ForeverBoard

ALEX PIPES

Faculty Advisor: Sotirios Kentros
Presentation Date: May 2nd, 2019

Student Objectives:

Going into this project I had spent an internship working on both the front and backend at a company. I found a lot enjoyment working on the front end and wanted to focus my project around using modern technologies to build more than a static web page. I also wanted to include some live networking and server push notifications in my project. I believe these two skills align with my interest and will give me a leg up going into the employment. To develop this, I wanted to use React.js, as I know it is a popular framework, and socket.io for live networking because I know it used in industry and it allows me to keep the whole stack in JavaScript.

Problem Specification & Requirements:

I would like to have infinite online white boards, that are live updating, and shareable by url. I want users to be able to free hand draw, while having other users see the updates live, without refreshing the page. I like to think of it as a mix between Google Docs and Microsoft Paint. This idea can be used in a classroom, between friends chatting, leaving hidden messages on undiscovered URLs, and sharing community drawings. The user should be able to click and drag anywhere on the whiteboard and then all other users at the same extension should see the update in a very short amount of time. Going to the same URL you have been to before should still show the drawings that were left there. Going to a URL someone else has been too should show you the drawings they left there.

To use, you should be able to just go to a common hostname at any URL extension (foreverboard.eastus.cloudapp.azure.com in my case). Then you should be presented with a white board that you can draw on. To draw on the white board, pick any spot in the white space, left click for a starting point and then drag. Pretty simple! You can add any extension you can think of (less than 255 characters) to that URL to identify it and it will bring you to that specific whiteboard. If someone has been there before, you'll see what they left (if they left anything at all). If someone is there currently, you might catch them drawing and be able to view it live as it is happening.

Requirements:

- Free hand drawing
- Live updating
- All URLs valid
- Drawings remain FOREVER unless cleared by the user
- Drawings remain FOREVER even if server goes down
- Must be fun!
- Available on the internet

Solution Processes and Design:

Taking the problem specification and breaking it down into it's technical details led to a few immediate architectural decisions. Those are that I need a server that can have many clients connected to it at once, the server needs to be able to send and receive from all these clients, and there needs to be a database to preserve the drawings for each URL extension that can be visited. Also Each URL can only have 1 drawing on it, any new lines made will only be added to the existing drawing. It was clear I had to work on multiple pieces. This can be broken up into a client piece that allows users to free hand draw in the browser, distributed at all URL extensions by a UI server, a server to handle client socket connections and live drawing notifications, and a database to store the drawings. The relationship between these pieces is shown in **Diagram 1**.

To begin, there is the client browser piece. This is the React.js component of the project. Most of the logic for the React piece is in a single file. ForeverBoard.js is where the drawing takes place and it uses both react and socket.io to be able to draw and send the drawings. On mouse down, a line the user is drawing starts. A line in data is represented as a list of points. A full drawing is represented as a list of lines. The data representation of a drawing is shown visually in **Diagram 2**. Once the user has mouse-down a new line (list) is

appended to the drawing (list of lines). While the mouse moves each point it moves over is saved as a point, or a map of X and Y, and appended to the latest line. When the user releases the mouse any more mouse movement actions will not be added to the latest line. A new mouse down action would cause a new line to be created.

The next challenge is to translate this data representation of a drawing into something visual the user can see. Due to the regular format of SVG tags, it is the picture format that was chosen for this project. In the SVG (Scalar Vector Graphic) format each line is represented as a path tag. In the path tag the beginning point of a line is labeled with an "M". Then each point the mouse has moves over join the "X" and "Y" coordinates of the mouse with an "L" character. This is essentially the same as saying "move to" the first point then "line to" all of the following points. This means the drawing is represented as a line between each pixel. With that in mind it becomes straight forward to translate our data structure to an SVG. Each of the lines becomes a path tag. Then we iterate of each of the mapped points and put them into text separated by an "L" character, besides the first point indicated by an "M" character.

This is where React became extremely useful. Though I have a data representation of a drawing and a way to translate this, the next challenge is how to translate this while a user is drawing to give continuous drawing feedback. React's render function solves this exact problem. Putting the list of list of lines (the drawing) as part of a react component's state and then returning html from the render function of the component meant that when the state is updated the view will be re-rendered. Putting the translation from data structure to SVG tags in the render function meant that every time we update our drawing state (once per pixel while the user is mouse-down) the react-component will re-render, meaning re-generate the svg, the new state on the user's machine. This gives the user the smooth feel and feedback of drawing on a screen.

Before moving off the drawing piece, I want to note that this html and javascript page has to be served for all URL extensions. Writing a small node.js server that routes all requests to the index.html file without redirecting them from the request URL solves this problem. I used express.js to handle this routing issue.

With the client side drawing problem solved, it is time to move on to the live updating aspect of the project. When a user finishes drawing a line, in other words, the mouse up event received, the client will update its local line state and also send a message to the socket server with the last line it drew. The logic to receive this notification is kept in server.js. The server receives this line notification in data format of an array of objects containing an X and Y attribute. The server then looks up what URL extension the request is coming from. With the line and the URL extension, it gets the currently cached drawing (possibly 0 if it is a new URL extension) and adds the line sent from the client to that cache, initializing it if needed.

Now that the server has an updated state of the drawing for the URL the client is at, it needs to update the state of the drawing for all clients connected at the same URL. I solved this problem by sending out a notification from the server that contained the URL extension and the newest line. Then the clients would compare the URL extension and decide whether or not it needs to append that line to it's local state.

This works for lines that are drawn while you are on the website but does not solve the problem of visiting the website where a drawing has been before you were there. To solve this, I added a notification from the server that forced the client to do a full refresh and was sent to newly connected clients. This notification contained the URL and an entire drawing (not just a line). When the client receives this notification, if the URL is the same as the one the client is on, it will completely replace its state of it's drawing with the one the server sent. Doing this made the clear button straight forward. It was a special signal to the server that cleared the server's state of the drawing, then server would just re-use the new-connection full refresh signal to update everyone connected.

At this point clients connected at the same URL will see the entire drawing stored by the server for this URL and can see live updates. During my implementation, the state of each URL and drawing was kept in a map in memory. This means that if the server goes down all drawings are lost. To fix this problem it is better to move the map of URL to Drawings to an external database. This means that if the server goes down no drawings for URL's will be lost. Due to the way I implemented the access and updating of the map, it was easy to change the setter and getter functions to point to a database instead of a global map.

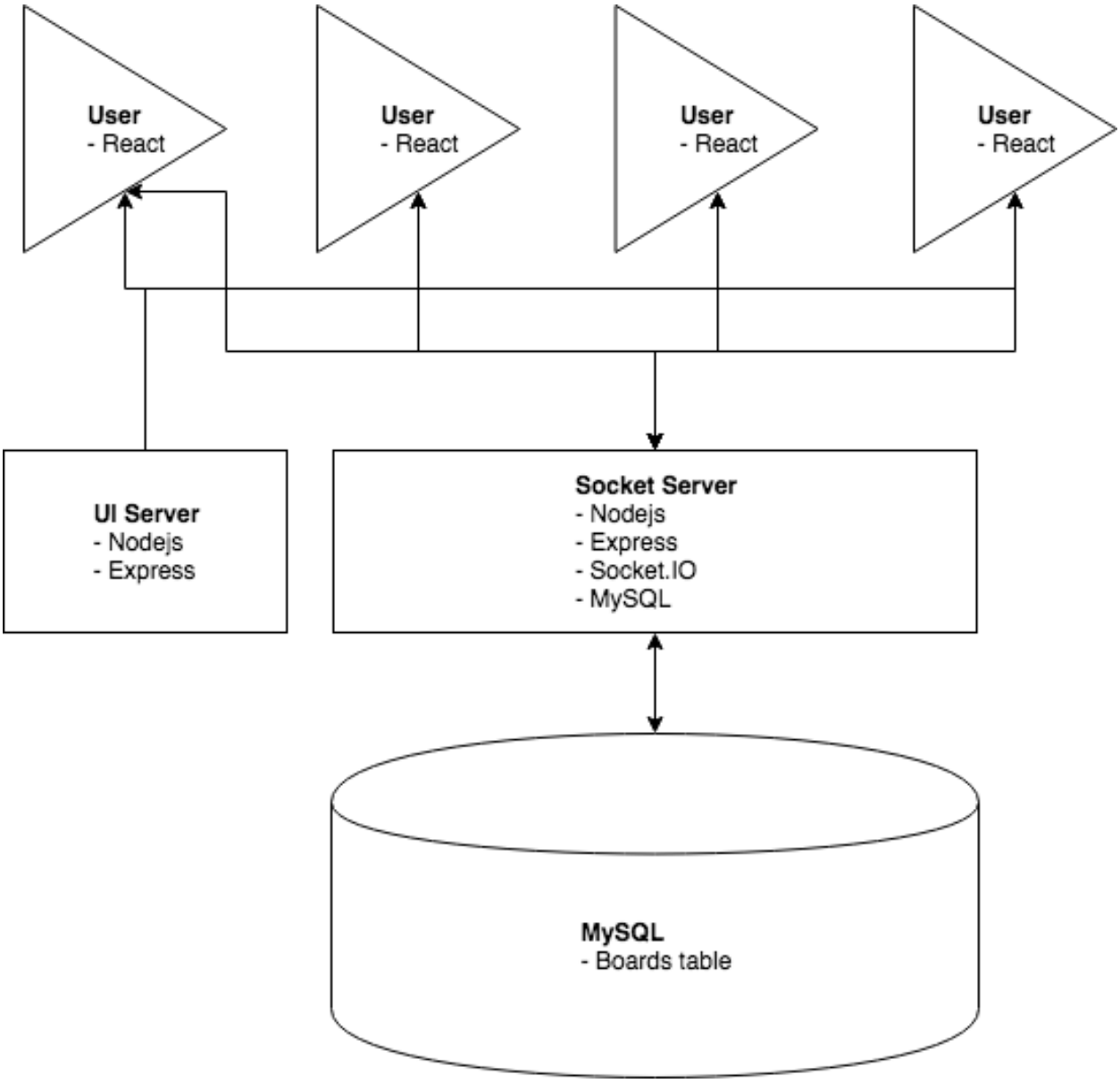
Currently the database has just a relationship between the URL and the Drawing. The drawing is stored in JSON. This is shown in **Diagram 3**. This JSON is stringified when setting the column and serialized when accessing it. Though the relationship between the URL path and the Drawing was sufficient for solving the problem, the complex data layout of a drawing itself can be separated across tables. That would make it much easier to update how a drawing is represented. For example, if you wanted to add color and it changed the data structure of a drawing, the entire database would need to be cleared since it would not serialize correctly anymore. If it was spread across tables this could be solved by adding a new column and defaulting the value to black.

When this was done being implemented the project should function as the project specifications declare. There are ultimately two main actions one can take on a white board, either draw a line or refresh the drawing. Clearing a drawing and connecting to a whiteboard are both variations on refreshing a drawing with the added detail that clearing will refresh it to nothing. The flow of these actions and how they are propagated from one client's computer to all connected users is shown in **Diagram 4**.

This goes through the implementation of the four main pieces for my solution. These four pieces are the in-browser drawing, the server that routes all URLs to the same html page, the socket server that responds and pushes notifications to the client, and the database that stores the current state of a drawing for each URL. With all of these implemented I reached a solution that sufficiently solved the problem proposal.

Diagrams

Diagram 1 Architecture Diagram:



Note: The connection between the UI server is only in one direction but the communication from the Socket Server to the clients is bi-directional. The Socket Server can push messages to the client.

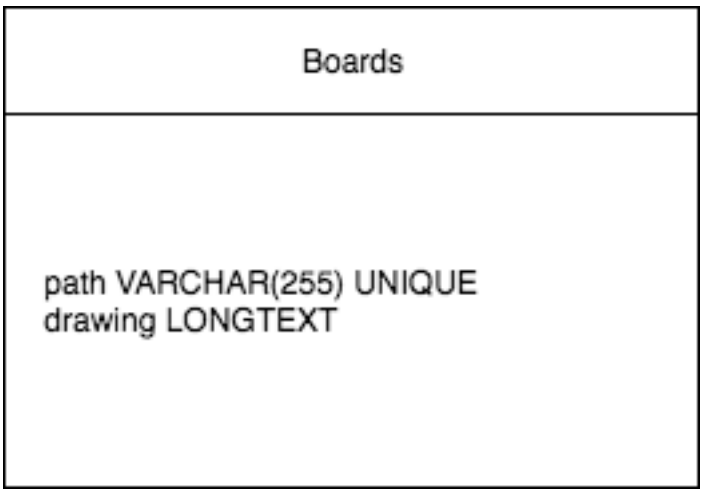
Diagram 2:

Data Structure of a Single Drawing

	Lines				
Line:	x, y	x, y	x, y	x, y	x, y
Line:	x, y	x, y	x, y	x, y	x, y
Line:	x, y	x, y	x, y	x, y	x, y
Line:	x, y	x, y	x, y	x, y	x, y
Line:	x, y	x, y	x, y	x, y	x, y
Line:	x, y	x, y	x, y	x, y	x, y

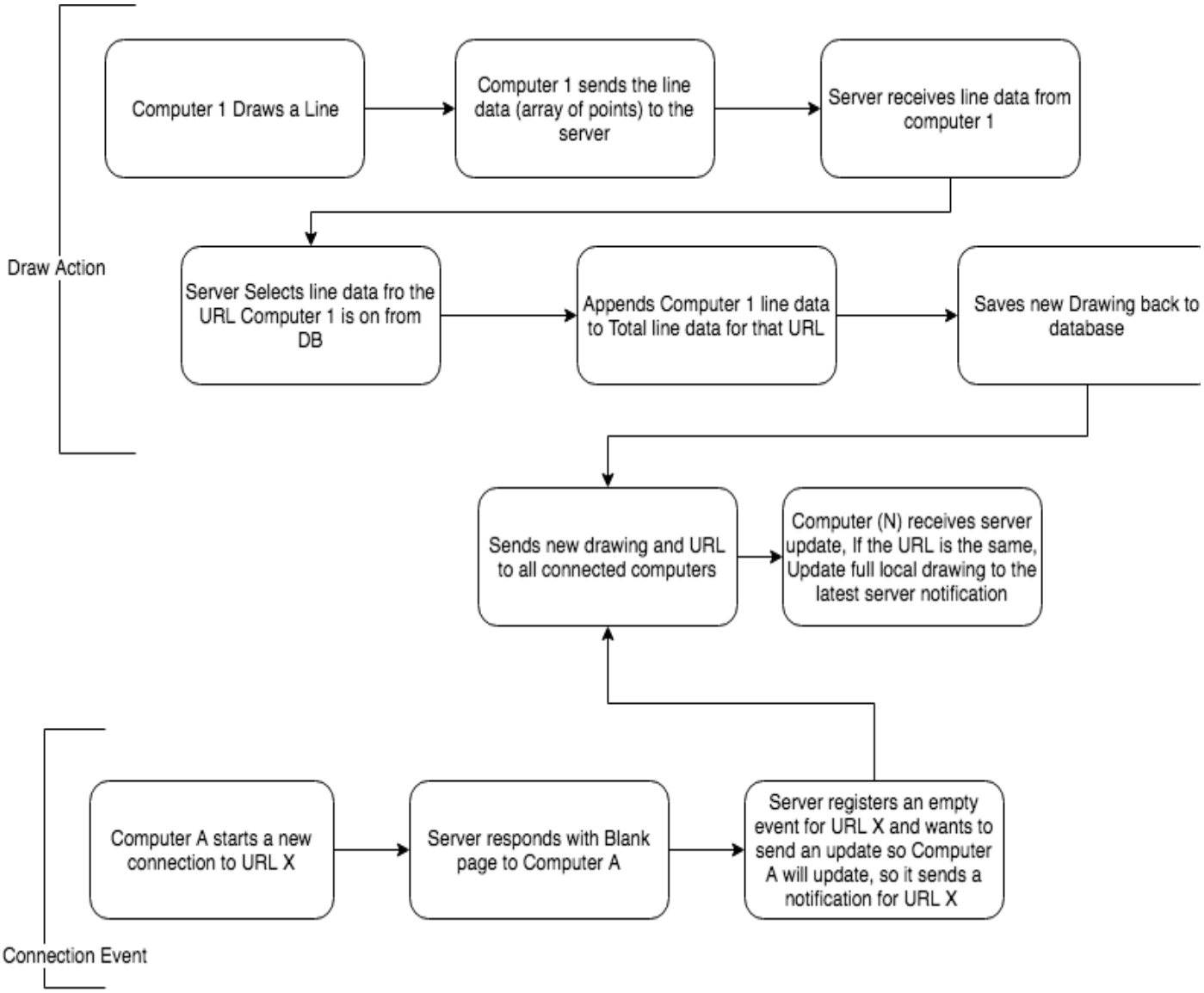
Note: On the client side, the drawing is a list of lists of maps. On the server side, it is an array of arrays of objects. This is because the socket.io library was converting our data structure. It didn't make sense to convert the drawings server side, because they would just be converted back to an array of arrays of objects when we send it back to the client.

Diagram 3 ERD:



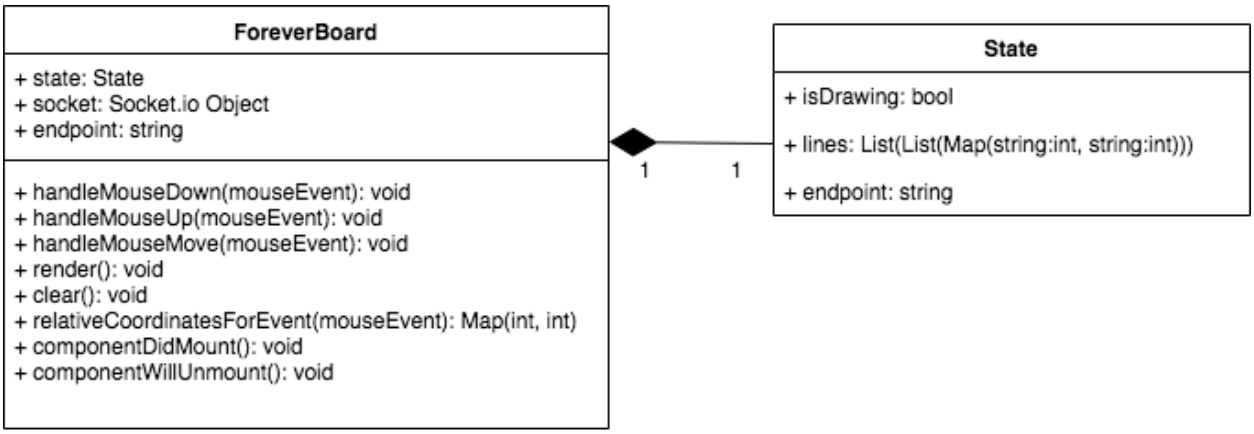
Note: This is my database schema. Future changes will have to be made to give the data representation of the Drawing their own table. What's important though, is the relationship between a path and a drawing. Path is a unique column, meaning that each Path can have exactly 0 or 1 Drawing.

Diagram 4 Flow Diagram:



Note: There are two main actions a client can take. It can either draw a line or be a new connect. A line drawing is an incremental notification but a connection event contains the entire drawing. Clearing the picture is equivalent to the server clearing its table then sending a reconnect event to everyone.

Diagram 5 UML Class Diagram:



Note: See here that Server.js functions are absent. This is on purpose as I am using an external library and am not aware of all of the members and functionality. The way I used the socket class was to add functions to the socket.io signal map. The signal map takes string signal names and maps to functions run when that name is received. Most of Server.js. which contains a bulk of the other half of the work, is implementations of functions to be put in the socket.io signal map.

Tools List:

- React – JavaScript UI library. Used to generate SVGs and create the UI.
- Socket.io – JavaScript library to communicate between user and server both directions in real time. Used for live updating of the drawings.
- Node.JS – Allows me to run JavaScript on my local computer. I used this for my webservers.
- Express – JavaScript library to handle browser requests. This is an alternative to PHP when running on node.
- MySQL – Used to store drawings incase the server goes down.
- Azure – Used for virtual machine deployment to be accessible by the web
- Ubuntu Command Line – The virtual machine we set up on Azure was running Ubuntu and I accessed it using SSH
- SSH – Remote access tool used to run commands on remote machines
- Visual Studio Code – Text editor with IDE-like features used to write and develop JavaScript
- Git – Used for versioning the software. Often used it to roll back to points after experimentation
- Github – Used to store the code with the git history
- SVGs – The format in which the drawing is shown in the browser
- JSON – The format we used to store the data required to draw the SVGs in the database

Project Journal (Time Schedule and Benchmarks):

Beginning of February

When I first started the project I wanted do the servers first. I believed getting the browsers to live update would be the hardest. Early on I knew I wanted to write the project in

one language to keep it simple. Since the browsers can only run JavaScript this locked me in to JavaScript for the backend as well. This meant learning node from the ground up.

It turns out node is a simple tool that allows someone to run JavaScript outside of the browser and came pretty naturally. This was also my first exposure to a package manager (npm). I used this to download socket.io for the server, which I knew I needed since I wanted the project to be live updating. Once I had node and socket.io working I ran into my first problem. What is a drawing?

I didn't know how to write the functions to send and receive drawings when I didn't even know what a drawing looked like. This was my first full pivot. I had to change into developing the browser side drawing first, then figure out how to send that to the server.

This started my first venture into React. I knew I wanted to use that as a front end framework as that is what they used at my internship. I started with a simple react page tutorial with a button on it. It was cool because react watched the files I was changing and automatically updated my browser to show my changes. This was very gratifying even if it was simple.

End of February into mid March

Now that I had react working on my machine I wanted to make the free hand drawing. I did some searching on how to make free hand drawings in the browser using React. I was able to find a good tutorial on how to generate SVGs in the browser. This was a great starting place and after I had finished was able to draw lines in the browser.

With the lines drawing in the browser I wanted to start sending them to the server and then back to all clients. I had never sent data over a socket so I started a socket.io and React tutorial. This led me to creating a second page with 3 buttons on it. Red, Blue, and Change background. This showed me how to emit signals to and from the server and receive them on both as well.

End of march into mid April

Now that I had free hand drawing and a live updating app, I had to integrate them somehow. In order to put the live signals into the drawing app I had to become deeply familiar with the event handlers which we had done in class before. Ultimately, I wanted to send the last line drawn when someone let go of the mouse.

Since the data structure was complicated, on my first pass I sent the entire drawing on mouse up (all the lines every time, not just the last one). When the server got any data structure I sent it to all clients. I also had to write code on the client, if the server sent a drawing, to replace my drawing with the one from the server. There were plenty of errors when I first did this and I used the browser console to see logs from my code. This showed me why the drawing being sent back from the server wasn't being drawn on the client.

It turns out the Socket library was changing the data structure of the drawing. The way we generated the SVGs from the lines variable expected it to be a list of list of maps. When we sent it to and from the server it came back as an array of arrays of objects. I had to write some client side code to convert this (which was a very difficult one liner) and it eventually worked.

This was cool and the first time I saw drawings live update but it was totally inefficient and had some bugs. For example, if two people drew at the exact same time, one line would be overridden. So my next challenge was to only send the last line drawn. The server would have to only send the clients the latest line, making the drawing incremental.

This required me to become even more familiar with the drawing data structure. Eventually though you could draw a line, it would push only that line to the server and the server would push only that line to all clients.

End of April

Now I wanted to add the URL piece to my project. This meant sending new information to the server. I added the path as an argument and wanted the server to send that to only clients connected at that path. It turns out I am not familiar enough with how socket libraries work and it was much easier to send the line to all clients with the path it came from as an argument, and have the clients ignore lines with different paths.

After, I wanted to save the drawings for each path so I created a database. I wanted the database to be more complex but ran into some difficulties so to get things moving I created a single table that saved the entire drawing as JSON. This was supposed to be temporary but snuck itself into the final project and would be the next thing I change.

I also had to do a little work to get the clients to show any existing drawing for that path if someone left something there. I created a full refresh signal that gets called on connection and sets the current drawing to the full drawing. This is very similar to the original prototype I had. I also used this full refresh to clear whiteboards for testing but found that it was a good feature.

Project Post Mortem

What did I achieve:

A live updating white board that is accessible on the internet. Users can see the line drawn as soon as the other user finishes drawing that line.

- Free hand drawing – User's can free hand draw with only one color
- Live updating – Drawings update only when a line is finished
- All URLs valid - Completely Achieved
- Drawings remain FOREVER unless cleared by the clear button – Completely Achieved
- Drawings remain FOREVER even if server goes down – Stored in a MySQL database but the schema is not complex or extensible
- Must be fun! – Definitely fun.
- Available on the internet – Deployed to azure, Completely Achieved

What didn't I achieve:

When I set out for this project I had a grand idea of what I wanted the project to be. I found that I spent way more time focusing on the technical issues than adding new features. I wanted there to be colors and erasers but it turned out, I spent most of my time on the

networking aspect. When I first started the database, I had an interesting idea for a schema but I ran into some problems and had to cut corners. This makes the database not extensible at all.

- Colors – Can't be done with just a css change. Each line needs a new variable that is the color, which would be a data structure change. This would require some refactoring that I couldn't do in time.
- Erasers – An eraser can just be the color of the background, so this sits on top of that. This would constantly cause the image to get larger and larger. Removing points and splitting lines would be a difficult alternative.
- DB Schema – Time Constraints, the original schema wasn't as simple but I was never able to effectively store the lines

What Did I Learn:

This project taught me a lot about not only coding but designing and planning as well. I've done very minimal JavaScript before approaching this project but after this project I feel well versed and comfortable using JavaScript. I do want to work on front ends when I get a full time job. I realized I like seeing my changes as soon as I make them in instead of needing to save, compile, and restart the server. Though I used JavaScript for the client and server, I found working the client more gratifying.

Learning how to use all these different JavaScript libraries was super helpful and my coding confidence has shot up. I learned how servers and clients talk to each other. Also I learned how important an extensible database is if I want to make future changes. Going into this project I felt lost because I've never really developed anything this size or had any of these functionalities. Reading documentation was the best way I learned how to use these libraries and frameworks, but I also watched some videos. Hearing someone explain something in their own words can simplify a lot of things.

Over the course of the project, I learned networking with socket.io, modern front end development with react, deploying a server with azure, and not quitting when I'm struggling.

What Can I Improve (Project):

Some of the things I can improve on with the project are:

- Database – right now it is a single table which just includes the drawing. I want to change the database design where it can also hold the color for each line, as well as store the contents the line across tables.
- Customizable title – I feel like it would add to the white board and maybe even rules for future people who haven't used that white board.
- Colors – I want the user to be able to mess around with different colors and see what art they can make.
- Eraser – should be able to erase certain parts of the white board. This was a lot more complex than I thought because if I split a line that must be translated into two lines in the database.
- Login – have a login to remember what white boards you have changed and maybe access private white boards
- Private boards – maybe for companies or classes, could be used for interviews or tutoring online.

Where Can I Improve (Process):

- When I knew I wanted to store drawings in the database. Deciding on a data layout much earlier in the project would've saved me some server work, as well as define a better schema.
- I spent a lot time learning about what a "Stack" is before picking and truly settling on the technologies. Now that I am more familiar with actually implementing a stack and not just theorizing about it, I know more of what to look for when choosing technologies.
- Time management was something I struggled with a little. Sometimes I spent far too long focusing on something that didn't move the project forward (such as a small bug).

- I'm still trying to become more comfortable with how callback functions work in JavaScript. The database library I used was asynchronous and this was a little difficult to wrap my mind around. Becoming more familiar with that style of programming will give me a leg up in JavaScript.

Grading Schema:

- 10% Report
- 10% Presentation
- 20% In Browser Drawing Implementation
- 20% Live Updating of the Drawing Across Users
- 20% Persistence of Drawings in Database
- 20% Unique URL's contain Unique Drawings that Live Update Individually
- 0% deployment

List of Deliverables:

- Appropriately Commented Source Code – Vital source code locations specified in the README included in the Zip
- Documentation of solution – Included in this report
- Sample Output – Video included in Zip file
- Executables and/or Projects – I included all of the relevant JS files and a README that includes their location in the Zip and how to run them
- Presentation Documents – Power Point included in Zip
- Project Journal – Included in this Report
- Project Post Mortem – Included in this Report
- A List of What Areas of were not Completed – Included in this Report
- Presentation of the Completed Project – Presented May 2nd, 2019