

Bingo Bonanza

Technical Specification

Name(s):	Brendan Simms , Shane Lennon
Student Number(s):	19500949 , 17496766
Module Code:	CA400
Supervisor:	Mark Humphries
Date Of Completion:	06/05/2023

Table Of Contents

1. Introduction	3
1.1 Overview	3
1.2 Glossary	4
2. General Description	5-11
2.1 Business Context and Motivation	5
2.2 Research	6-10
2.3 Target User Characteristics & Design Objectives	10-11
3. Design	12-15
3.1 System Architecture Diagram	12
3.2 Data Flow Diagram (Outside Of Game)	13
3.3 Data Flow Diagram (Inside Of Game)	14
3.4 Context Diagram	15
4. Implementation	16-42
4.1 Authentication	16-25
4.2 Configuring a Game	25-30
4.3 Hosting a Game	30-32
4.4 Playing a Game	32-36
4.5 Book Generation	37-38
4.6 Pseudo Random Number Generation	39
4.7 Websocket Server	40-42
5. Problems We Encountered & How We Solved Them	43-50
5.1 Websocket Server	43
5.2 Book Generation	43-44
5.3 Issues with React State	44-46
5.4 Sending Arrays over Sockets	47-50
6. Results & Conclusion	51-53
6.1 Directions For Future Work	51-53
6.2 Conclusion	53
7. References	54

1. Introduction

Abstract

‘Bingo Bonanza’ is an all in one online platform for both people who enjoy playing bingo and business, charities or other organizations that wish to host a night of bingo. This application allows all users to play/create real bingo games for real cash prizes. This is done using a wide aspect of computer technologies, architecture and frameworks that will be discussed in depth throughout the entire document. Our motivations for choosing to create an online bingo application as our final year project can be found below alongside a detailed description of our design, implementation, problems faced and an in depth explanation of various aspects of our application.

1.1 Overview

As aforementioned, ‘Bingo Bonanza’ is an application that has two distinct purposes for two distinct types of users. We have coined them ‘Host’ for those who wish to host their own games of bingo and ‘Player’ for those users who wish to play games of bingo and potentially win cash prizes. ‘Bingo Bonanza’ is a web application that has been designed for all devices.

All users can create an account on our Web-Application and accounts are not unique to the user type but rather the user selects a user type when they complete logging in. This user type then determines the sections of the web application accessible to the user for the session.

We aim to provide users with an interactive game of bingo that allows many players to play a single hosted game. We achieve this through using Websockets, in the form of Socket.io, to provide full duplex communication between a Host and all Players in that given hosts game. We use AWS Lambda to provide the PRNG Generation and generation of our players game tickets.

1.2 Glossary

- **Host** - Users who wish to set up and host their own games of Bingo
- **Player** - Users who wish to play a game of Bingo
- **Bingo Books** - A set of six Tickets where the entire contents of a Book is ninety numbers one instance of each number in the range of one to ninety
- **Bingo Ticket** - A set of fifteen numbers ranging from one to ninety
- **Bingo Line** - The smallest component of a Bingo Book, a line is a set of five numbers ranging from one to ninety
- **Asynchronous** - Asynchronous means the process run independent of all other processes and can be called at any time
- **JSON** - JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects
- **React** - React is a free open source Javascript library. It is generally used to build User Interfaces
- **Check** - A player in a bingo game wishes to see if they have won a prize
- **Valid Check** - When a player has checked and it is a valid win of the current prize.
- **AWS**- Amazon-Web-Services
- **PRNG** - Pseudo Random Number Generation
- **CSPRNG** - Cryptographically Secure Pseudo Random Number Generation
- **LCG** - Linear Congruential Generation is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation.
- **XORshift*** - PRNG algorithm that applies an invertible multiplication (modulo the word size) as a non-linear transformation to the output of an xorshift generator
- **XORshift** - PRNG algorithm that works by generating the next number in their sequence by repeatedly taking the exclusive or of a number with a bit-shifted version of itself
- **Blum Blum Shub** - PRNG algorithm that takes the form $x(n+1)=x(n^2)\bmod M$ where M is the product of two large prime numbers.
- **Websockets** - WebSocket is a communications protocol for a persistent, bi-directional, full duplex TCP connection from a user's web browser to a server
- **Socket.io** - A websocket framework

2. General Description

2.1 Business Context and Motivation

There were two major reasons we decided to develop the application we did and they are as follows:

1. Due to the fallout from the Covid-19 epidemic a lot of businesses have been struggling since then to maintain a regular stream of customers like they possibly did before the epidemic. The industry that suffered this effect the most is potentially the entertainment industry, this is the industry that Bingo falls under. Brendan, one of the members of this project, worked in a Bingo both before Covid-19 lockdowns were put in place and after they were lifted and he noticed that the numbers of customers dramatically dipped after the lockdown lifted compared to the numbers of customers pre-covid. It was a general consensus that customers either got used to playing online Bingo games or just didn't feel safe enough to return to a venue where a large amount of people would be present. This is especially pertinent as the majority of the customer base would be considered at high risk of serious health issues if they got Covid due to their advanced age in most instances. Due to this decline in the player base it eventually led to the closing of the Bingo Hall business. We wanted to introduce a way to play remotely so that people who cannot physically attend the event can still be involved.

2. Bingo halls pay a large amount of money to companies to provide the infrastructure to allow them to be able to run their business, the main pieces of these infrastructure that cost a major amount of money are the Bingo Software, the Servers and the Tablets that the customer use to play the Bingo games. We felt that this expensive architecture being installed was a huge drain on Bingo companies resources when there is a potential to do this using Distributed Systems and a Web Application that allows customers to play Bingo on their own phones and for Bingo halls to host games of Bingo with nothing more than a laptop or a mobile device.

2.2 Research

Pseudo-Random Number Generation

Our project heavily relied on being able to generate pseudo-random numbers for the game of bingo to be played so we decided to start off our project by creating our PRNG algorithm. Due to this being a critical section of our application and a lack of knowledge or experience with PRNG, we carried out a huge amount of research on creation of a PRNG algorithm. We needed to be able to assure within reason that the output of our PRNG algorithm was both unpredictable (by a human or advanced computer algorithm) and had some level of arbitrary randomness.

Randomness in a PRNG is given to it by the seed and due to this we determined a way to generate a new random seed each given time without ever having the user access this information. We determined that the best way to do this was by using system time (hours, minutes and seconds) converted to seconds when the algorithm is called, we then decided we would use the Secrets Python library, which provides CSPRNG methods, to select a random number below this system time in seconds.

We then decided that a single PRNG algorithm may not be enough, so we put our researching efforts into the various different implementations of PRNG that exist in the world today and decided that we would pick three entirely different implementations and use each of them in our implementation where a random one was used each time. Originally to do this we proposed using current system time of seconds (0-59) and have a series of if/else if statements, where we took the current seconds and modulo it by the prime numbers below 60 and if it resulted in 0 the condition was passed, we quickly realized that this was not fair as the first if condition would be hit more than others, so we decided instead to implement the secrets library from Python again and select one of the 3 implemented PRNG algorithms from a List.

After this was all determined the only thing left to determine was the actual algorithm used to generate the random number. We looked into a vast amount of existing PRNG algorithms, their benefits and disadvantages and attempted to pick

vastly different implementations. After our research was completed we decided to use LCG, XORshift* and Blum Blum Shub as our three implementations.

Front-end framework

When starting our project we had to decide on what framework we would use for the front-end of our application. When deciding this we had to consider whether we would be making a mobile application or it would be a web browser application that mobile users can access. We decided to use react due to its flexibility and many libraries, but in react we initially decided with react native for the reason that it allows you to develop for both mobile and web simultaneously. This would allow us to provide bingo-bonanza to both mobile and web users, however we did encounter some problems neither of us owned iOS products so developing for them proved difficult. Another issue we encountered was our lack of experience with react native we encountered too many problems early on that we felt may lead to more in the future, we then decided that we could have a web application that mobile users can use so we settled on basic react.js the reason for this was due to familiarity from previous projects.

Login Authentication

For our project we needed a login authentication to allow users to sign up for our application and save certain information about them such as their names, address, balance and user type. We had a look at a few different ways of implementing Authentication such as Firebase Auth, and AWS-Amplify, we settled on AWS-Amplify's Auth as again there was some previous experience on previous projects but another reason for the choice of Amplify was due to the fact that we were already contemplating on using AWS Lambda functions as our backend and this way we could keep a lot of our project on the same Cloud Computing Platform. Amplify Auth also provides email verification for login and password updates so we can provide more security on fake accounts being created. Amplify Auth also provides tokens that we can use in our application on registration and login.

For storing the users data we talked about two ways the first was using a database and encrypt the users passwords to create secure user-data. The other method was

using AWS Cognito User Pools. Using AWS-Amplify allowed us easy integration of Cognito User Pools so this ease of use was a factor that went into consideration when choosing a storing method for user data. One of the main selling points of the Cognito User Pools is that they provide a secure way of storing users information. Firstly to access the cognito pool you would need the correct permissions and it also stores users passwords so even we cant see them or access them. With storing users data in a database we could run into an issue of bottleneck using Cognito User Pools will prevent this issue. We can also create custom user attributes such as balance and user-type that we can use and update throughout our application.

Cloud Computing Provider

When choosing a cloud computing provider we had to consider the type of application we were making and what services we would need. As we were making a web application it will need hosting services as well as data storing and backend hosting. The Platforms that came to mind were Microsoft Azure, AWS and Google Cloud Computing. Neither of us used Google Cloud Computing before so we decided that it would most likely not be used. Both of us had previous experience with Azure and AWS on work experience but we decided we had more experience with AWS as it had been used on previous projects. AWS also provided more services at our disposal and mentioned in previous sections the services provided were perfect for our application so it made it an easy decision for us.

User Interface & User Experience

As our application is designed for both mobile and desktop we had to come up with a design that suited both, looking at a bunch of different websites that can be used on both the web and mobile we found out that a more simplistic design is better, this is because if the application had more design features on the website that it would be quite difficult to translate these to a mobile device with varying screen sizes and small screen real estate. For this reason we decided to use simple design to keep it looking modern but also to make it look attractive to our users. We used a simple colour scheme of light blue whites and greys to keep the application modern and minimalistic.

The gameboard itself has more colour in it as the bingo balls provide the colour needed to make you feel like you're playing a game, this is why most of our boxes are rounded such to make it feel more “game like”.

Data Structures & Time Complexity

Our game relied on users being able to have a specific number of books (3, 6, 9, 12) for each game they play. These books contain 90 numbers in each one which exist in sets of 5 that each correspond to a specific bingo ticket and line. This introduced an issue of how we were going to store this data and how this data could potentially have time complexity implications. The time complexity issues were of the utmost importance as these lists would need to be iterated through constantly within our game for reasons such as determining a players best ticket and whether or not they had a valid check.

We originally proposed two potential solutions to this one was using a database that would store this information and the other was using already established data structures. We decided upon further research that the database would not be a suitable option due to the sheer amount of data it would have to contain and the relative query times required to retrieve the information would not be usable for this when the number of players in a game was large. This left us with using built in data structures and storing them within the web application itself. The question then became “what data structure would we use?”.

We decided that simplicity was our friend as the bingo books were generated in Python but our webapplication was in Javascript so using a more advanced data structure than Lists/Arrays would not be possible due to the potential implications of the Data structure not being supported on both Javascript and Python. We knew Arrays in Javascript had a lookup time complexity of $O(n)$ but a get time complexity of $O(1)$ meaning it was perfect for its purpose and it did not cause a negative effect on the speed at which the game of bingo could be played.

Websockets

Due to the nature of our project we required full duplex communication between the Host and Player during the bingo game to be able to constantly update both the

Player and the Host with events as they happened. Websockets are the standard to do this full duplex communication but there is a vast amount of websocket frameworks that can be used. Before we conducted our research on the frameworks we decided to try determine what we were looking, in terms of the functionality, from the websocket framework, we determined that the only real aspect we needed from the framework was the ability to create rooms to allow messages/events to only be broadcasted to the players within a certain game.

We found that a lot of frameworks offered this so the decision then just became based on the standards within the industry and the familiarity between ourselves with the framework. We decided to use Socket.io.

2.3 Target User Characteristics & Design Objectives

Our target user was very important when discussing our project as they heavily influenced our UI design approach. The main demographic that play bingo are elderly men and women. While this is the main demographic of our target users we didn't want to discourage other potential users from using the web application so although our UI design was implemented with our target user in mind we wanted to make sure it was inclusive.

Our target users, in general, have issues with technology. This made the goal of our design objectives very apparent. We wanted to ensure that we used a minimalistic and simple design so that the web application was easy to navigate and easy to use. We achieved this by clearly laying out navigation menus and making joining into a game a simple series of pressing buttons and entering in a username string to find the Host's current game. We also introduced various UI prompts to ensure that if an incorrect entry happened the user was prompted that it occurred and informed with an easy to understand message. An example of this is in our Authentication pages where if a user types in a username that does not exist the error message will be this:

UserNotFoundException: User does not exist.

OK

Sign In

Email address
someemail@gmail.com

Password
••••••••

SIGN IN

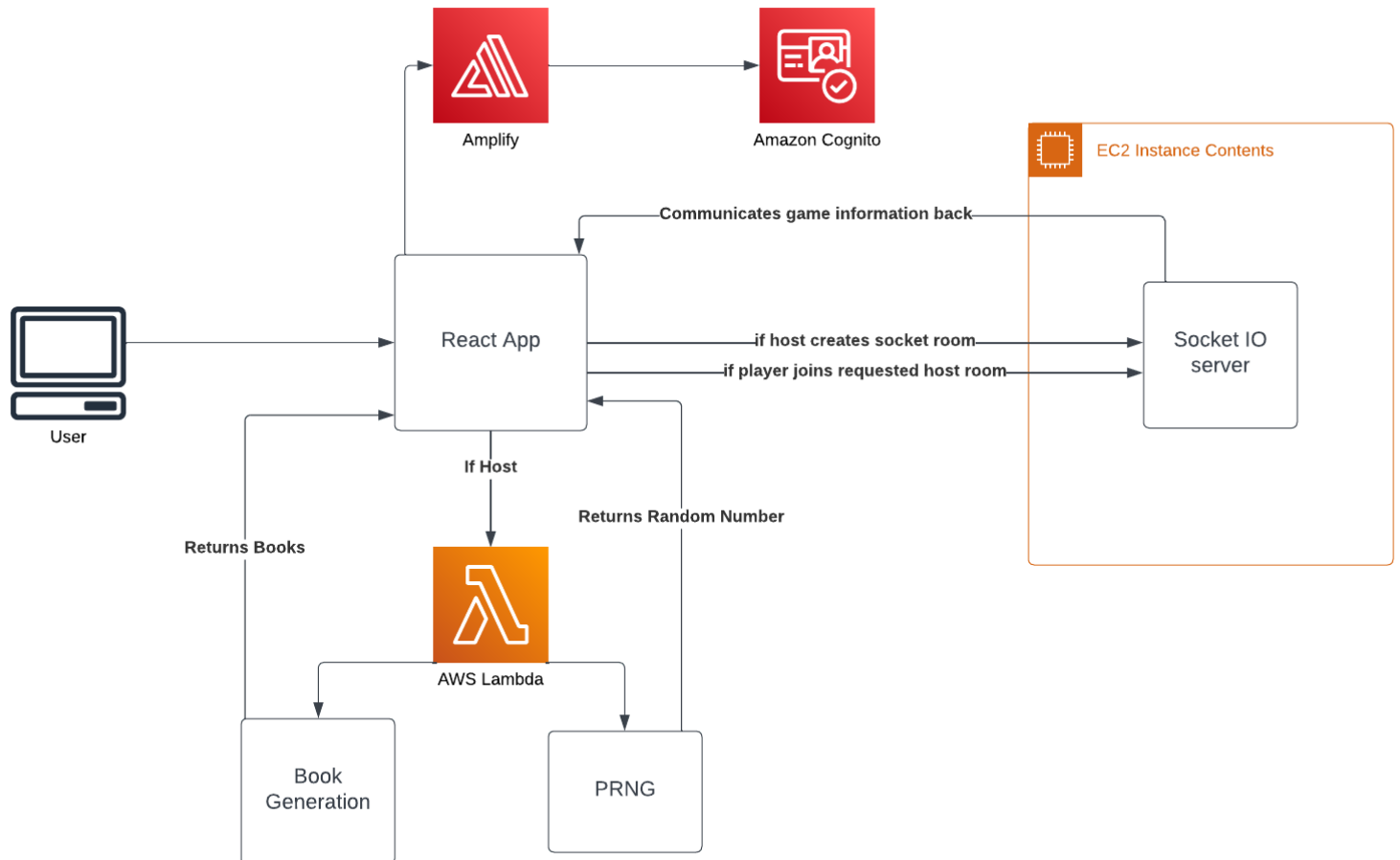
FORGOT PASSWORD

No account yet?

SIGN UP NOW

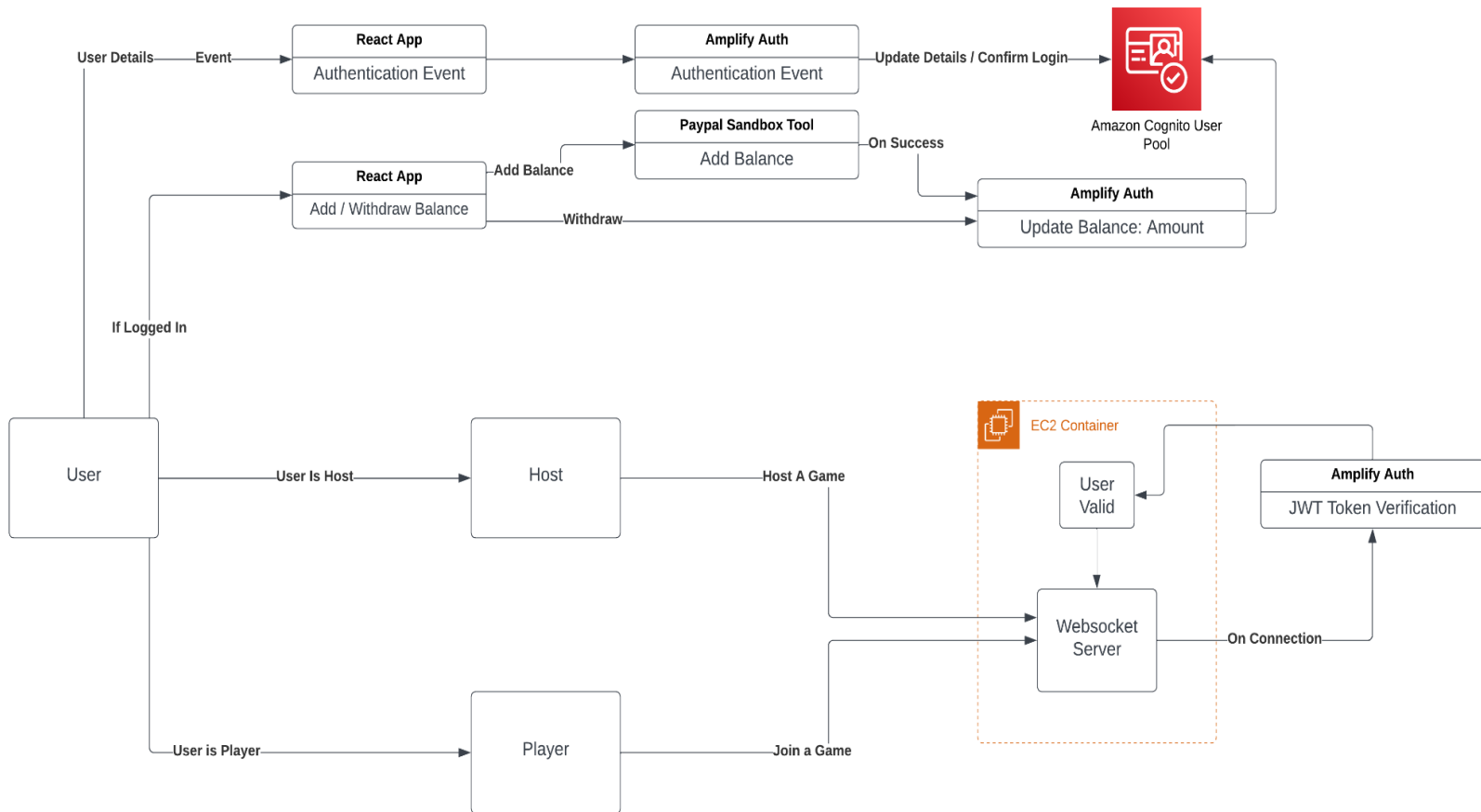
3. Design

3.1 System Architecture



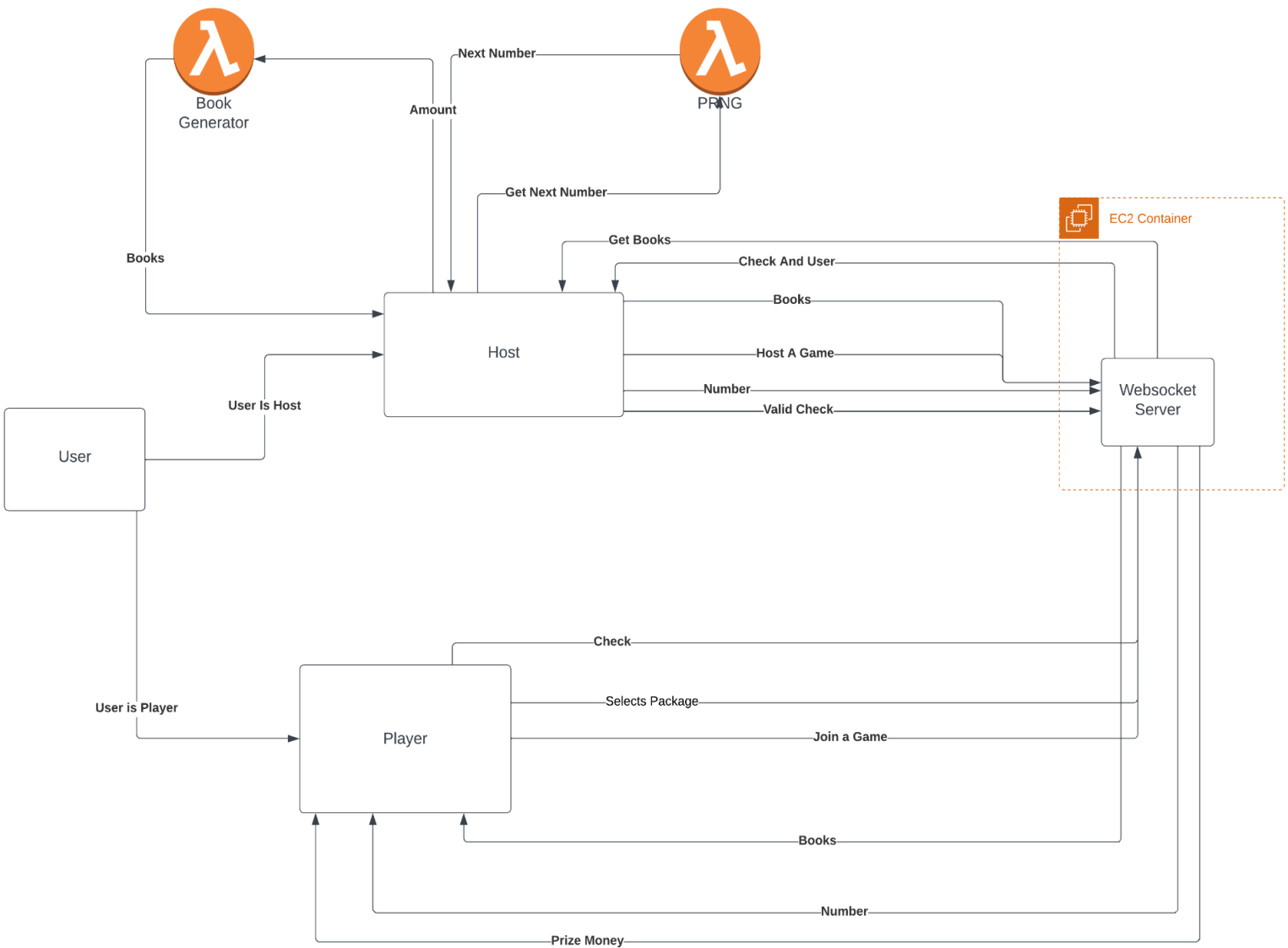
The Diagram above shows the system architecture of Bingo Bonanza, it shows all of its components from front-end react application, to back end with our socket io server, Lambda functions and our Cognito user pools. It shows a high level of the communication between components.

3.2 Data Flow Diagram (Outside Of Game)



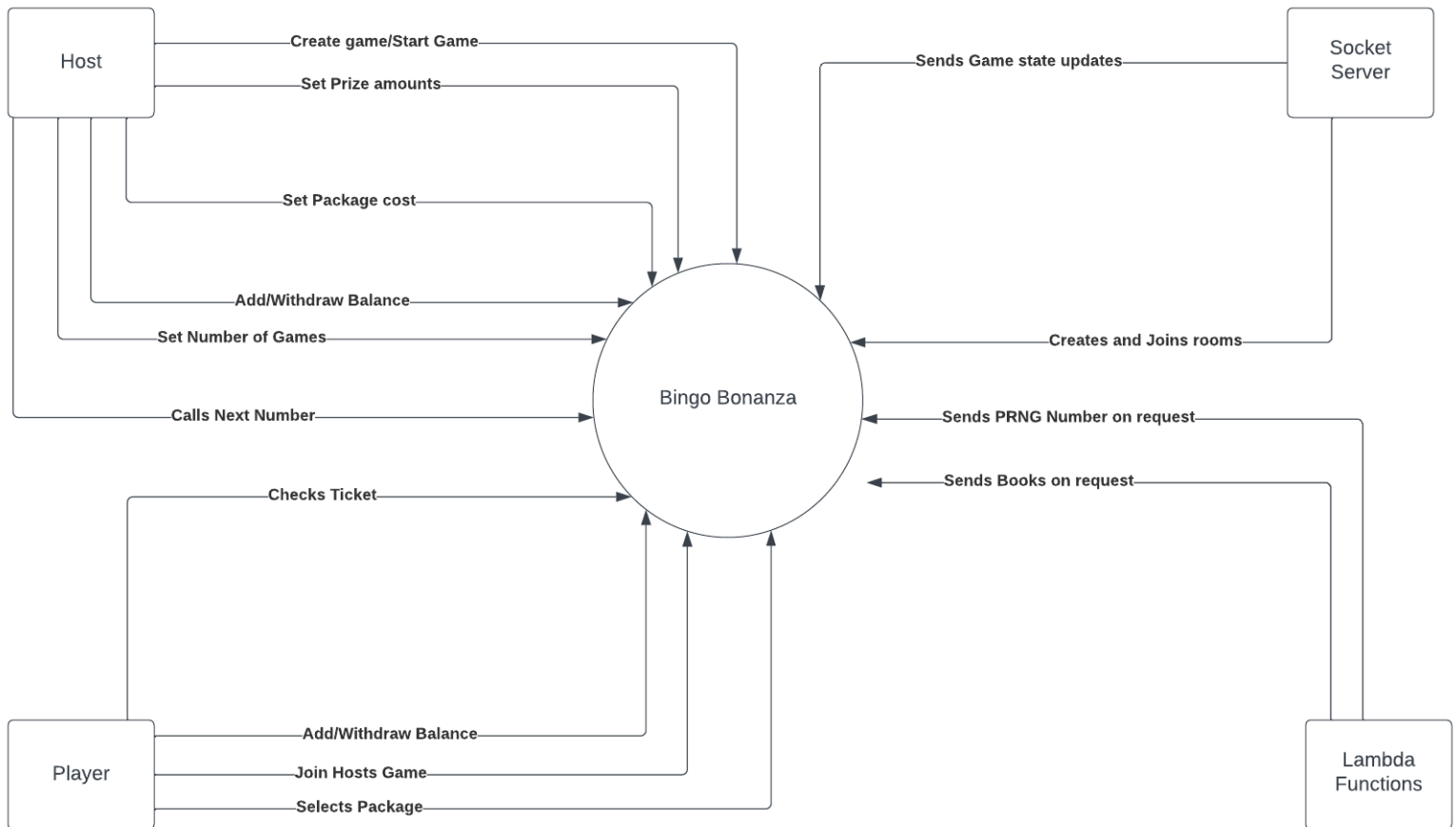
We decided that our Data flow diagram would be best served by splitting it into two parts as the amount of events data and events our Websocket Server would have in the diagram would make it unreadable. The above is the Data Flow Diagram outside of the game functionality. This shows how the Data within our application flows through the various components of our system architecture when user events occur.

3.3 Data Flow Diagram (Inside Of Game)



The above diagram shows how data flows within our game, events are triggered either by Users interacting with the application or by other events. This shows in a high level the events that the websocket sends and receives to both the Player and the Host during the game of bingo.

3.4 Context Diagram



The above diagram shows our context diagram, which is also sometimes called a level 0 data-flow diagram. It shows a high level flow of data between our components, it's displayed in a simple manner to be understood easily.

4. Implementation

4.1 Sign up/Login

The first page users will be greeted with is our sign up/ login page. The ui design of this page is quite simple as it needs to leave a good impression and this theme is passed on throughout the app.

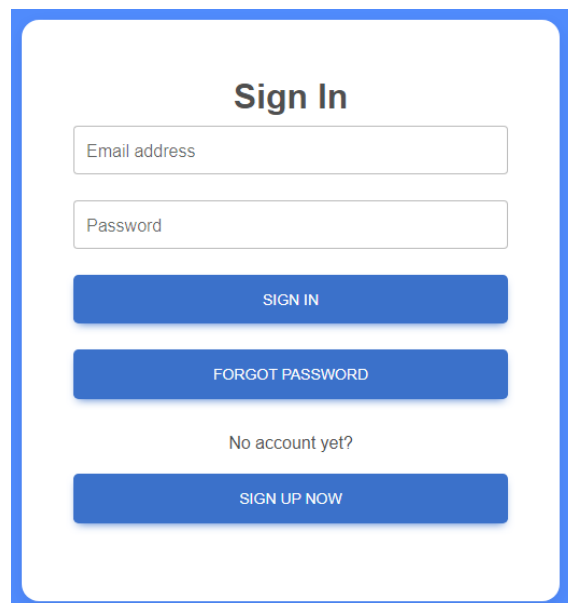
The Page consists of a few different components

- Sign In
- Sign Up
- Forgot Password
- Confirm SignUp
- Signed In

All of this authentication uses AWS Amplify Auth to handle the data and store it in an AWS Cognito User Pool.

Sign In

The sign in page consists of an email field and a password field, here if the user has already signed up can login using the email and password they created the account with.



The image shows a 'Sign In' form within a blue-bordered container. At the top, the title 'Sign In' is centered. Below it are two input fields: 'Email address' and 'Password'. Under the password field is a blue 'SIGN IN' button. Below that is a blue 'FORGOT PASSWORD' button. Further down is the text 'No account yet?' followed by a blue 'SIGN UP NOW' button.

Authentication.js

```
const initialState = {
  username: "",
  password: "",
  authCode: "",
  name: "",
  birthdate: "",
  city: "",
  UserType: "",
  balance: 0
}

function Authentication () {
  const [formState, updateFormState] = useState(initialFormState)
```

When the user enters their details in the input fields it updates the formstate initially set.

Authentication.js

```
const signIn = async () => {
  const {username, password} = formState;
  var LoggedIn = true;

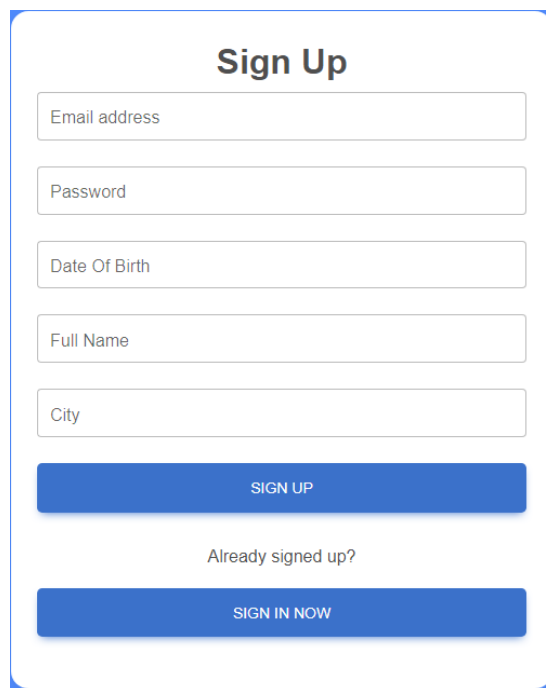
  try {
    await Auth.signIn(username, password);
  }
  catch (err) {
    window.alert(err)
    LoggedIn = false
  }
  if (LoggedIn === true) {
    updateUser(user);
    updateFormState(() => ({ ...formState, formType: 'signedIn'}));
  }
};
```

When the user hits sign in it calls our sign in function here we get the username and password from the formstate and we have a try and catch statements which will try to call (Auth.signIn) on the users entered details this Auth function is a provided function from aws-amplify to verify the users details are contained in our aws Cognito User Pool. If the users details are correct the catch does not trigger and we go to the if statement which will stay true and update the users state to the current user and will update the formstate to “signedIn”.

If the user's Login does not work it will trigger our catch which will change the logged in variable to false and provide a window alert of the error.

Sign Up

The sign up page contains a few fields that need to be filled out in order to register for the application.



The image shows a 'Sign Up' form with a blue border. At the top, the title 'Sign Up' is centered. Below it are five input fields: 'Email address', 'Password', 'Date Of Birth', 'Full Name', and 'City'. Each field has a light gray border and a small label inside. Below the fields is a large blue button with the text 'SIGN UP' in white. Underneath the button is a link that says 'Already signed up?'. At the bottom is another large blue button with the text 'SIGN IN NOW' in white.

Once the user fills in these fields it will update the formstate as shown in sign in. When the user clicks our sign up button it will trigger our signup function which contains a try and catch as before.

Authentication.js

```
const signUp = async () => {
  try {
    const {username, password, birthdate, name, city} = formState

    if (birthdate === "" || name === "" || city === "") {
      window.alert("Please Fill in all criteria to complete SignUp")
    }
    else {
      try {
        const {user} = await Auth.signUp({username, password, attributes: {birthdate, name, 'custom:city': city, 'custom:balance': '0'}})

        updateFormState(() => ({ ...formState, formType:"confirmSignUp"}));
      }
      catch(err) {
        console.log(err)
      }
    }
  }
}
```

This try grabs the users entered data from the formstate it will check to ensure that the fields that are not mandatory by our cognito users pool are also entered as they are required by us, we do this by checking if the formstate data for these fields are empty, if they are we send a window alert asking for all data to be filled in.

The else statement also contains try and catch, the code will try to call (Auth.signUp) which is provided by aws amplify if this succeeds it will update the formstate to “confirmSignUp”.

Authentication.js

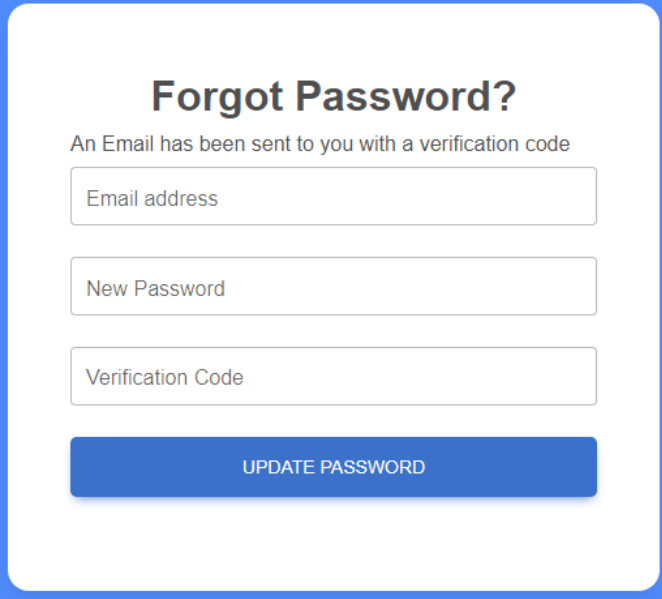
```
catch (err) {
  if (err.name === "InvalidParameterException") {
    window.alert("Please use a valid email address")
  }
  if (err.name === "UsernameExistsException") {
    var ResetPassword = window.confirm("This User already exists, would you like to reset your password?")
  }
  if (ResetPassword === true) {
    updateFormState(() => ({ ...formState, formType:"forgotPassword"}))
  }
  if (err.name === "AuthError") {
    window.alert("Please fill the entire before before registering")
  }
}
```

The code also has an except to go along with the try here we manage specific errors to ensure the user has an idea of what field may be causing them to not be able to sign up.

The if statement for the username already existing will update a variable called `resetpassword` and provide a window alert asking if they want to reset the password, if this is the case it updates the formstate to forgot password.

Forgot Password

The Forgot password page is accessed when a user enters their email address and hits the “forgot password button”.

A screenshot of a 'Forgot Password?' form. The form is enclosed in a blue border. At the top, the title 'Forgot Password?' is centered in bold. Below it, a message states 'An Email has been sent to you with a verification code'. There are three input fields: 'Email address', 'New Password', and 'Verification Code'. At the bottom, there is a blue button labeled 'UPDATE PASSWORD'.

Here users will have to enter their email address again and the password they wish to change to and the verification code sent to their email address. Once the user hits Update Password button it will trigger the `(confirmNewPassword)` function

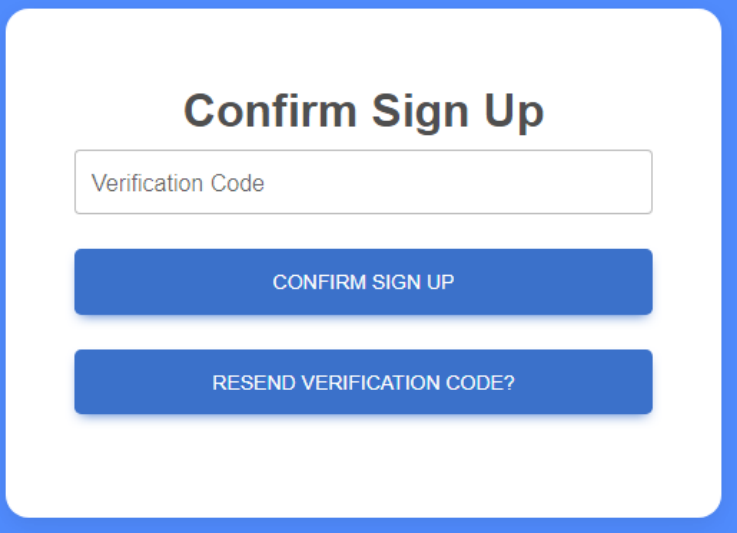
Authentication.js

```
const confirmNewPassword = async () => {  
  const {username, authCode, password} = formState  
  var PasswordChanged = true  
  
  try {  
    await Auth.forgotPasswordSubmit(username, authCode, password)  
  }  
  catch (err) {  
    window.alert(err)  
    PasswordChanged = false  
  }  
  if (PasswordChanged === true) {  
    updateFormState(() => ({ ...formState, formType: "signIn"}))  
  }  
};
```

Similar to other Auth functions we grab the user data from the form state and assign a variable “PasswordChanged” to true. We again have a try and catch which will attempt to call (Auth.forgotPasswordSubmit) to aws amplify if this succeeds it will move on to the if statement and update the formtype to signIn. If this fails it will provide a window alert and change the variable to false.

Confirm Sign-Up

The confirm Sign-Up is triggered after the sign up stage updates to the “confirmSignUp” stage.



The image shows a web form titled "Confirm Sign Up" enclosed in a blue border. At the top, the title "Confirm Sign Up" is centered in a bold, dark font. Below the title is a text input field with the placeholder text "Verification Code". Underneath the input field are two blue buttons with white text. The first button is labeled "CONFIRM SIGN UP" and the second button, positioned below the first, is labeled "RESEND VERIFICATION CODE?".

Here amplify auth automatically sends a Verification code to the users email, they then must enter that and click confirm sign up. This triggers the (confirmSignUp) function.

Authentication.js

```
const confirmSignUp = async () => {
  const { username, authCode } = formState;

  var ConfirmedSignUp = true
  try {
    await Auth.confirmSignUp(username, authCode);
  }
  catch (err) {
    window.alert("verification code does not match.")
    ConfirmedSignUp = false
  }
  if (ConfirmedSignUp === true) {
    updateFormState(() => ({ ...formState, formType: "signIn" }));
  }
};
```

Here we call (Auth.confirmSignUp) to verify the users auth code and confirm their verification. And then it will change the formstate to “signIn”.

The resend verification code button triggers a function similar which will then call

```
await Auth.resendSignUp(username).catch("username is not valid")
```

This code will use Auth to resend the code.

Signed In

This page will be shown once the user has verified their account and has logged into the application. Here the user will select their user type that they wish to proceed with.

A screenshot of a user selection screen. It features a white rounded rectangle with a blue border. Inside, the text "Which Type of User Would You Like to Be?" is centered at the top. Below this text are four blue buttons with white text, stacked vertically. The buttons are labeled "HOST", "PLAYER", "CONFIRM PLAYER TYPE?", and "SIGN OUT?".

Which Type of User Would You Like to Be?

HOST

PLAYER

CONFIRM PLAYER TYPE?

SIGN OUT?

Authentication.js

```
const signIn = async () => {
  var user = await Auth.currentAuthenticatedUser();
  var UserUpdated = true;
  const {UserType} = formState
  try {
    if (user.UserType !== {UserType}) {
      await Auth.updateUserAttributes(user, {'custom:UserType':UserType})
    }
    else {
      //pass;
    }
  }
  catch (err) {
    UserUpdated = false;
  }
  if (UserType !== "") {
    if (UserUpdated === true){
      user = await Auth.currentAuthenticatedUser();
      updateUser(user);
      localStorage.removeItem("isLoggedIn");
      localStorage.setItem("isLoggedIn", "true");
      window.location.reload();
    }
  }
  else {
    window.alert("Please select a User Type before clicking the Confirm Button")
  }
}
```

Once the user selects the confirmed user type it will trigger this function. Here we get the current user using the Auth function and we grab their usertype from the formstate. In the try we check to see if the user's usertype is not the same as the one they just selected if so we update the user's usertype in cognito using an Auth function (updateUserAttributes).

If the user's user type is not empty we call the (updateUser) function which updates the userstate to the current user in our react app. We then set a local storage item "isLoggedIn" to true and reload the page to access the main home page of the user type they selected.

If the user has no usertype it means they did not select one and so we will display a window alert error asking for them to pick a user type.

4.2 Configuring a Game

If the user has selected the Host usertype and wishes to host a game of bingo the first step they must take is configuring their game, this is a series of different web pages that a host must enter data on to set the configurable parameters for the game of bingo, these configurable parameters are:

- Number of games
- Cost of each package
- Prize Money for single line, double line and full house of each game

We use State and update the state to store these configurable parameters. React gives a lot of built in methods for not only manipulating the state but for creating code that only runs when the values of these states have been updated. The State is set to a base default when the webpage is first loaded. The configurationStage value within the React state determines the webpage the user sees.

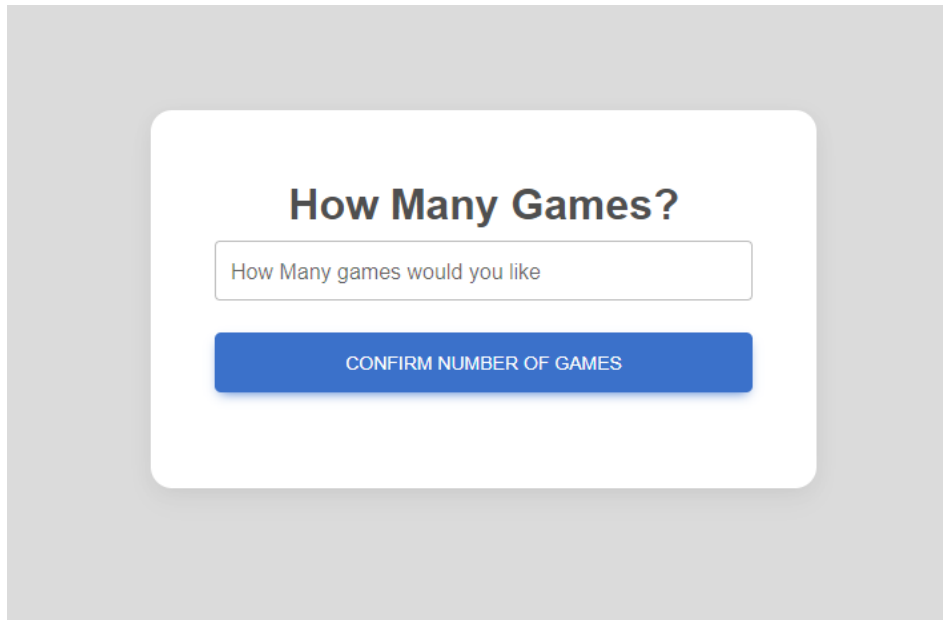
HostGame.js

```
const initialState = {
  currentGame: 0,
  numberOfGames: 0,
  CurrentPrizeSL: 0,
  CurrentPrizeDL: 0,
  CurrentPrizeFH: 0,
  TotalCost: 0,
  games: [],
  configurationStage: "HowManyGames",
  Package1: 0,
  Package2: 0,
  Package3: 0,
  Package4: 0,
  TotalSpend: 0,
}
```

Number Of Games

The number of Games pages is the first page that the host sees when they click the Host Game button within our Application. It is just a simple Container with a text

box, we have put in measures to ensure that when the submit button is clicked the user has only used a valid number and that the number is less than 20.



The image shows a web form titled "How Many Games?". It features a text input field with the placeholder text "How Many games would you like". Below the input field is a blue button labeled "CONFIRM NUMBER OF GAMES". The form is centered on a light gray background.

HostGame.js

```
const setNumberOfGames = async () => {
  const {numberOfGames} = gameState

  if (!isNaN(numberOfGames) && numberOfGames < 20 && numberOfGames !== "" && numberOfGames !== 0) {
    var i = 1

    if (games.length !== 0) {
      games.length = 0;
    }

    while (i <= numberOfGames) {
      var gameObject = SetGameObject("Game" + i);

      games.push(gameObject);

      i = i + 1;
    }

    updateGameState(() => ({ ...gameState, games: games}))
    updateGameState(() => ({ ...gameState, configurationStage: "WhatPrizeMoney"}))
  }
  else {
    window.alert("Must be a valid number and must be less than 20.")
  }
}
```

In this code we have an if condition to ensure four things:

- The User has entered a valid number (!isNaN(numberOfGame))
- The Number of games is less than 20
- The textbox was not blank
- The Number of games is not 0

If any of these conditions are not true the user is prompted with an error message to inform them that the entered data “Must be a valid number and must be less than 20”. If the conditions are satisfied in the if statement the application then proceeds to update the React State with an array of objects. Before this update occurs we ensure that the array is empty and then use a while loop and create the object for each game. The React State after the update is entirely complete is an array containing a series of objects. The object contains information such as gamename, first line prize, double line prize and full house.

HostGame.js

```
function SetGameObject (name) {  
  const currentGameObject = {  
    gamename: name,  
    PrizeFL: "",  
    PrizeDL: "",  
    PrizeFH: ""  
  }  
  
  return currentGameObject  
}
```

Prize Money

The Prize Money webpage is used to set a number amount as the prize money you wish users to get when they successfully check on that stage in that game of bingo. This webpage is just a series of containers containing text box entries and a submit button. The amount of containers there are directly relates to how many games you entered to play in the previous section. We use a javascript function map to ensure that this is true and use the previously updated React State for games. We use the index of their position within the array as the key for this map function.

HostGame.js

```
<HorizontalScroll>
  {games.map((game, index) => {
    return (
      <MDBContainer fluid>
        <MDBRow className='d-flex justify-content-center align-items-center vh-30'>
          <MDBCol col='12'>
            <MDBCard className='bg-white my-5 mx-auto' style={{ borderRadius: '1rem', maxWidth: '500px'}}>
              <MDBCardBody className='p-5 vw-30 d-flex flex-column'>
                <h2 key={games.gamename} className='fw-bold mb-2 text-center'>{game.gamename}</h2>
                <MDBInput wrapperClass='mb-4 vw-30' onChange={onPrizeMoneyChange} name='PrizeFL' label='Single Line Prize Money?' id='formControllg' />
                <MDBInput wrapperClass='mb-4 vw-30' onChange={onPrizeMoneyChange} name='PrizeDL' label='Double Line Prize Money?' id='formControllg' />
                <MDBInput wrapperClass='mb-4 vw-30' onChange={onPrizeMoneyChange} name='PrizeFH' label='Full House Prize Money?' id='formControllg' />
              </MDBCardBody>
            </MDBCard>
          </MDBCol>
        </MDBRow>
      </MDBContainer>
    );
  })}
</HorizontalScroll>
```

We also introduce a horizontal scroll wheel to allow all the forms to be displayed on the one page without causing issues.

When this data is fulfilled in and the form is submitted this triggers this event function:

HostGame.js

```

const setPrizeMoney = async () => {
  const {numberOfGames} = gameState
  var i = 0;
  var AllFilledIn = false;

  while (i < numberOfGames) {
    var FirstLine = gameState.games[i]["PrizeFL"]
    FirstLine = parseInt(FirstLine);
    var SecondLine = gameState.games[i]["PrizeDL"]
    SecondLine = parseInt(SecondLine)
    var ThirdLine = gameState.games[i]["PrizeFH"]
    ThirdLine = parseInt(ThirdLine)

    var PageCost = FirstLine + SecondLine + ThirdLine;

    TotalCost += PageCost

    localStorage.setItem("TotalCost", TotalCost)

    i+=1;
  }

  if (user.attributes['custom:balance'] < TotalCost) {
    window.alert("You don't have enough balance to host this game ensure the total prize money is not greater than your balance")
  }

  else {
    var NewBalanceNum = parseInt(user.attributes["custom:balance"]) - TotalCost

    updateGameState(() => ({...gameState, TotalCost: TotalCost}))

    var NewBalanceString = "" + NewBalanceNum

    await Auth.updateUserAttributes(user, {'custom:balance':NewBalanceString});

    updateGameState(() => ({...gameState, configurationStage: "SetPricing"}));
  }
}

```

This code then proceeds to update the previously made GameState.games array objects so that each “Prize” variable has a integer value that was input by the user before the submit button was pressed, we then must ensure that the Host has enough balance to fund this game to do this we sum the “Prize” values and then add them to a variable called total cost. We then use an if statement to ensure that the current user has a balance larger than the total cost if they do the React State is updated and they proceed to the next page if not a Window Alert is used to inform the Host that they do not have enough balance to Host this game.

Cost Of Each Package

The cost of each package page, like its predecessors in the configuration of a game, is a series of four containers each containing a text box followed by a submit button for when these values are filled in and the Host wishes to proceed to the

next page. This works in a similar manner to How Many Games so I will not go into detail about this code.

4.3 Hosting a Game

Hosting a game is handled by connecting to our socket io server.

HostGame.js

```
if (gameState.configurationStage == "GameStart" && isWebSocket == null) {  
  const { io } = require("socket.io-client");  
  
  const socket = io("ws://ec2-52-211-225-16.eu-west-1.compute.amazonaws.com:1025/", {  
    extraHeaders: {  
      "useridtoken": user.signInUserSession.idToken.jwtToken,  
      "usertype": user.attributes['custom:UserType'],  
      "username": user.attributes.email,  
      "package1": gameState.Package1,  
      "package2": gameState.Package2,  
      "package3": gameState.Package3,  
      "package4": gameState.Package4,  
    }  
  });  
  
  updateIsWebSocket(socket);  
}
```

Here we create a const called socket which will connect to our ec2 instance of our socket server on port :1025. The extra headers pass extraHeaders to the server that contains the details of the game that was just configured.

HostGame.js

```
<div className="tableFixHead">
  <table className="table">
    <thead>
      <tr class="table-light">
        <th scope="col">#</th>
        <th scope="col">Username</th>
      </tr>
    </thead>
    <tbody>

    {PlayersInSession.map((player, index) => {
      return (
        <tr class="table-light">
          <th scope="row">{index + 1}</th>
          <td>{player}</td>
        </tr>
      )
    })}

    </tbody>
  </table>
</div>
</MDBCardBody>
```

This table is shown on the waiting for players page; it maps all the players that have connected to the Host's socket room so the host can see who is ready to join.

#	Username
1	thesimms89@gmail.com

Here the Host can start the game once all the players have joined.

This will update the game stage to “completed” and the game board will be shown and the game will begin.

4.4 Playing a Game

They player who joins a game will be given their books depending on the package selected, to display the best ticket available to them ie. the one closest to completion for this stage of the game we developed a grading algorithm

JoinGame.js

```
function DetermineBestTicket (Stage) {  
  var Multiplier = 0  
  var ScoreArray = []  
  
  if (gameState.SelectedPackage === "Package1") {  
    Multiplier = 1  
  }  
  
  if (gameState.SelectedPackage === "Package2") {  
    Multiplier = 2  
  }  
  
  if (gameState.SelectedPackage === "Package3") {  
    Multiplier = 3  
  }  
  
  if (gameState.SelectedPackage === "Package4") {  
    Multiplier = 4  
  }  
  
  if (Stage === "FirstLine") {  
    var BestTicket = BooksForGame.filter(books => {  
      books.filter(tickets => {  
        var Score = 0  
        tickets.filter(lines => {  
          var i = 0  
          Score = 0  
          while (i < 5) {  
            if (Numbers.includes(parseInt(lines[i]))) {  
              Score += 1  
            }  
            i += 1  
          }  
          ScoreArray.push(Score)  
        })  
      })  
    })  
  }  
}
```

This code is for scoring when the game stage is “FirstLine”

Here we filter through the books and then filter through the tickets, a while loop then iterates through the line and increments the score if the number that has been called out is in the line.

JoinGame.js

```
if (Stage === "DoubleLine") {  
  var BestTicket = BooksForGame.filter(books => {  
    books.filter(tickets => {  
      var FirstScore = 0  
      var SecondScore = 0  
      var ThirdScore = 0  
      var x = 0  
  
      while (x < 3) {  
        var i = 0  
        while (i < 5) {  
          if (Numbers.includes(parseInt(tickets[x][i]))) {  
            if (x === 0) {  
              FirstScore += 1  
            }  
            if (x === 1) {  
              SecondScore += 1  
            }  
            if (x === 2) {  
              ThirdScore += 1  
            }  
          }  
          i += 1  
        }  
        x += 1  
      }  
    }  
  }  
  
  var OneAndTwo = FirstScore + SecondScore  
  var OneAndThree = FirstScore + ThirdScore  
  var TwoAndThree = SecondScore + ThirdScore  
  
  var Score = Math.max(OneAndTwo, OneAndThree, TwoAndThree)  
  
  ScoreArray.push(Score)  
  
  console.log(ScoreArray)  
})  
}
```

This is the Double line scoring for when the game is in the “DoubleLine” stage. Here we again filter through the books and the tickets to get to the lines. Our while loop of (X < 3) is designed to go through each line of the ticket and keep track of it so that if the number that has been called out is on one of those lines we will increment the score for that line. For example line one and two or line one and three.

The score is then determined by `math.max` of these three scores to see which one is the highest.

JoinGame.js

```
if (Stage === "FullHouse") {
  var BestTicket = BooksForGame.filter(books => {
    books.filter(tickets => {
      var Score = 0
      var i = 0
      var x = 0

      while (x < 3) {
        var i = 0
        while (i < 5) {
          if (Numbers.includes(parseInt(tickets[x][i]))) {
            Score += 1
          }
          i += 1
        }
        x += 1
      }
      ScoreArray.push(Score)
    })
  })
}
```

This is the scoring for the “FullHouse” stage. Here we filter through the books and filter through the tickets the same as last time with the “while (x<3)” we increment if the number is in that line as we are just waiting for the book to be complete.

JoinGame.js

```
var i = ScoreArray.indexOf(Math.max(...ScoreArray));
```

Here we get the index of the highest score inside of the score array. If they have the same score it will take the first score in the index.

JoinGame.js

```
if (Stage === "FirstLine") {  
  if (i < 18) {  
    FirstIndex = 0  
    if (i < 3) {  
      SecondIndex = 0  
    }  
    else if (i < 6) {  
      SecondIndex = 1  
    }  
    else if (i < 9) {  
      SecondIndex = 2  
    }  
    else if (i < 12) {  
      SecondIndex = 3  
    }  
    else if (i < 15) {  
      SecondIndex = 4  
    }  
    else if (i < 18) {  
      SecondIndex = 5  
    }  
  }  
}
```

Here we have a bunch of if statements that will get the index of the ticket that has the highest score that goes from (i < 18) all the way to (i < 72).

The first index is what book the best ticket is in, and the second index is what ticket it is in that book.

JoinGame.js

```
return BooksForGame[FirstIndex][SecondIndex]
```

Here we then return the index of the ticket with the best score.

The next bit of code I will talk about is a function that will update the balls you are waiting on in the UI.

JoinGame.js

```
function updateWaitingUI() {
    var i = 0

    if (WaitingNumbers.length !== 0){
        while (i < WaitingNumbers.length){
            var waitingballGraphicElement = document.querySelectorAll("[id='waitingballGraphic']")
            var waitingballGraphicText = document.querySelectorAll("[id='waitingballText']")
            console.log(waitingballGraphicText)

            for(var x = 0; x < waitingballGraphicElement.length; x++){

                for(var z = 0; z < waitingballGraphicText.length; z++){
                    if (parseInt(waitingballGraphicText[z].innerText) < 10 ){
                        waitingballGraphicText[z].className = "single"
                    }
                    else {
                        waitingballGraphicText[z].className = ""
                    }
                }

                if (WaitingNumbers[i] < 15){
                    waitingballGraphicElement[x].className = "valign-wrapper blue"
                }
                else if (WaitingNumbers[i] < 30){
                    waitingballGraphicElement[x].className = "valign-wrapper green"
                }
                else if (WaitingNumbers[i] < 45){
                    waitingballGraphicElement[x].className = "valign-wrapper orange"
                }
            }
            i++
        }
    }
}
```

Here we loop through the numbers you are waiting for, we then get the html element with the id of “waitingballGraphic” and “waitingballText” and put all instances of these elements in a list. From this we then loop through that list to get each of those “waitingballGraphic” elements. The next for loop will check if the number inside the waiting list is single or not as if its single digit the balls sizing needs to be updated to a classname single.

For the waitingballGrapics we just change the classname to have the additional blue/green etc to change the colour of the balls.

4.5 Book Generation

Our book generation algorithm exists as an AWS Lambda function; it is one of two Lambda functions we use within our project both of which are written in Python. Our Book Generation consists of one algorithm that is responsible for dispersing a List containing all the numbers from 1-90 into a set of Lists embedded within each other where the most embedded List is a List containing 5 integer values. The Numbers are selected at random.

I will not go through every part of this code but will rather focus on the parts that gave us trouble or constitute rules that can't be broken in terms of Bingo Tickets.

The first part of this code that sets out an unbreakable rule is this code:

Book_Generator.py

```
# Calculates the Lower Limits of the number (x) and returns it, the Lower limits are defined as rounding down to the nearest 10 or 0 and adding one to it
#This is used to ensure that two numbers of the same lowerlimit cannot be appended to the same line of a ticket i.e 1 and 7 cannot be on the same line
def CalculateLimits(x):
    # Gives the Second digit of the number example: 87 % 10 = 7
    x_mod = x % 10

    # If statement to ensure when mod = 0 lower limit is still calculated correctly
    if x_mod != 0:
        # Calculates the number (x) lower limit by taking x_mod away from x and adding 1
        x_lower_limit = (x - x_mod) + 1
    else:
        # Calculates the lower limit by taking away 9 from original number x
        x_lower_limit = x - 9

    return x_lower_limit

# If an item in the list belongs to the same range as our number to be put into the list
if CalculateLimits(item) == CalculateLimits(number):
    # Set Variable range_exists
    range_exists = True
# If the range does exist do this
if range_exists == True:
    number = secrets.choice(Numbers)
```

The first snippet above is used to calculate the range the number is contained in the ranges are (1-10, 11-20 and so on). The second snippet of code is used to ensure that no line on a ticket contains two numbers that fall within the same range. This is an essential rule in bingo and cannot be broken.

When we were implementing this code we were having an issue where every couple of runs the code would get stuck in an infinite loop. This issue was caused as when the program gets to the final ticket there are only 15 numbers left in the original list of 1-90. As aforementioned it is a golden rule that no two numbers on an individual line can fall in the same number range, therefore it tracks to assert that when on the final ticket if the List containing numbers has 4 numbers that fall within the same number range it will be impossible to both complete the final ticket and obey the rule. To prevent this infinite loop from happening we needed to run a preemptive check to ensure that no 4 numbers of the remaining 15. This error was also replicated on the second line final ticket so we needed to replicate the preemptive check but ensuring 3 numbers don't exist within the same range. If these checks fail the data structures are cleared and the process is restarted. The code for that is as follows:

Book_Generator.py

```
# Checks the Possibility of a Ticket being able to be created for the final ticket by comparing the LowerLimits for each number in the list to all the numbers in the list
# If there are more than 4 instances where this comparison is true it is impossible to generate the final ticket
def CheckPossibility(Numbers, x):

    i = 0
    Possible = True
    while i < len(Numbers):
        counter = 0
        tmp = Numbers[i]
        for item in Numbers:
            if CalculateLimits(tmp) == CalculateLimits(item):
                counter += 1

        if counter >= x:
            Possible = False

        i += 1

    if Possible == True:
        return True

    else:
        return False

# If on the final run of PopulaateTickets
if len(Numbers) == 15:
    # If it is possible
    if CheckPossibility(Numbers, 4) == True:
        pass
    #If its not possible restart the entire process
    else:
        Restart(Book, Ticket_One, Ticket_Two, Ticket_Three, Ticket_Four, Ticket_Five, Ticket_Six)
        Numbers.clear()
        new_number_list = FillNumbersList()
        for item in new_number_list:
            Numbers.append(item)
```

4.6 Pseudo Random Number Generator

As mentioned previously in this our PRNG has three different implementations: Blum Blum Shub, LCG and XORshift*. This is our second Lambda function we use within our project. I won't touch on all the parts of the code within this implementation but will rather focus on the interesting or essential parts of the code.

```
def Decide_Algorithm():  
  
    Algorithms = []  
  
    i = 0  
    while i < 100:  
        Algorithms.append("LCG")  
        Algorithms.append("XOR")  
        Algorithms.append("BBS")  
  
        i += 1  
  
    algorithm = secrets.choice(Algorithms)  
  
    if algorithm == "LCG":  
        result = LCG_Main()  
        # Used for generating Sequence for testing  
        # result = LCG_Test()  
        return result  
  
    elif algorithm == "BBS":  
        result = BBS_Main()  
        # Used for generating Sequence for testing  
        # result = BBS_Test()  
        return result  
  
    elif algorithm == "XOR":  
        result = XORStarMain()  
        # Used for generating Sequence for testing  
        # result = LCG_Test()  
        return result
```

This snippet determines what algorithm is used. This is just done by appending 100 counts of a string that represents one of the three algorithms to a list. Then we use the Python secrets library to select one at random and the if/else if statement matches it to the correct algorithm then runs it.

Our implementations of each of the PRNG algorithms are commented throughout and you can reference them on any questions you may have about the individual algorithm and how it works.

4.7 Websocket Server

The websocket is one of our most essential parts of our application, it is essential as without it we cannot provide full duplex communication between our host and player. For our websockets we decided to use the Socket.io framework to allow us to create rooms for individual players, while we did like this framework and found it useful we found some issues with it, we had instances where if you disconnected and attempted to reconnect instead of removing the old socket connection and replacing with the new one both connections would just persist, to try to combat this we attempted to implement our own objects that tracked rooms and the players connected to it. Using this we were able to just update our internal object with the new SocketID and keep this information as up to date as possible as it allowed us to know where to send information at certain points.

```
const Rooms = []

const templateRoomObject = {
  roomName: "",
  roomHostSocketID: "",
  Players: [],
  Package1: 0,
  Package2: 0,
  Package3: 0,
  Package4: 0,
}

function RoomObjectInstance (arg1, arg2, arg3, arg4, arg5, arg6) {
  const RoomObject = {
    roomName: arg1,
    roomHostSocketID: arg2,
    Players: [],
    Package1: arg3,
    Package2: arg4,
    Package3: arg5,
    Package4: arg6,
  }

  return RoomObject
}
```


We also learned that it was possible to pass information and bind it with the particular websocket using the TCP protocol of the handshake stage. This was incredibly useful for us as it allowed us to safeguard some of the events that our websocket server used by limiting it to only specific user types.

```
io.on("connection", (socket) => {  
  Package1 = socket.handshake.headers.package1  
  
  Package2 = socket.handshake.headers.package2  
  
  Package3 = socket.handshake.headers.package3  
  
  Package4 = socket.handshake.headers.package4  
  
  username = socket.handshake.headers.username;  
  
  usertype = socket.handshake.headers.usertype;  
  
  AccessToken = socket.handshake.headers.useridtoken;  
  
  RoomToJoin = socket.handshake.headers.roomtojoin  
  
  SocketID = socket.id  
})
```

Another interesting aspect of our is our AWS JWT token aspect, this uses a node module to give us the functionality to confirm that the JWT token passed during the sockets connection stage is indeed issued from a our Cognito userpool, this is especially useful as it provides an extra layer of security that no connections can be formed to our websocket server without first having a valid JWT token.

```
async function Verify () {
  const {payload} = decomposeUnverifiedJwt(AccessToken)

  var clientId = "" + payload.aud;

  const verifier = CognitoJwtVerifier.create({
    userPoolId: "eu-west-1_FYm9eQUYe",
    tokenUse: "id",
    clientId: clientId
  });

  try {
    await verifier.verify(
      AccessToken
    );
  }

  catch (err) {
    console.log(err);
    isValidUser = false;
  }
}
```

5. Problems We Encountered & How We Solved Them

5.1 Websocket Server Implementation

Websockets proved to be quite difficult in the beginning, as neither of us had much experience with websockets or in particular websockets in react. In the beginning we attempted to use Node.js websockets and we quickly realised that using Socket IO would be a better solution for two reasons, it had more documentation and was also more of an industry standard socket package.

Once we had decided on socket IO our inexperience using websockets caused us some real issues. When following the documentation we could not get the sockets to work properly as we were attempting to run the socket server on the host's client side. Our reasoning for this was that we were developing a serverless project and our backend up to this point was being handled by our lambda functions.

We quickly learned that this is not the way that socket servers work and they need a separate file to run a server in a terminal. As we did not want to have a hosted server we decided to see if we could run two processes on the same terminal ie. basically run our react and socket server on the same terminal, and although we found some ways to do something similar ultimately we knew we needed to set up a hosted server on AWS. This process was quite easy to do. We set up an ec2 instance and cloned a github repository that contained our socket server and connected the sockets.

5.2 Book Generation Issues

I mentioned previously some of the issues we had in our Book generation, the main issue being that we would be stuck in an infinite loop occasionally when the program was run, we figured out that this was due to the rules that we created when pushing numbers into their relative array. It is a standard rule that no two numbers that fall in the same range (1-10, 11-20, ... so on) can be present on the same line of a ticket. We enforced this rule but there were instances on the final ticket of a book when only 15 numbers were left in our original List of 1-90.

If 4 of these numbers fall within the same range at this stage the program would infinitely loop, it was also an issue entering the second line of the final ticket where if 3 numbers fall within the same range the infinite loop occurs. To prevent this we created a function that checked if it was possible to complete the book. We did this by running a comparison of every item in the list vs all the other items in the list, if

both values had the same range a counter was incremented, if this counter was then greater or equal to the second parameter of the function the program would return false otherwise it would return true.

```
# Checks the Possibility of a Ticket being able to be created for the final ticket by comparing the LowerLimits
# If there are more than 4 instances where this comparison is true it is impossible to generate the final ticket
def CheckPossibility(Numbers, x):

    i = 0
    Possible = True
    while i < len(Numbers):
        counter = 0
        tmp = Numbers[i]
        for item in Numbers:
            if CalculateLimits(tmp) == CalculateLimits(item):
                counter += 1

            if counter >= x:
                Possible = False

        i += 1

    if Possible == True:
        return True

    else:
        return False
```

If this function results in True we continue the computation as intended, otherwise if it is false the data structures are cleared and reset and the program is restarted.

```
if len(Numbers) == 15:
    # If it is possible
    if CheckPossibility(Numbers, 4) == True:
        pass
    #If its not possible restart the entire process
    else:
        Restart(Book, Ticket_One, Ticket_Two, Ticket_Three, Ticket_Four, Ticket_Five, Ticket_Six)
        Numbers.clear()
        new_number_list = FillNumbersList()
        for item in new_number_list:
            Numbers.append(item)

first_line = []
second_line = []
third_line = []
```

5.3 Issues With React State

We store a lot of our important variables within React State and use this for future calculations or functions within our application, we had instances of issues where after updating our State the update would not be made instantly but rather the

update would be staged for a later render by React itself, we initially tried to find a way to force this update to be instantaneous but we quickly realised that this was virtually impossible to do so we decided the best way to do this was rather use Global variables outside of the React Components functions. This allowed us to update these variables in real time and then use the updated values instantaneously. This became especially important when playing multiple games as our function that updated the “BooksForCurrentGame” Variable would not be accessible as

soon as it was updated.

```
function BooksForCurrentGame (CurrentGame) {  
  var Index = 0  
  var StartPoint = 0  
  if (gameState.SelectedPackage === "Package1") {  
    Index = 3  
    if (CurrentGame !== 1) {  
      Index = 3 + ((CurrentGame - 1) * 3)  
      StartPoint = 3 * (CurrentGame - 1)  
    }  
  }  
  if (gameState.SelectedPackage === "Package2") {  
    Index = 6  
    if (CurrentGame !== 1) {  
      Index = 6 + ((CurrentGame - 1) * 6)  
      StartPoint = 6 * (CurrentGame - 1)  
    }  
  }  
  if (gameState.SelectedPackage === "Package3") {  
    Index = 9  
    if (CurrentGame !== 1) {  
      Index = 9 + ((CurrentGame - 1) * 9)  
      StartPoint = 9 * (CurrentGame - 1)  
    }  
  }  
  if (gameState.SelectedPackage === "Package4") {  
    Index = 12  
    if (CurrentGame !== 1) {  
      Index = 12 + ((CurrentGame - 1) * 12)  
      StartPoint = 12 * (CurrentGame - 1)  
    }  
  }  
  
  TempBooksForGame = gameState.books.slice(StartPoint, Index)  
  
  return TempBooksForGame  
}
```

The code above previously updated `gameState.BooksForCurrentGame` until we updated it to use a global variable rather than the react state. This entirely fixed the issue we encountered.

5.4 Sending Arrays over the Sockets

We generate the books for each player on the hosts side and send them via websockets to the appropriate player. We do this to protect our Lambda function as much as possible by only exposing our Public URL in one section of the code. We had issues in attempting to send the players books over the array due to their complex nature (at least 270 pieces of data - 1080 pieces of data for just one game). To solve this issue we compose a piece of code that would be able to deconstruct the array into a string and send that string across the websocket server. This code was complex due to the nature of our embedded arrays. Once the array was deconstructed it could be reconstructed on the opposite side in a similar manner.

```

function ArrayToString(arg1, Books) {
    var HowMany = 0;

    var i = 0

    if (arg1 === "Package1") {
        HowMany = 3
        HowMany = HowMany * gameState.games.length
    }
    if (arg1 === "Package2") {
        HowMany = 6
        HowMany = HowMany * gameState.games.length
    }
    if (arg1 === "Package3") {
        HowMany = 9
        HowMany = HowMany * gameState.games.length
    }
    if (arg1 === "Package4") {
        HowMany = 12
        HowMany = HowMany * gameState.games.length
    }

    console.log(Books)

    var ArrayString = ""

    while (i < HowMany) {
        var x = 0
        while (x < 6) {
            var z = 0;
            while (z < 3) {
                var a = 0
                while (a < 5) {
                    ArrayString = ArrayString + " " + Books[0][i][x][z][a]
                    a += 1
                }
                z += 1
            }
            x +=1
        }
        i += 1
    }

    return ArrayString
}

```

This code works by first equating which package the user has as this is important

for the second index in our Books Array as the upper limit will be equal to the value equated by multiplying the value assigned based on books by the total number of games. After that we begin by using a while loop that works while the variable is less than the "HowMany" variable calculated previously. The next while loop works while the variable is less than 6 as this is the number of tickets in each book. The next while loop works while the variable is less than 3 this is used to indicate the three lines off each ticket. After this the final while loop works while the variable is less than 5, this represents each number in the line. We then let a variable "ArrayString" equal itself + " " + The integer value at the Books variable index [0][i][x][z][a] where i, x, z and a are the variables used for each while loop.

The next step of this process is then the reconstruction on the Player side of the application this was slightly easier than the deconstruction of the array.

```
socket.once("SendBooks", (ArrayString, HowMany) => {
  var NumberOfBooks = 0

  if (HowMany === "Package1") {
    NumberOfBooks = 3
  }
  if (HowMany === "Package2") {
    NumberOfBooks = 6
  }
  if (HowMany === "Package3") {
    NumberOfBooks = 9
  }
  if (HowMany === "Package4") {
    NumberOfBooks = 12
  }

  const temp = ArrayString.split(' ')
  var CurrentLine = []
  var CurrentTicket = []
  var CurrentBook = []
  var AllBooks = []
  var x = 1

  while (x < temp.length) {
    if (x !== 0) {
      CurrentLine.push(temp[x])
    }
    if (x % 5 === 0) {
      CurrentTicket.push(CurrentLine)
      CurrentLine = []
    }

    if (x % 15 === 0) {
      CurrentBook.push(CurrentTicket)
      CurrentTicket = []
    }

    if (x % 90 === 0) {
      AllBooks.push(CurrentBook)
      CurrentBook = []
    }
    x += 1
  }

  Books.push(AllBooks)
})
```

This code works by splitting the string based on where there are spaces “ “. This creates an index of all integer values that were contained within the Books array in the order they could be found originally. We then use a while loop for a variable x while it is less than the total number of indexes after the array is split. After this all we need to do is use temporary arrays and push these into each other at specific stages, the specific stages are indicated by the if statements, the first index is a blank string so we avoid pushing that after that we push to current Line, when the variable $X \% 5 === 0$ (indicative of a Line) we push this array into a ticket array and then clear the original CurrentLine array, after that we use the next if statement to represent tickets when $x \% 15 === 0$ we push this array into a Book array and clear the current ticket array. When $x \% 90 === 0$ we push the Book array into the All Books array and clear the Book array. This is done for all values and when it is complete the array constructed will be the exact replica of the array that was generated on the Host side.

6. Results & Conclusion

6.1 Directions for future work

Introducing Reporting

We could introduce a reporting system for Hosts where we store relevant information for a session within a CSV file that is saved on the Hosts local machine. This CSV file could contain information like: How Many Players in the Session, Total Cost for the Host, Total Sales of Packages, Winners for each prize in a session and more. This CSV file could then be searched for through a specific path and displayed within our reporting application where all CSV files are selectable and clicking one opens a HTML file that displays the data stored in a clear and easy to understand manner.

This would be important for our app as a lot of the intended hosts for our application are the businesses who wish to use this rather than traditional bingo software companies, this reporting is essential for those businesses to be able to know the money they made/lost on a night and what their attendance is from event to event so that they can plan and prepare for the future and keep accurate track of the money made every session.

Player Profiles

We currently have a placeholder page in our web application for player profiles but this would not be impossible or difficult to implement. We could create a database that replicates some of the players information from our Cognito user pool such as username, address, email, etc. using a Lambda function. After this replication is completed every time a player plays a game we can store significant stats such as the total number of games played, total prizes won and more to a stat section within their profile. This would give the user a sense of ownership with their account and might encourage more engagement with the application. We could also allow them to set profile pictures and potentially introduce a friendship system like social media applications have through the profile.

Adding Fault Tolerance

Due to the nature of our project the entire application can be described as a distributed system by the fact that 3 major components within our application are spread out over 2 lambda functions and an EC2 instance. There is the possibility that these services become unavailable for some reason or another. We learned in our module CA4006 - Concurrent and Distributed programming that when speaking about fault tolerance within a distributed system a good industry standard to help improve fault tolerance is creating multiple copies of servers with distributed systems on them that information is replicated to so that if a system does go down there is still an available version online with the most up to date information possible. By introducing such a system within our application we would not only improve the fault tolerance of our application but also improve the performance of the system when multiple clients are connected to it by redirecting traffic, after a certain threshold is hit on our first instance, to the backup servers.

To support this we would need some form of a Message Queueing system so that the two instances of the server could communicate with each other, to do this we could use RabbitMQ which is an industry standard for Message Queueing systems, we could in theory use a appropriate middleware alternatively such as MPI (Message Passing Interface).

Security Refinements

Due to the nature of this project Security was a key part of the design process and was constantly considered during the project, especially when speaking about our Websocket server and our Lambda functions. Although we introduced features to protect these components there is definitely further work that could be done to improve the security associated with these applications. Within the Websocket server we would need to introduce an element that prevents people from being able to generate false events and sending them to the socket server, this could potentially negatively impact our websocket servers implementation so it would be important to protect, we could do this by ensuring that all events have a signature that is generated by the JoinGame/HostGame Javascript files if the signature is not as expected we could just refuse the event on the websocket server.

Another Security refinement we could make relates to our lambda functions, this would be to prevent people calling these functions and decreasing their throughput and latency by putting an immense workload on the Lambda functions. We could implement a similar ideology to this in the one suggested for the websocket server where we use a signature on the code and if the signature does not match the expected signature the lambda function does not run.

6.2 Conclusion

We consider our project, Bingo Bonanza, a huge success. We set out with the objective of being able to create a distributed system that was capable of allowing users to host and play bingo all within the same application. We believe we achieved this goal to a high standard and we learned a lot throughout the process of designing this application. We learned how to design lambda functions and how to use those lambda functions inside of an application, we learned about working with React and designing a React application, we learned about Websocket Servers in terms of how they function and how to use them and we learned about general project development. All of what we learned has greatly increased our confidence in our personal developer skills.

We were satisfied with the time complexity of the application overall especially when considering some of the complex data sets we were working with. The longest part of our project is actually in getting the books generated from our Lambda function and when we ran a test of producing 50,000 books the program was capable of doing this in just under 1 minute which we would deem as a more than acceptable time complexity for this issue. We know this application can be improved and may not be of a high enough standard to compete with the other bingo software companies but we are very proud of the application as it stands right now considering the time spent in development and the fact that two people carried out this work compared to a team like the aforementioned companies would have. We hope to keep developing this project in the future but are delighted with what we have learned and the confidence we have gained from carrying out this project.

Thank you for reading this document.

6. References

1. Socket IO documentation Server in socketio/docs
<https://socket.io/docs/v4/server-installation/>
2. Socket IO documentation Client in socketio/docs
<https://socket.io/docs/v4/client-installation/>
3. React documentation in react.dev
<https://react.dev/learn>
4. Amplify Authentication set up in docs.amplify.aws
<https://docs.amplify.aws/lib/auth/getting-started/q/platform/js/>
5. Building Lambda Functions with Python in docs.aws.amazon
<https://docs.aws.amazon.com/lambda/latest/dg/lambda-python.html>
6. Instances and AMI's in docs.aws.amazon
<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instances-and-amis.html>
7. Blum-Blum-Shub in Wikipedia
https://en.wikipedia.org/wiki/Blum_Blum_Shub
8. Xorshift* in Wikipedia
https://en.wikipedia.org/wiki/Xorshift#xorshift*
9. Linear Congruential Generator in Wikipedia
https://en.wikipedia.org/wiki/Linear_congruential_generator
10. Managing State in react.dev
<https://react.dev/learn/managing-state>