‘matchmaking’ service in which a user (or users) seek to form an ad-hoc band, by either creating a room and seeking out a specific instrument for a user to fill, or for a user seeking to fill a room’s specific instrument role. This could be an interesting way for users to network with each other and to form fun bands, or to challenge themselves by exiting their comfort zones and playing with strangers.

In the end, JamSpace was a successful foray into the world of full stack web development and web based communication systems. To make it stand out as a unique web application, given enough time, it can definitely have some work put into it, but as it stands it proves that it can be done.
Resources and Citations

Documentation

Bower. (n.d.) Retrieved from https://bower.io/
DuoBango. (n.d.) sipML5 and webrtc2sip Retrieved from https://www.doubango.org/
Google AngularJS. (n.d.) Retrieved from https://angularjs.org/
MLab. (n.d.) Retrieved from https://mlab.com/
MongoDB. (n.d.a) Retrieved from https://www.mongodb.com/
ReactJS. (n.d.) Retrieved from https://reactjs.org/

Citations


Appendix

1 - Installation Instructions

1. Install NodeJS
2. Install Bower
   a. “npm install bower” in directory containing “server.js”
3. Install all node packages
   a. “npm install” in directory containing “package.json”
4. Install all bower packages
   a. “bower install” in directory containing “bower.json”
5. Dependencies are all installed now
6. Run JamSpace by typing node server.js in directory containing “server.js” and going to localhost:8080 in the web browser
7. Note: Only tested on Google Chrome browser, functionality can’t be guaranteed on other browsers
2 - Source Code Listing