

CA400 - Technical Specification



Created By Students: Munashe Lawrence Chifungo & Fiona Tuite

IDs: 16469764, 16389251

Supervisor: Monica Ward

Contents

Glossary	1
Abstract	2
Introduction	2
Overview	2
Motivation	2
Research	2
Design	3
1. Front-End	3
2. Back-End	5
Implementation & Sample Code	9
Problems Solved.....	25
Results	27
Future Work	27

Glossary

1. **React.js** - React is an open-source JavaScript library that is used for building user interfaces/UI components.
2. **Firebase** - Firebase is a BaaS (Back-end as a Service)
3. **Firebase Authentication** - Firebase feature that enables a user to sign up/login
4. **Firebase Firestore** - A Firebase real-time noSQL database that allows us to store data from our web app.
5. **Express.js** - A back-end web application framework for Node.js
6. **Heroku** - A PaaS (Platform as a Service) that allows developers to build, run, and operate applications from the cloud.

Abstract

Echo labs is a web application designed to virtualize the programming lab session. It's made for use on desktops, laptops and devices you would traditionally use for software development purposes. The main functions of this application are to allow users in the form of students and tutors to communicate with each other and write code together. During the pandemic it was difficult to assist students, in particular first year programming students during lab sessions. It was brought to our attention that this could be mitigated by a platform that allowed students to program in scenarios similar to that of an in person lab, except from their home.

Using the Echo Labs platform, students can begin editing code and answering questions from the lab. When they need help they simply request help at the click of a button and all the tutors are notified that a student is in need of assistance and a tutor can then join the students editing session. From there both parties can communicate with each other and the tutor can help the student. The goal of Echo labs is to ensure that students get help when they need it.

Introduction

Overview

This technical spec is a description of the process through which Echo Labs was developed. It contains an explanation of our motivation for developing this software, the research conducted before developments started and of course a detailed explanation of the project's design and it's implementation. Additionally, this document contains some of the problems we encountered during development and how we went about resolving them if they were resolved. Finally we discuss the results of our efforts and what we would do in the future to further develop this software.

Motivation

The pandemic was unprecedented, it resulted in the complete overhaul of various aspects of day-to-day life, no exceptions for students and academia. Modules that could be taken from home were taken from home and while this was effective for theoretical subjects, practical ones suffered greatly. While the use of communication software like Zoom and Microsoft Teams was quickly adopted and proved useful, these did not usually provide the functionality to effectively replicate the necessity of in person, practical lab sessions.

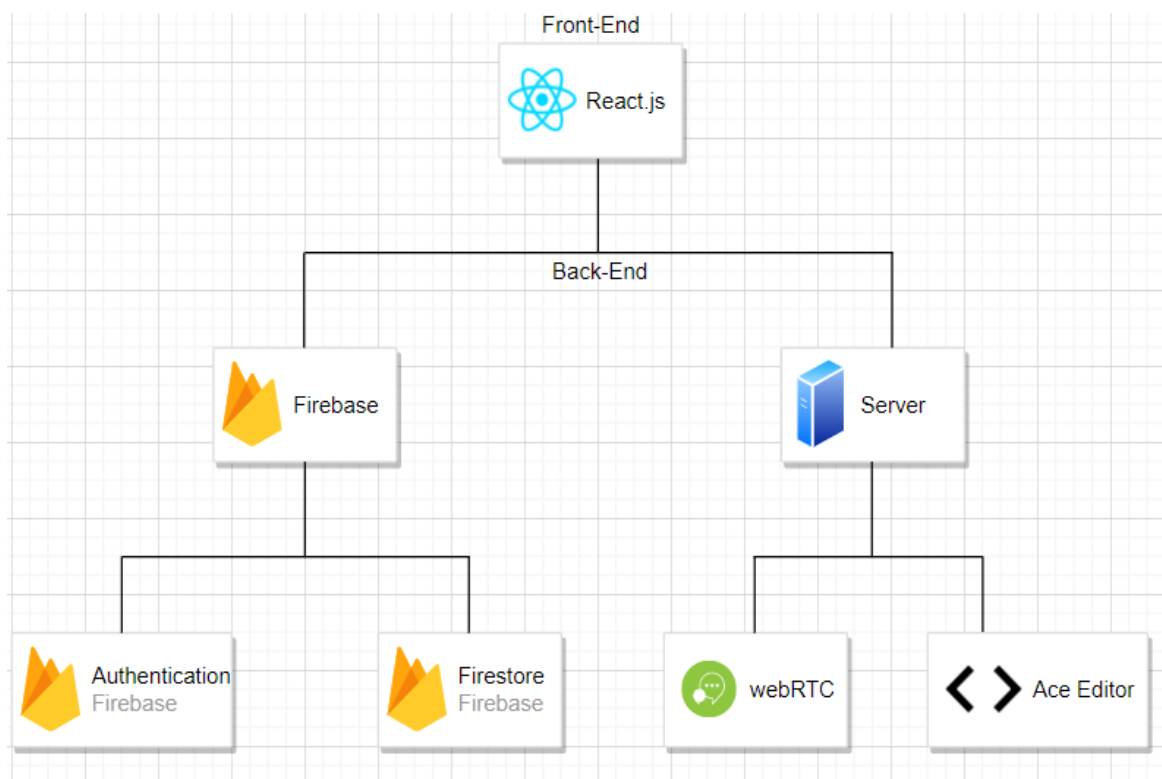
Luckily, for a subject like Computer Science, the only equipment you need to begin learning is a computer so lab work from home was more than possible. However, for beginners the tutelage you receive during labs is extremely valuable as well as communicating with your fellow students and discussing how a given problem may be solved. Thus, we decided to develop a web application that replicates the lab scenario for both students and tutors. Using the Echo Labs platform, it's possible to learn programming with your peers and tutors just an arm's reach away, from the comfort of your own home.

Research

Before we could attempt to accomplish our goal, there was a lot of research and planning that was necessary as we had not built an application like this before. We discussed the core functions of the application and made mockups of what the UI would look like.

Once we had our goals set out it was time to begin researching. We were unfamiliar with frameworks that we had decided to use. For our implementation we chose React for the front end of the web app, express for the backend and firebase as the database. We chose this stack as it would allow us to build a responsive and modular application. As well as this, our app relied heavily on utilising WebRTC for its audio/video communication. This too was something we had not worked with before. From here we began relearning Javascript, learning React, familiarizing ourselves with using Firebase and the fundamentals of building a communication platform that utilised WebRTC.

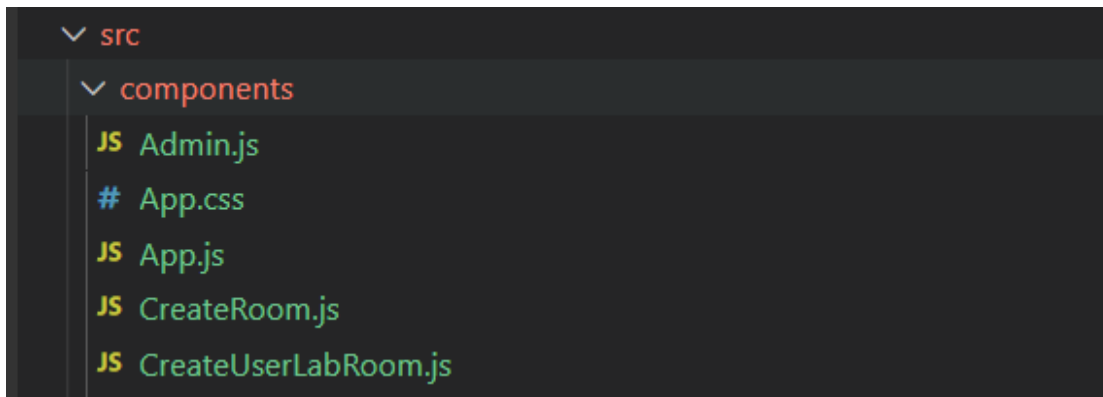
Design



1. Front-End

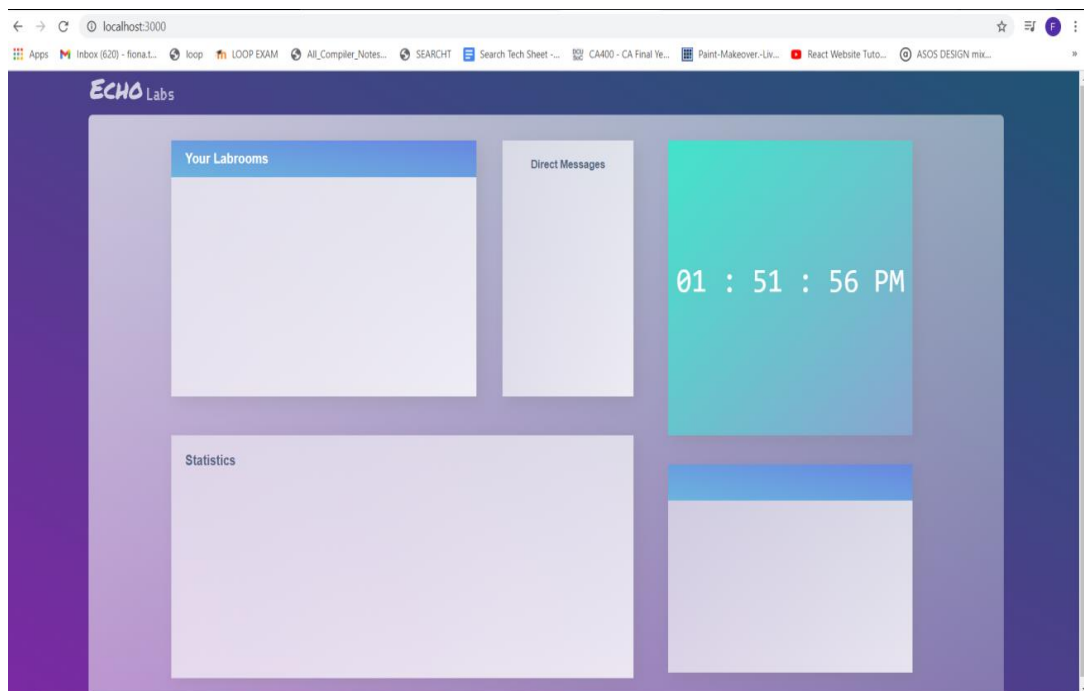
To implement the front-end of Echo Labs we used React.js which is an open-source JavaScript library used to build User interfaces and UI components.

We chose to use React.js because its whole purpose is to create web app components with ease. A component for instance can be a new web page, a button to be displayed on a page, a Bar chart etc. Below is a sample of our component's directory layout (FYI this is just a sample of the files inside it, it's not the full directory).



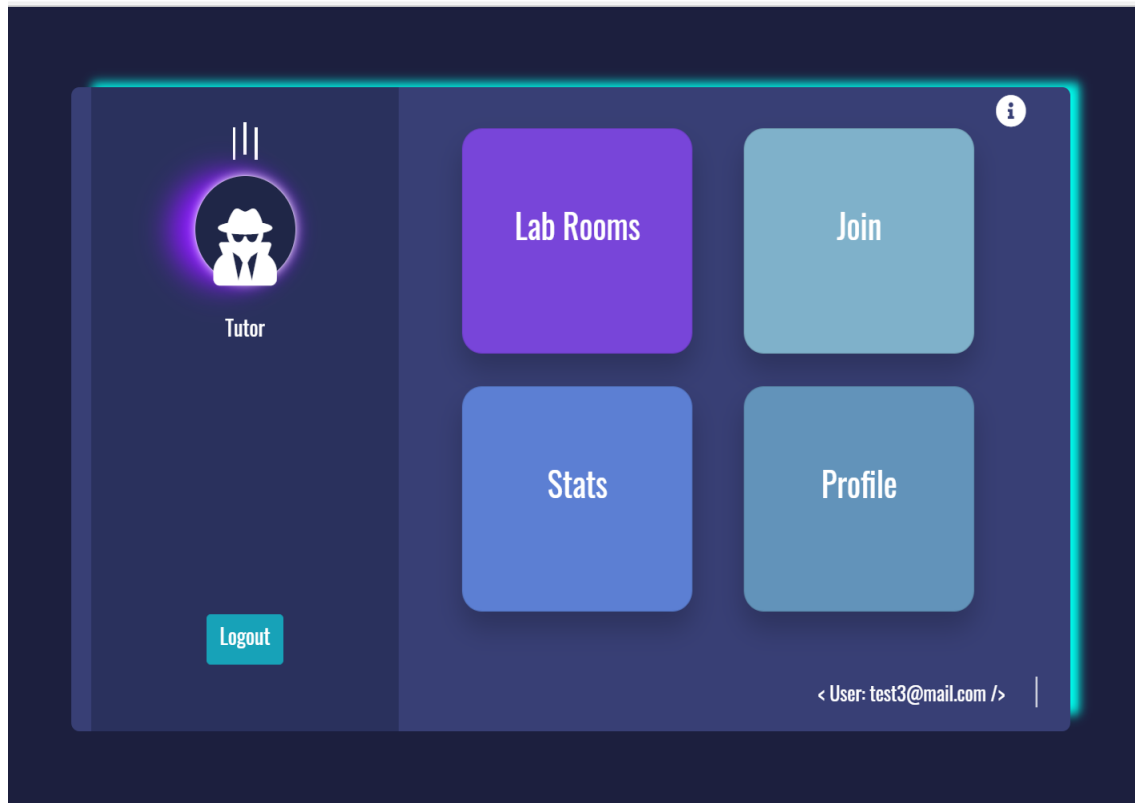
As well as that, React updates any changes made in real time . So if you initialise a new button, this will be displayed on your web app in real time, there is no need to reload the web app.

When designing Echo Labs we paid close attention to how a user may feel when navigating through the app. We came up with different design styles, below is our first dashboard implementation :



We figured this was not an ideal approach, every component here is crammed onto the user's dashboard. We believed if we continued using this approach the dashboard would end up looking very cluttered with no space for any new features. So for instance, if we displayed the users list of lab rooms under "Your Lab rooms" above, the list and its font size would be designed to be quite small so it would fit the lab rooms card. This is unfeasible as from what we learned in Web Design in second year it may make reading text quite difficult for the user and they may find it quite hard to use the application. Web design has taught us that we need to create a web app with simplicity, clarity, and accessibility in mind. Due to this, we decided a better approach was necessary.

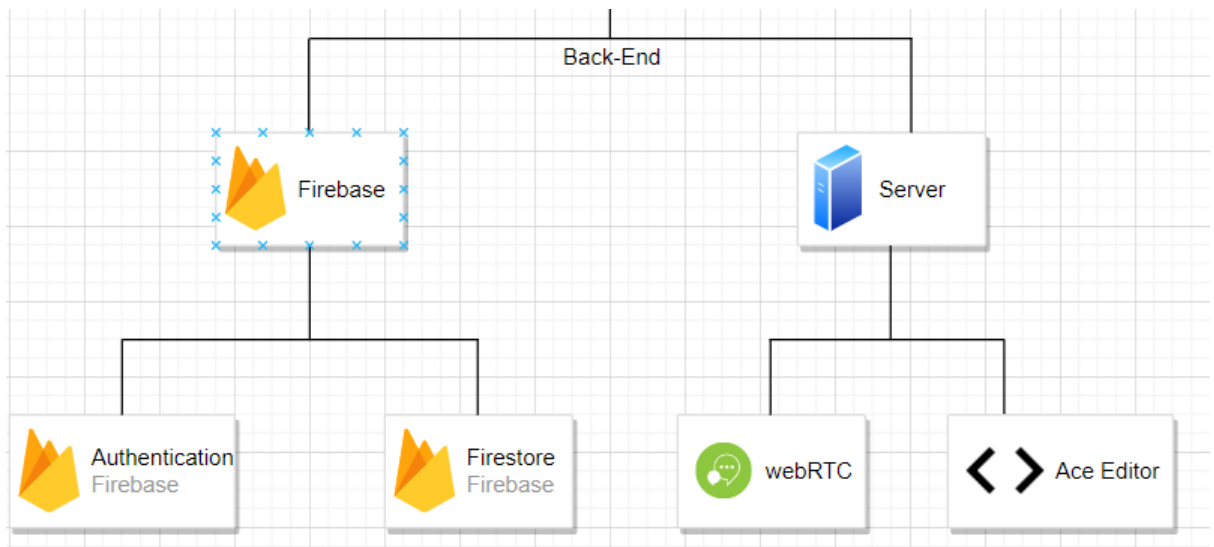
Below is our current Dashboard layout. The dashboard now contains cards that clearly display its purpose. Each card navigates to different functions of the web app:



As well as that, it displays a user's role (Tutor) and their email address in the corner. If they ever need help, there is also an icon in the right hand corner that explains how the app works.

2. Back-End

Our back-end architecture consists of the following features:



2.1 Firebase

Firebase is a BaaS (Back-end as a Service) with various features, the features we use are firebase Authentication and Firestore.

Firebase Authentication

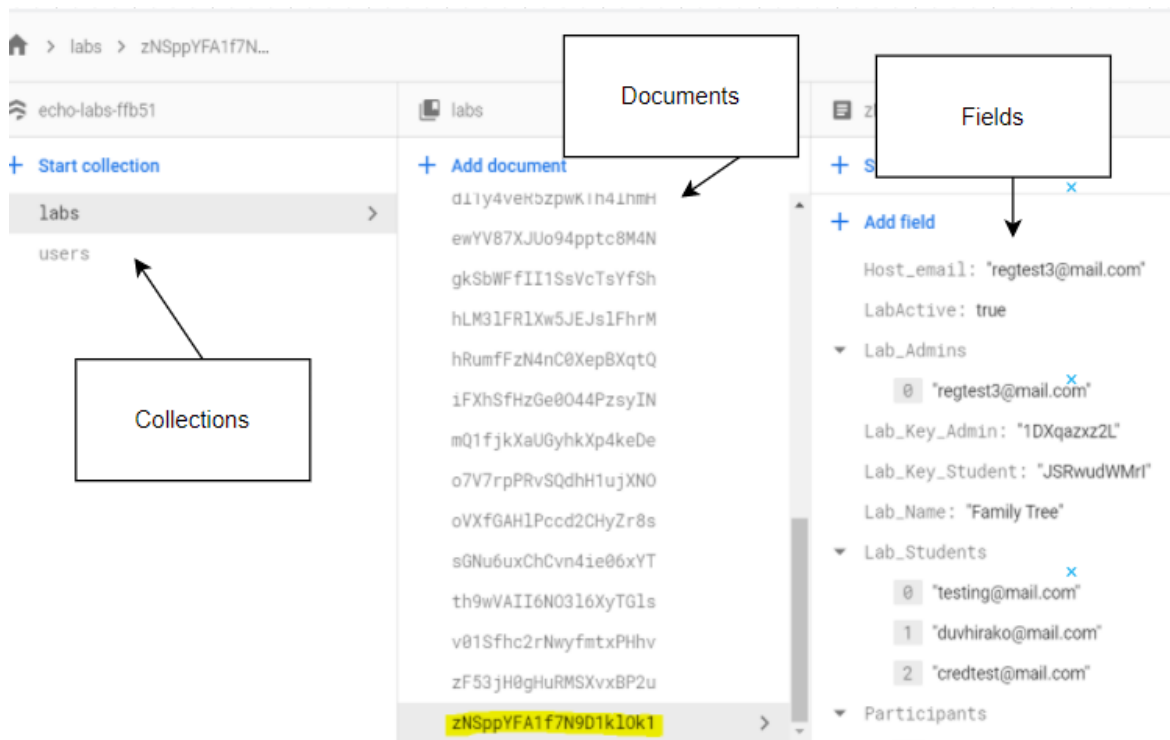
This allows us to enable an authentication flow into our front end of our web app. It allows a user to sign up, log in and log out.

<div> <input type="text" value="Search by email address, phone number or user UID"/> <input type="button" value="Add user"/> </div>				
Identifier	Providers	Created ↓	Signed in	User UID
dede@gmail.com		6 May 2021	6 May 2021	jtBzsXma6Udygs3sv6uLpk4yU8h2
robinonescu98@gmail.com		5 May 2021	5 May 2021	040xmDcmuQOUvy0BecACnp7n...
deku@gmail.com		5 May 2021	5 May 2021	bWIOOwVt03bQ97h78y1q8bjbe9F2

When a user signs up to our web app their email is stored as an identifier into firebase authentication.

Firebase Firestore

A real-time noSQL database - meaning we do not use tables/rows:

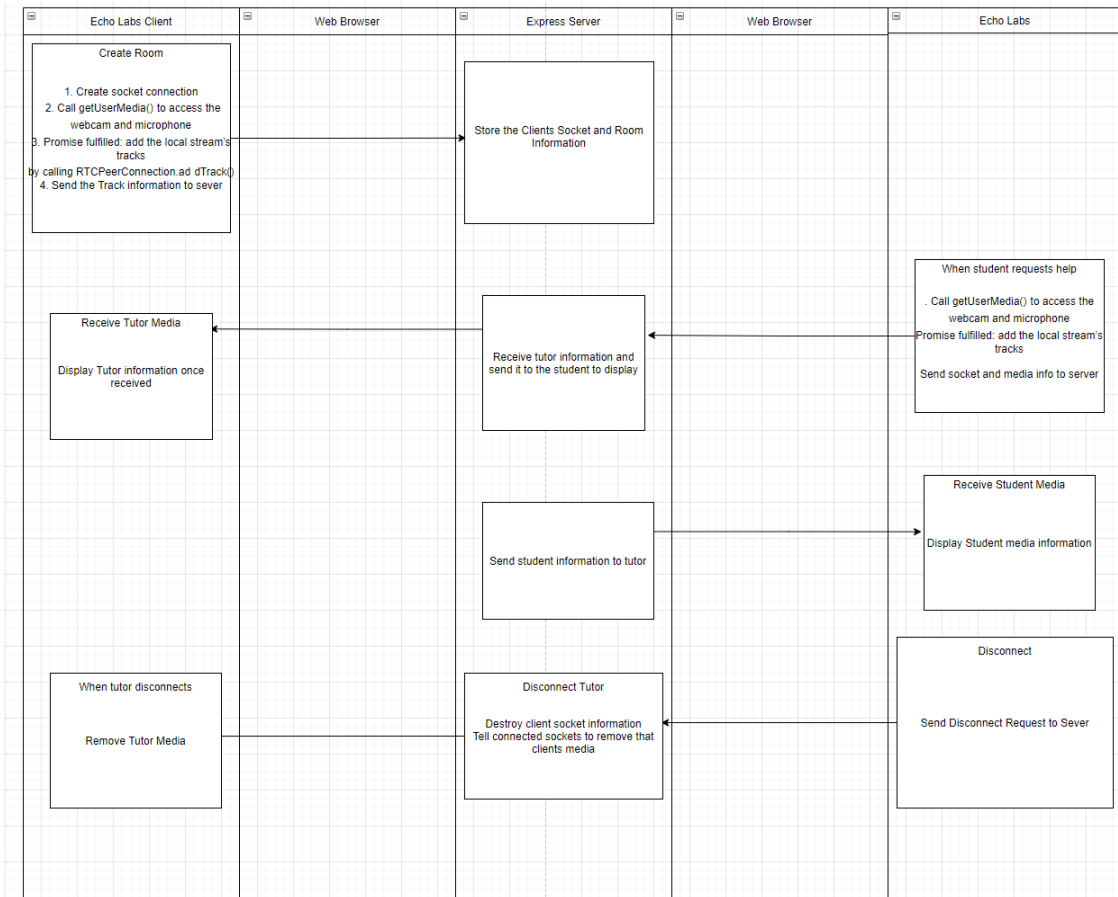


Instead, as you can see from the above image, the data is split into collections. Inside each collection, documents are stored (can be seen as a record). Each one of these documents are given a unique ID, and each document itself has fields inside.

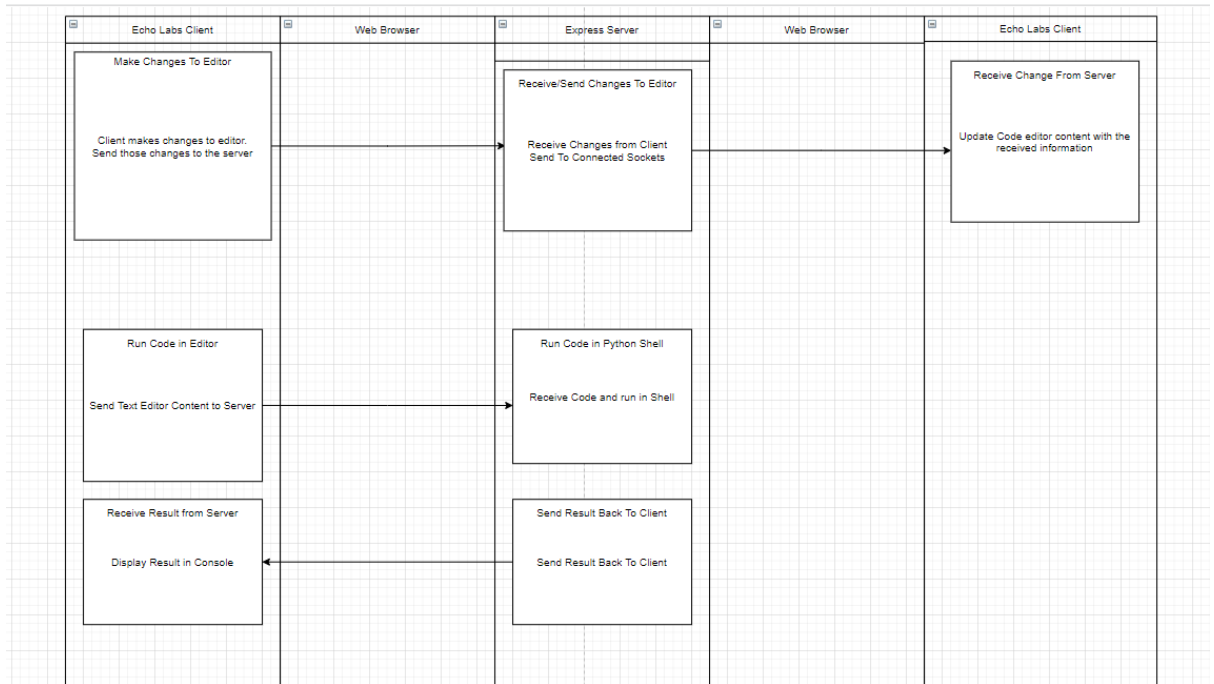
So, for example, above we are looking at the 'labs' collection which contains many documents. The document we are currently looking at is the highlighted document with an ID of 'zNSppYFA1f7N9D1k10k1'. This document contains fields such as Host_Email which maps to the data regtest3@mail.com and Lab_Students which maps to an Array with data inside.

2.2 Server

The server backend is built using Express. It handles sending and receiving media information between clients connected to the same room. As well as this it handles the real time code editor updates between students and tutors in the same lab session. Below is a diagram detailing how the WebRTC communication is handled and disconnected between a student and a tutor.



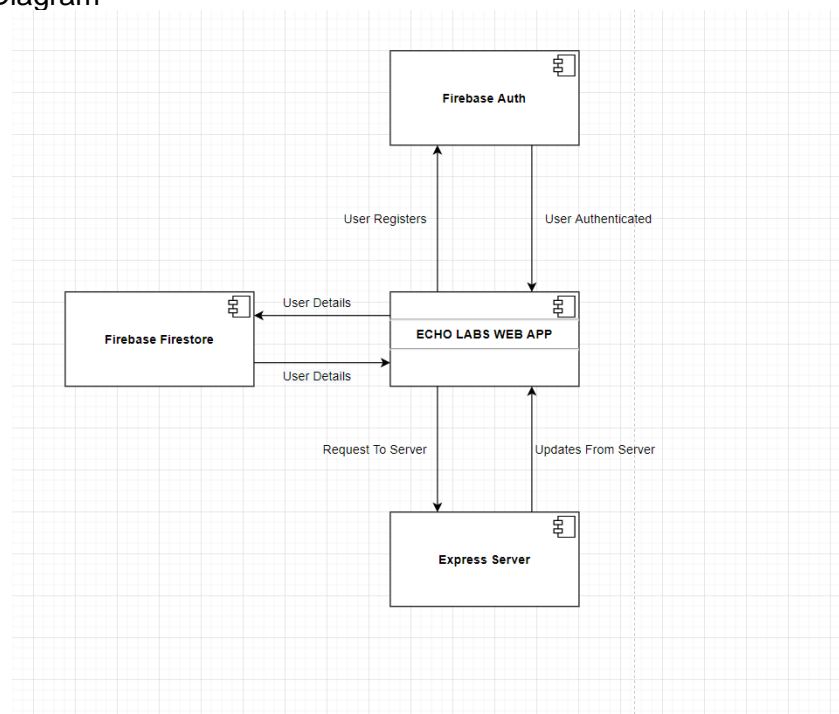
The code editor relies on the same server to send and receive changes. When a change is made the change is sent to the server and thereafter it's sent to the other connected clients in real time. The Express server is also responsible for running the python code being written in the code editor. When a user clicks run, the code is sent to the Express server where our python shell is running. The code is executed on the server and the result is sent to the client where it's displayed in the bottom terminal of the code editor. Below is a diagram that illustrates the communication between the server and the code editor.



The application is deployed using heroku where the backend server is run alongside the front end react components.

Implementation & Sample Code

Component Diagram



Account Authentication

For our application, we handled account creation using Firebase's authentication API implemented via a react context. The following functions are held in the auth context.

```
> function updateUserInfo(){ ...
}

|

//Context lets us share user data across multiple components
const AuthContext = React.createContext()

> export function useAuth(){ ...
}

export function AuthProvider({ children }) {

  //This state keeps track of the current user
  const [ currentUser, setCurrentUser ] = useState()

  // This state is used to check if a user is logged in, Set to true if logged in
  const [ loading, setLoading ] = useState(true)

> function signup ( email, password ){ ...
}

  // Login Function
> function login(email, password){ ...
}

  //Log out function
> function logout() { ...
}

  //Reset Password Function
> function resetPassword(email){ ...
}

  //Update email function
  function updateEmail(email){

    return currentUser.updateEmail(email)

  }

  //Update Password Function
> function updatePassword(password){ ...
}
```

The use of a context allows us to pass the authentication data throughout our entire application's component tree, without manually providing the props at every level. Once a user enters their details on the front end the sign-up function that's part of the Firebase API is run.

The image shows a 'Sign Up' form on a dark background. At the top, the title 'Sign Up' is centered in a large, white, sans-serif font. Below the title are three input fields. The first field is labeled 'Email' and contains the text 'test@mail.com'. The second field is labeled 'Password' and contains six dots. The third field is labeled 'Confirm Password' and is empty. Each input field has a small icon on the right side. Below the input fields is a large, solid blue button with the text 'Sign Up' in white. Below the button, the text 'Forgot password?' is centered in a light blue font. At the bottom, the text 'Already have an Account? Log in' is centered, with 'Log in' in a light blue font and 'Already have an Account?' in a white font.

This information is then stored in the user database and the user can then login with those same credentials on the login page. The following code details the information and functions that are sent to the children of our auth context.

```

useEffect (() => {
  // Executes the following
  //ComponentDidMount - Called the first time a comp is mounted
  // ComponentDidupdate - Called when comp is updated ( When a comp gets new props or the state changes)
  const unsubscribe = auth.onAuthStateChanged(user => {
    console.log(user)
    setCurrentUser(user)
    setLoading(false)
    updateUserInfo()
  })

  return unsubscribe //ComponentWillUnmount - Clean up code returns unsubscribe
}, []) // List Effect hook dependencies in the array, empty array means hook is only called one time

console.log(loading) // Check loading status of user

const value = {
  currentUser,
  signup,
  login,
  logout,
  resetPassword,
  updateEmail,
  updatePassword,
}

return (
  <AuthContext.Provider value= {value}>
    { /*if not loading don't render children*/ }
    { !loading && children }
  </AuthContext.Provider>
)
}

```

The functions and user information in value can be accessed via the useAuth default function. In our App.js file we wrap our components in our AuthProvider component which contains our auth context. As a result of our auth context provider the user's authentication status is known by all of the providers children.

```

import Requests from "../Requests"
function App() {
  return (
    <Container>
      <Router>
        <AuthProvider>
          <Switch>
            <PrivateRoute path="/create-room" component={createRoom}/>
            <PrivateRoute path="/room/:roomID" component={Room}/>
            <PrivateRoute exact path="/" component={Dashboard}/>
            <PrivateRoute exact path="/update-profile" component={UpdateProfile}/>
            <PrivateRoute exact path="/user-lab-rooms" component={UserLabRooms}/>
            <PrivateRoute exact path="/createLabRoom" component={CreateLabRoom}/>
            <PrivateRoute exact path="/join" component={join}/>
            <PrivateRoute exact path="/Lobby/:lobbyID" component={Lobby}/>
            <PrivateRoute exact path="/requests" component={Requests}/>
            <Route path="/signup" component={Signup} />
            <Route path="/login" component={Login} />
            <Route path="/forgot-password" component={ForgotPassword} />
          </Switch>
        </AuthProvider>
      </Router>
    </Container>
  );
}

```

This implementation is also useful for creating our private routes. The private routes above are pages that can only be accessed while a user is logged in. The private route function checks the authentication state of a user before allowing them access to a page. If a user is not logged in, they are sent to the login page via the useHistory hook. The following snippet details the private route function.

```

/*Prevents users from accessing content while they're not logged in*/

import React, { useContext } from 'react'
import { Route, Redirect } from "react-router-dom"
import {useAuth} from "../contexts/AuthContext"

export default function PrivateRoute({component: Component, ...rest}) {

  const { currentUser } = useAuth()

  return (
    <Route {...rest} render={props => {

      /*Needs to be expalined in more detail*/

      return currentUser ? <Component {...props}/> : <Redirect to ="/login"/>

    }}>

    </Route>

  )
}

```

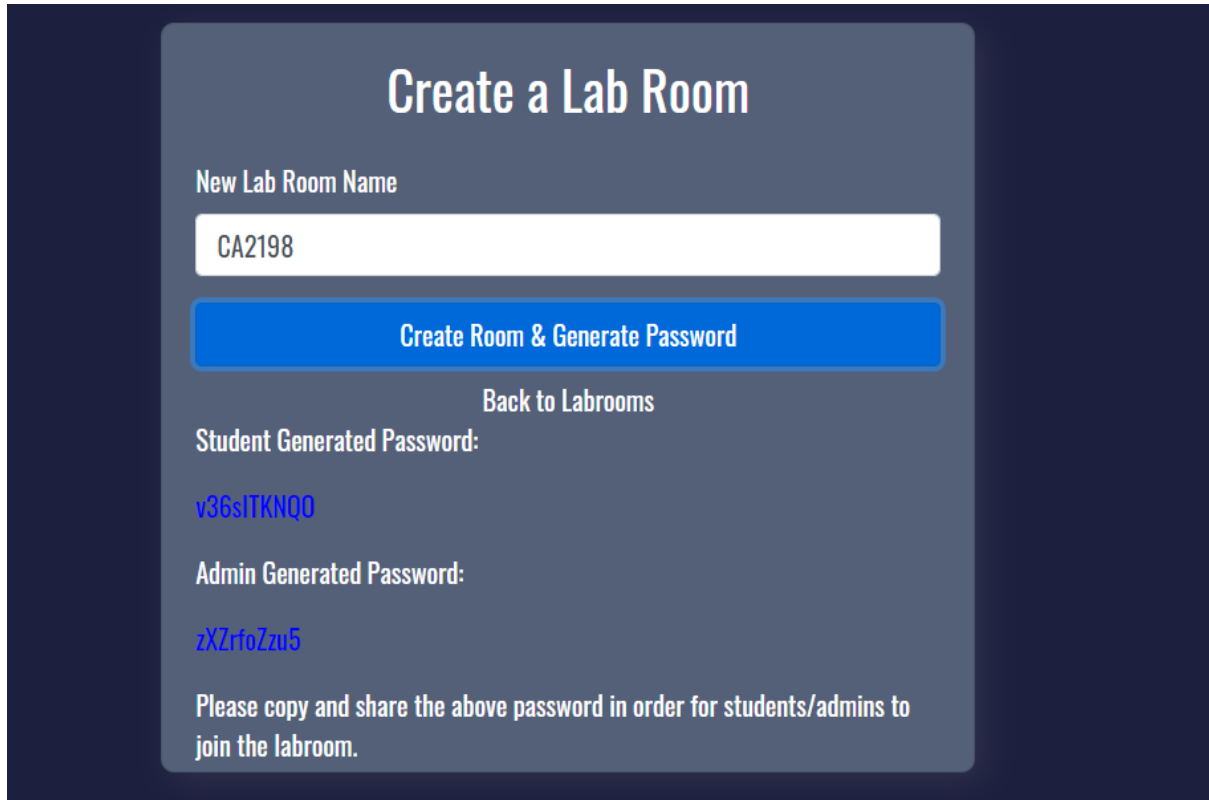
Once a user's account is created and authenticated, we store their information in a user collection in the firestore database where a student object would maintain the information in the below object example.

User
Email: user@mail.com
Admin: true/false
inCall: false/true
Labs: -labID-1 -labID-2
lastLoginTime: dd-mm-yyy 00:00 UTC +1
roomID: "ewYV87XJUo94ppte8N"

The user's roomID is their personal lab session when they begin editing code during the lab. The inCall status tracks whether a user is currently in a call. Labs is a list of lab ID's for the labs/modules that a user is currently part of on the platform. The Admin status is whether a

student has admin permissions on the platform or not. Admins are permitted to create labs, they can then enter that lab as a tutor and invite other students and tutors to take part in that lab.

Creating a lab room

The image shows a web interface for creating a lab room. It has a dark blue background. In the center is a light blue rounded rectangle containing the title 'Create a Lab Room' in white. Below the title is a label 'New Lab Room Name' followed by a white text input field containing 'CA2198'. Underneath the input field is a blue button with white text that says 'Create Room & Generate Password'. Below the button is a link 'Back to Labrooms' in white. Then, there are two sections: 'Student Generated Password:' followed by the password 'v36sITKNQ0' in blue, and 'Admin Generated Password:' followed by the password 'zXZrfoZzu5' in blue. At the bottom, there is a white instruction: 'Please copy and share the above password in order for students/admins to join the labroom.'

A user who is a tutor has admin permissions and will have the option to create a new lab room. This is where a user inputs the name they want to call their new lab.

```

async function checkLabRoomExists(studentpassword, adminpassword){
  const labroomName = document.getElementById("labroom-name").value; //get
  const labcollection = db.collection('labs'); // gets the labs collection
  //adds to labs collection

  //check if collection contains new lab name
  const labCheck = await labcollection.where('Lab_Name', '==', labroomName)
  const currentUser = auth.currentUser;
  const email = currentUser.email;
  const uid = uuid();

  if (labCheck.empty) { //if nothing exists in the labroom collection with
    console.log("collection doesn't contain lab: " + labroomName)
    db.collection('labs').add({ //adds the newly created lab to the lab
      Lab_Name: document.getElementById("labroom-name").value,
      Lab_Key_Student: studentpassword,
      Lab_Key_Admin: adminpassword,
      RoomID: uid,
      Host_email: email})
    addHosttoLab(); //automatically add the lab id and information to the
  }
}

```

Once the user clicks the 'Create Room & generate' button, a function checkLabRoomExists() is called. Within this function, A call is made to the Firestore database to get all the relevant labs collection information. A search query is then implemented to find any documents within the lab collection that may already contain the lab name that the user has inputted. If the search query comes back empty, that means that a lab with that name does not already exist in the collection therefore, the function adds a new document to the database with the following fields:

Lab_Name, Lab_key_Student, Lab_Key_Admin, RoomID and Host_email

2 keys are generated as you can see above ^, these keys are displayed on the users UI. The student key is given to students while the Admin key is given to those in which the user wants to appoint as tutors. These keys are used to allow a student/tutor to access a lab room with specific privileges.

Joining A Lab Room with A Key

Once a student/tutor has gotten a key from a lecturer they can join a lab room.

Join A Lab

Please enter Lab room code

XjvNhJd2ly

Join Room

Successfully joined : CA34090

```

async function adminCheckKey(){ //checks if user key is associated with lab_Admin if not it calls studentCheckKey() t
  const labcollection = db.collection('labs');
  const labcodeinput = document.getElementById("labcode-input").value; //users inputted code
  console.log(labcodeinput);

  const adminKeyMatch = await labcollection.where('Lab_Key_Admin', '==', labcodeinput).get(); //checks if there is an
  if (adminKeyMatch.empty) { //if nothing shows up stating that there is a match
    console.log('Password incorrect - admin');
    studentCheckKey(labcollection, labcodeinput); //check if the key matches the student key
    return;
  }
  console.log('password correct'); //else password is correct
  adminKeyMatch.forEach(doc =>{
    console.log(doc.id, '=>', doc.data());
    lab = doc.data().Lab_Name;
    const labDoc = labcollection.doc(doc.id);
    labDoc.update({
      Lab_Admins : firebase.firestore.FieldValue.arrayUnion(auth.currentUser.email) //add current users email to
    })
    admin = true;
    addLabToUsersDocument(doc.id) // if key is valid add lab to users firestore lab list, display lab on their
    document.getElementById("printStatus").innerHTML="You have been granted access to: " + lab + " as a tutor.
    handleVisibility();
  });
}

```

Within Join.js, when a user inputs a key, a series of checks are run. These checks basically verify if the user is going to be a tutor within a lab or a student participant.

The check above ***“adminCheckKey()”*** takes the user's inputted key value and checks if it exists within any of the lab collections documents under the field 'Lab_Admin_Key'. If there is a match, the user is added to an admin list which is stored within an array in the documents fields within the firestore database. ***“addUserToLabList()”*** is also called which adds the lab ID to the users lab list which is found in the users document within firestore. However, if that search comes back empty, another check is run called ***“studentCheckKey()”***, which checks if the inputted key exists within any of the lab collection documents under the field 'Lab_Student_Key'. If a key exists, the user is put into a student list which is stored on the firestore database. ***“addUserToLabList()”*** is also run

Document within lab collection:

▼ Lab_Admns

0 "munashe@mail.com"

▼ Lab_Students

0 "123elonmusk@tesla.com"

1 "test3@mail.com"

2 "regtest3@mail.com"

3 "idtestaccount@mail.com"

Document within user collection - lab ID is stored within the labs field which is of type array

+ Add field

Admin: true (boole

email: "regtest3@mail.com"

inCall: false

▼ labs

0 "th9wVAII6NO3l6XyTGIs"

1 "dMLbEgceViJszRyiO0i9"

2 "oVXfGAHIPccd2CHyZr8s"

3 "0SVxTj9NK1KLbzQwV2Nr"

Displaying The Users Lab Rooms

```

export const getLabsToDisplay= async() => {
  const userId = auth.currentUser.uid;
  const fetchPromises = [];

  if (userId){
    // console.log("getting labroom List for user: " + userId);
    const userDoc = await db.doc(`users/${userId}`).get(); //gets the users entire doc
    const userData = userDoc.data();

    const labs = userData.labs; //gets the users lab list from their firestore collection
    if (labs){ //if labs exist
      labs.forEach((labId:String) => { //for each lab in the users lablist
        const nextProm = db.doc(`labs/${labId}`).get(); //get the lab fields for the labId
        fetchPromises.push(nextProm); // and store the lab info into fetchpromises
      });

      const labsToDisplay = await Promise.all(fetchPromises);
      const labArray = labsToDisplay.map((lab) => {return lab.data().Lab_Name}); //
      console.log(" in labstodisplay " + labArray)
      return labArray;
      // this.setState({ labArray});
    }
  }
}

```

On the dashboard, the user has an option to go to 'lab rooms.' Here, the user's lab rooms which are stored within the users document on firestore are displayed. When a user goes to the lab room page, the function **getLabsToDisplay()** is called. This function queries the database to find the users labs list. Once found, the function goes through each of the labs within the array and adds them to a list. The items at this point are labids in which we want to get the specific lab names from the id.

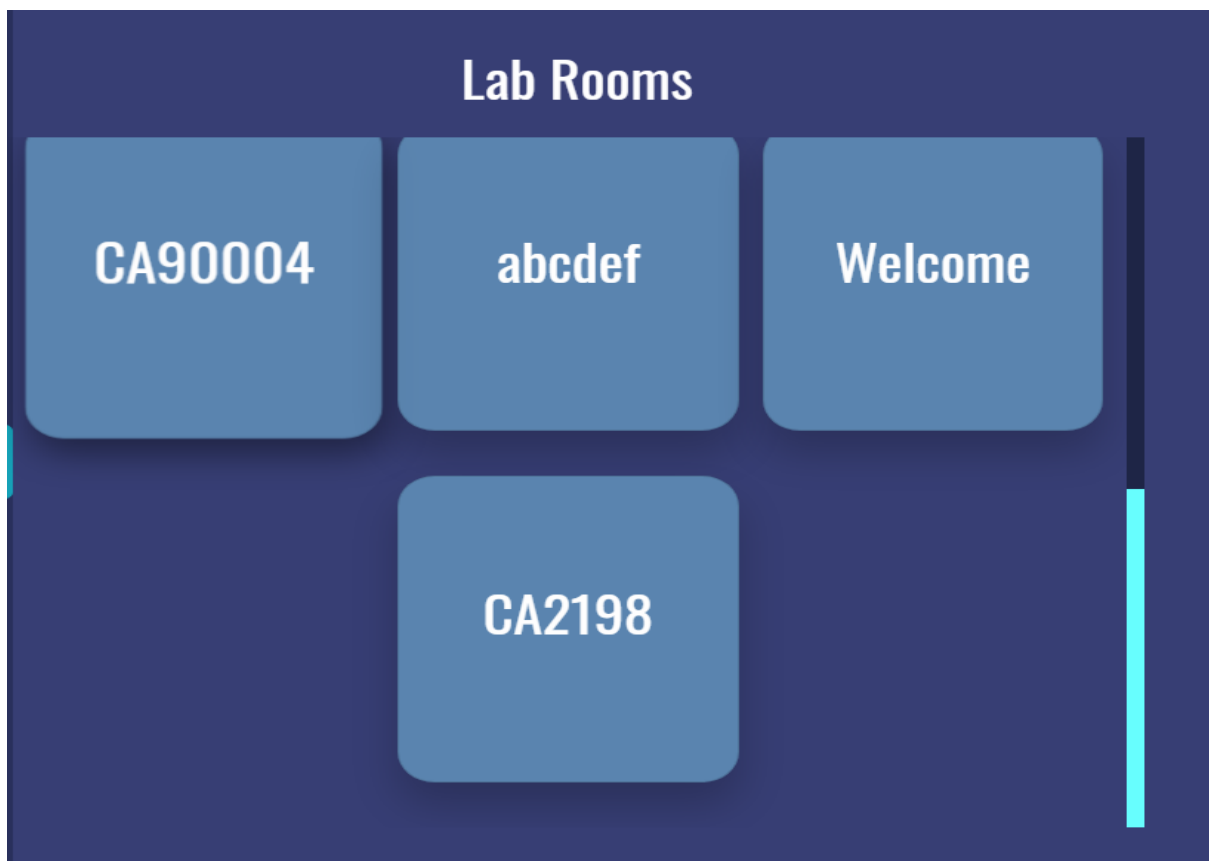
In this case,

`const labArray = labsToDisplay.map((lab) => { return lab.data().Lab_Name});`

This goes through all the lab IDs within the array and gets their Lab_Name. The lab IDs in the list are replaced with the lab's name.

```
render(){
  const array = this.state.labArray;
  return (
    <div id="row" className="lab-row">
      {
        // below Array.from() is used with an arrow function to manipulate the items in labArray - so for each item in the array, display
        Array.from(array, child =>{
          const a = child;
          return(
            //each lab card is associated with a specific key, this key is generated by shortid
            <Link to="/create-room" onClick={() => getLab(a)} className="lab-links" style={{textDecoration: "none"}}>
              <Card id="lab-card-style-2" key={shortid.generate()}>
                <Card.Body>
                  {/* display the lab card and its name(.Lab_Name taken from the lab collection in firestore) on the dashboard */}
                  <h2 className="dash-cards-h2">{child}</h2>
                </Card.Body>
              </Card>
            </Link>
          )
        })
      }
    </div>
  )
}
```

Once that function has run successfully, a render is called in which it displays the user's lab list as a row of cards:



Joining The Lobby

```

//Function that checks if the current user is a tutor/ student
// if it's a tutor the Help requests button is displayed.
// else the Start Code button is displayed
async function checkTutors(){
  var labid = await getLabData(lab);
  const labDoc = await db.doc(`labs/${labid}`).get();
  const labData = labDoc.data();
  const tutors = labData.Lab_Admns;
  if (tutors.includes(auth.currentUser.email)){
    document.getElementById('requestButton').style.display = 'block';
  }
  else{
    document.getElementById('startCoding').style.display = 'block';
  }
}

```

When a user joins an active lab room, depending on what list the user's email is linked to within the labs document on firestore, they would either be displayed as a Student participant or a Tutor. The UI for both is slightly different due to their different permissions. The tutor's UI of the lobby will have a button called 'requests', while a student won't have this option, instead they will have a button displayed as 'Start Coding'.

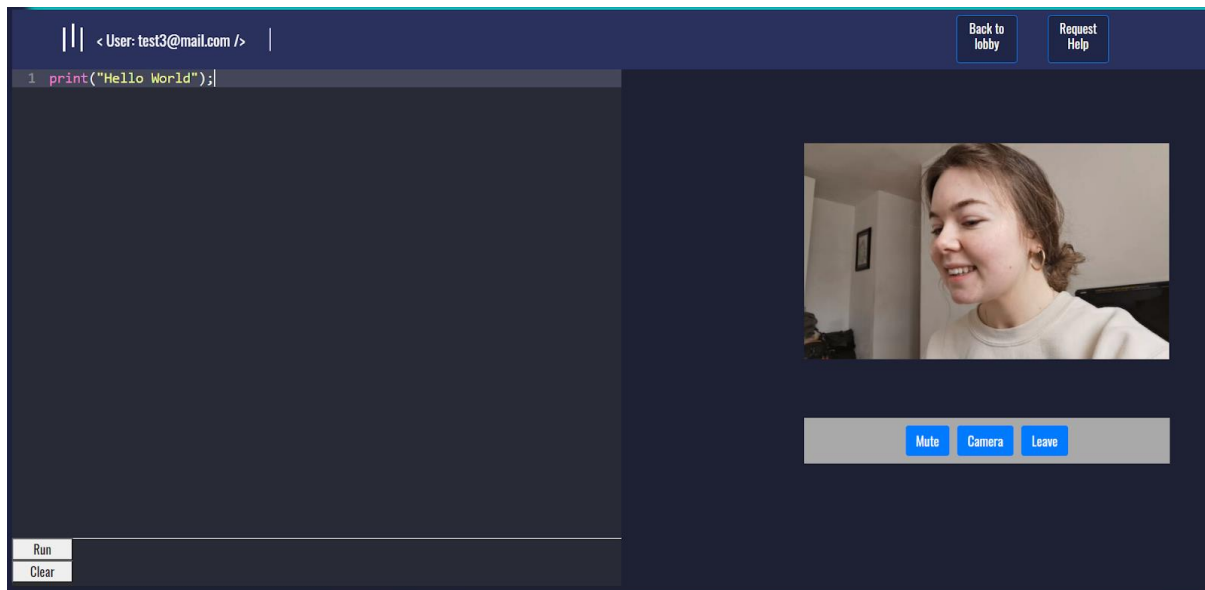
```

const SetParticipants = ()=>{
  useEffect(() => {
    const interval = setInterval(async() => {
      UpdateParticipants(labName);
      var labID = await getLabData(labName);
      await getParticipants(labID);
    },2000); //runs every 2 seconds
    return () => clearInterval(interval);
  },[]);
};
SetParticipants();

```

setParticipants() is a function that is run on an interval every 2 seconds. This is done to ensure that the participant list being displayed on the UI is constantly being updated.

Entering a Code Session



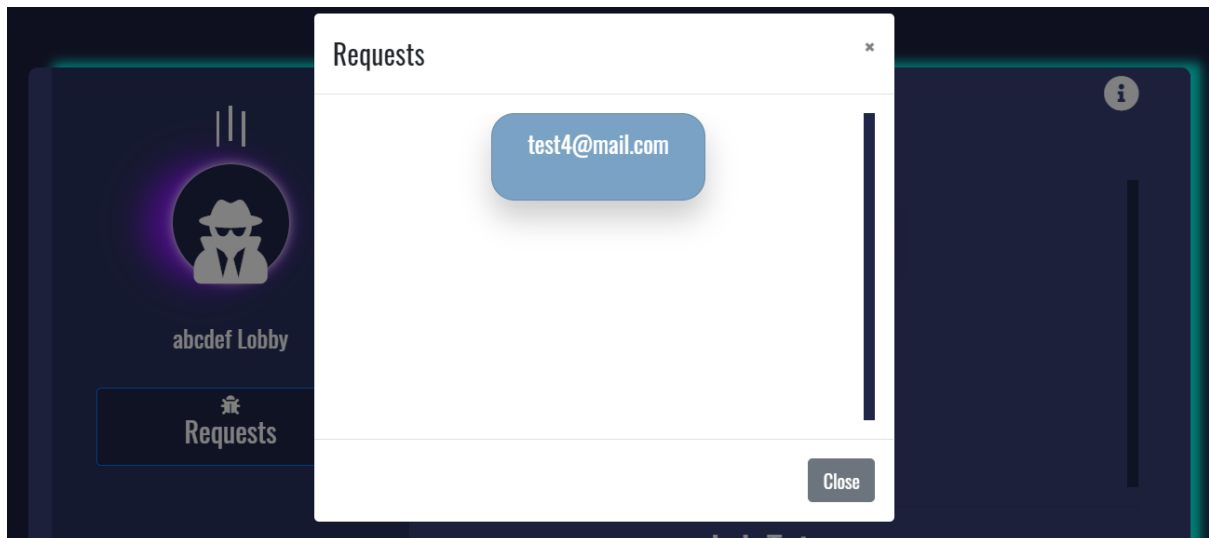
Within the coding session, a student user is presented with a text editor that runs python along with the ability to have their camera on/off. There are buttons on the top of their session called 'back to lobby' and 'Request Help'

Requesting Help

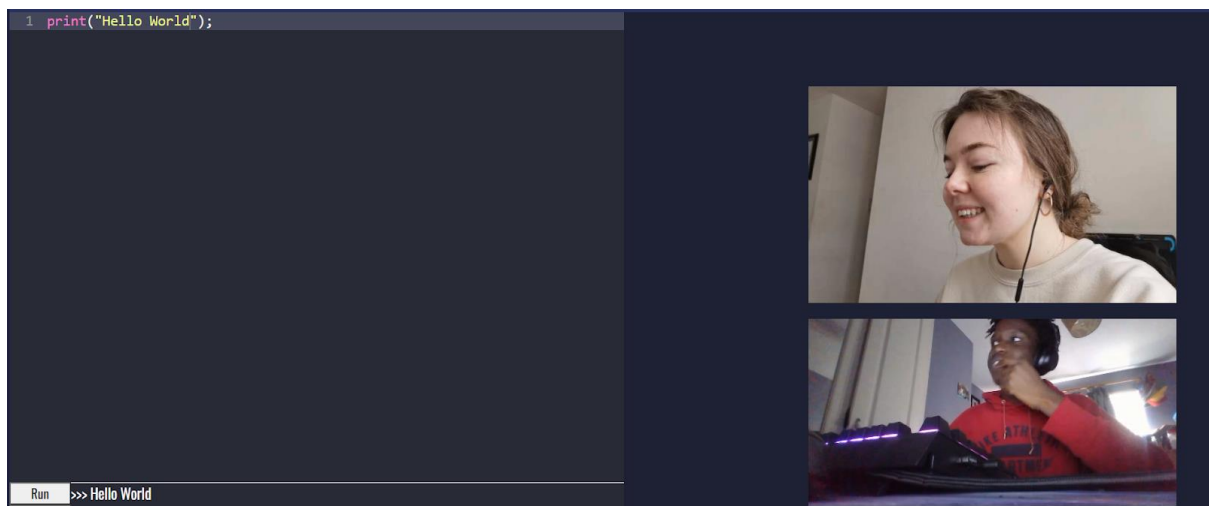
```
//used when someone presses request Help - Function calls UpdateRequests
//which updates a specific labs requests list on firestore with the current users email.
async function requestHelp(){
  try{
    alert("Requesting help");
    UpdateRequests(lab);
  } catch{
  }
}
```

```
//Function updates request list on firestore in the labs collection
export const UpdateRequests = async(lab) =>{
  const labcollection = db.collection('labs'); // gets the labs collection from firestore
  const labCheck = await labcollection.where('Lab_Name', '==', lab).get(); //queries the lab collection to find a lab that h
  if (labCheck){ //if there is a match
    labCheck.forEach(doc =>{
      const labDoc = db.collection('labs').doc(doc.id); //gets the labs document infomartion
      labDoc.update({ //updates the labs fields in order to add the current users email to the request list.
        Requests : firebase.firestore.FieldValue.arrayUnion(auth.currentUser.email)
      });
    });
  }
}
```

Once a student clicks the 'Request Help' button, The function '**updateRequests()**' is called in which the current user's email is added to a request list stored on firestore. This function runs on a 2 second interval in order for the list to be constantly up to date. The users email who requested help, is then displayed within the tutors request list below:



Tutor Joins The call



When a tutor joins a call with the student, the student's name is popped off from the request list. Both the student and the tutor can interact with the text editor as it's peer-to-peer.

Server

We used Express and Socket.io for the server implementation. Socket.io is used to manage all client server communications. This was ideal as it meant we could keep all the server code in one place and the functionality as necessary. In addition, it allowed the use of PyShell to run python scripts on the server and have the results sent back to the clients to display on the front-end console. In the below code snippet below shows how the server handles a new user connecting to a room. It first checks if that user's ID is already connected to a room. If it is, that older connection is closed as users can only be in one room at a time. After this the user can connect to the new room. In addition, this prevents a refresh error bug (more details in problems solved section).

```
// Event to check if a person connects to socket server. "Socket" is the socket object for one person
io.on('connection', socket => {

  //Attaching event listener to socket event called "Join room". Used on the client side
  socket.on("join-room", infoFromClient => {
    console.log(infoFromClient)
    let roomID = infoFromClient.roomID
    let clientID = infoFromClient.clientID
    console.log("join succesful")
    // console.log("CLIENT ID: ", userID)

    //If theres a refresh from the same client, disconnect that old socket
    if(clientID in clientToSocket){
      const oldSocketID = clientToSocket[clientID]

      console.log("old socekt id: ", oldSocketID)

      let room = users[roomID];

      if (room) {
        room = room.filter(id => id !== oldSocketID);
        users[roomID] = room;
      }
      socket.broadcast.emit("user-leaves", oldSocketID)
      delete socketToRoom[oldSocketID]
    }
  })
})
```

The server also handles the peer-to-peer connection of the code editor. Echo Labs is designed to replicate the pair programming utilised in the lab session in DCU, thus within a room the connected users can both interact with the same code. It is pair programming so it's designed so one user can program at a time just like in labs when students work from a single machine. The goal is to promote communication and collaboration between students and tutors. The snippet below shows how the server handles receiving code. When the code is received, there's a check to ensure that it's not the same as previously sent code to prevent redundancy. When the code passes that check, the code is then sent on to the other sockets in the room.

```
socket.on("update-code", code => {

  const roomID = socketToRoom[socket.id]

  // SEND CODE TO EVERYONE EXCEPT MYSELF
  if(code !== codeStore){

    codeStore = code

    console.log(codeStore)
    //checher peek to make sure that the code is not the same before sending it to the others
    socket.broadcast.emit("receive-update-code", codeStore)
  }
})
```

The server's final main functionality is to run the python scripts being written in the aforementioned code editor. When a user clicks the "run" button on the client side, code is sent as a string to the server. The code is then run through our python shell and the result is returned to the client as detailed in the snippet here.


```
//Send code to the server and have it run in the python shell
socket.on("run-code", code => {
  console.log("CodeToRun: ", code)
  try{
    PythonShell.runString(code, null, function (err, result) {
      if (err) {socket.emit("receive-result", err)}
      console.log('execution finished', result);

      socket.emit("receive-result", result)
    })
  }

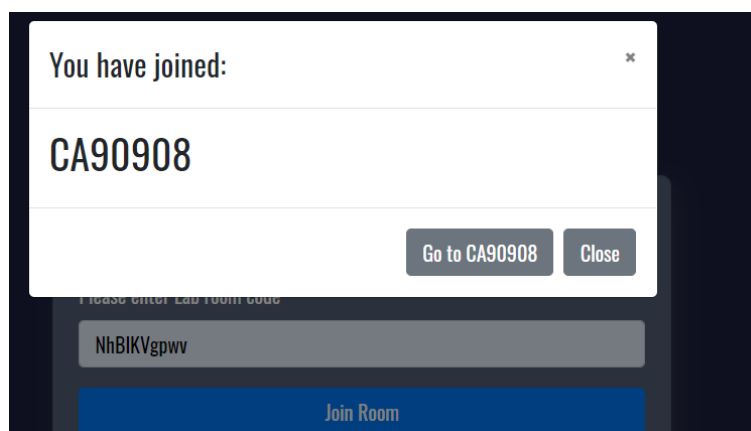
  catch(e){
    console.log(e)
  }
})
```

Problems Solved

Initially, we wanted to implement our app to be hosted using Firebase. We believed this could be done with any application no matter how big or how small. However, once we implemented a server to Echo Labs and attempted the hosting implementation process, we figured out that Firebase did not accept custom servers. It would host our front end and firestore implementations but would not host any of the server's functionality. In this case, we had to opt to a different hosting platform that accepts a custom server. We chose heroku.

User Testing allowed us to identify bugs in the web app that needed to be fixed. For instance, some users had reported that when trying to click on card links they had to specifically click the text inside the card which took them a while to figure out. This was fixed by making the whole card a link.

Users also reported that once they joined a lab room they didn't understand where to go after. In this case we implemented a pop up that allowed a user to go straight to the lab room they just joined:



Our server implementation itself was not without problems. Our implementation of WebRTC through the Express server and socket.io proved to be tedious at times. During user testing

we discovered that users without cameras would not send any media to the other clients. While we expected that they would of course not send their visual media we did not anticipate that they would also not send audio too. In addition, WebRTC handshake defaults meant that if a client never sent media to its peers, it would also not receive the media from its peers resulting in what was essentially a blank call for both parties involved.

To solve this problem, we included a check for a user's available devices in our room script. This check lets the room know that the user does not have a camera, but they do have a mic, otherwise, the absence of a camera results in a false value for both audio and visual media. Now camera-less users can communicate with their peers and send blank visual data. The following images include the code for the device check as well as an image of the scenario where one of the peers only has a microphone and no camera.

Another issue we encountered involved page refreshes. The user's media and connection information are collected within a useEffect function on the room script. This useEffect function is designed to run once when a user goes onto the page, otherwise every time the dom is refreshed. Unfortunately, this meant that refreshing the page, resulted in the same user creating a new socket connection and sending new media information before reentering the call. The result of this was the same user appearing two or more times depending on the amount of page refreshes, in the same call. To solve this problem, we sent the user's unique ID provided by firebase to the server and stored it in an object as a key with the socket ID as the value for that key. Now, when a user enters the room the server checks if that user's ID is in the object, if it is, the old socket ID is destroyed, disconnecting it from the call/room and the connected clients are instructed to destroy the old sockets media as well. This prevents duplication on refreshes. Additionally, this same function prevents users from attending more than one lab session at the same time.

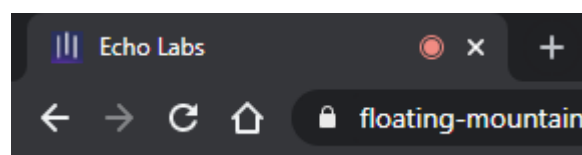
```
//If theres a refresh from the same client, disconnect that old socket
if(clientID in clientToSocket){
  const oldSocketID = clientToSocket[clientID]

  console.log("old socekt id: ", oldSocketID)

  let room = users[roomID];

  if (room) {
    room = room.filter(id => id !== oldSocketID);
    users[roomID] = room;
  }
  socket.broadcast.emit("user-leaves", oldSocketID)
  delete socketToRoom[oldSocketID]
```

During testing we also encountered the issue of the calls not being fully disconnected after a user returned to the lobby, or if they switched to a different url on the web app. We realised that on url changes, the browser was not being instructed to release the user's camera and microphone as well as this, the server was not being instructed to disconnect the client from the lab session. This was indicated by the recording symbol still showing after page changes:



To solve this issue, a listener for url changes was included in its own useEffect. This useEffect only runs when the history hook is used, which is essentially on any url change. Now, when a user clicks the back button on chrome the call is disconnected by virtue of this function as media.

```
//Disconnects when user changes the URL
useEffect(() => {
  return history.listen((location) => {
    userTracks.current.getTracks().forEach(track => track.stop());
    socketRef.current.disconnect()
  })
},[history])
```

Similar functions were implemented in the return to lobby and the leave button in the room.

Results

For user testing we included a sample space of users who were both current or ex-students of computer science as well as users who didn't know anything about computer science or programming but would have been interested in learning to code. We had some testers participate as students and some as tutors. 76% of testers rated the UI a 4 out of 5 indicating that it was easy to use and understandable. 75% of tutors agreed that it was not difficult to respond to help requests sent by students as well as this 100% of students found it easy to request help. More details on this are contained in the testing documentation.

Overall testers seemed satisfied with the experience of using the application with 84% agreeing that it would be beneficial for their learning experience.

At the conception of this project, we knew that it was an ambitious undertaking especially since these were frameworks and technologies that we were not overly familiar with. However, based on our proposal we achieved what we set out to do, which was to provide a solution for programming lab sessions at home. What resulted was a web application that allows students to join lab rooms, begin programming in those rooms and when needed, request help from tutors as if they were physically present in a lab in college.

Future Work

Expanding Echo Labs

For future developments we would like to expand echo labs in order for it to be an app that is available on mobile devices such as android and IOS.

As well as this, adding additional features such as screen sharing, text chat messaging and student pair programming as a group would be highly beneficial.

User testing allowed us to get an idea on what features users may have also liked to be added, however due to time constraints we had to put these on hold, these included: A setting in order to display the UI in a light theme mode as well as a dark theme mode and the ability to have a user profile picture and be able to change it.