**Ask Ouija**

Ask Ouija is a social game app where people work together to answer questions. In the app, a user will be presented with a list of questions to choose from, or will be able to type in their own question. The other users will then work together to answer that question. These players will be randomly selected to provide a single letter of the answer. Once the letter is provided, another player will be randomly selected to provide a second letter to the answer, and so on, until a word or phrase is spelled out. Instead of providing a letter, a player can choose to “finalize” the answer, meaning they believe the question has been sufficiently answered and it’s time for another player to ask a new question.

Similar to TrashTalk, Ask Ouija will use randomly generated 4 character codes to establish player groups. A player can choose to create a new group or join an existing group. The app will make use of four Firestore collections:

1. ao-questions  
   This collection will be used to store a set of prewritten questions. Each round, a set of questions will be randomly selected from this collection and presented to the “Question Asker.” The Question Asker will then select one of these questions or write their own. The structure for the documents is as follows:  
   ao-questions: {  
    documentID: {  
    questionText: “What should I buy with my stimulus money?”,  
    }  
   }
2. ao-avatars  
   This collection will be used to store a set of avatars. When a user joins a group, an avatar will be assigned to that user at random. The avatars will be stored as Base64 encoded strings. The structure for the documents is as follows:  
   ao-avatars: {  
    documentID: {  
    avatarText: Base64 encoded string,  
    }  
   }
3. ao-games  
   This collection will store the core data that is crucial to the functioning of the game, such as player information, the game code, the question being asked, the answer as constructed so far, who asked the question, and who is currently answering. The structure for its documents is as follows:  
   ao-games: {  
    documentID: {  
    gameCode: 4 digit randomly generated game code,  
    owner: uid for the player who created the game,  
    players: [  
    {  
    avatar: Base64 encoded string,  
    displayName: the player’s name,  
    uid: the player’s unique identifier,  
    },  
    ],  
    question: The question being asked,  
    questionAsker: {  
    avatar: Base64 encoded string,  
    displayName: the name of the player asking the question,  
    uid: the unique identifier of the player,  
    },  
    answeringSpirit: {  
    avatar: Base64 encoded string,  
    displayName: the name of the player answering the question,  
    uid: the player’s unique identifier,  
    },  
    answer: The answer as constructed so far,  
    status: The current status of the game,  
    }  
   }
4. ao-answered-questions  
   When a round is completed, the answer, question, and some basic game information is written to this collection. While this collection isn’t used for any critical game functions, it can be used as a log of sorts to help you find any especially funny or clever question and answer combinations. You could also use the data stored in this collection to see if players are writing their own custom questions, and then steal those questions and add them as new prewritten questions in your ao-questions collection.

The ao-questions and ao-avatars collections will require you to create the collections and associated documents using the Cloud Firestore console. We will do this in a future step. The other two collections will be created and maintained by the app code.

Ask Ouija does not require an account to play and so we will not be making users log in or register. Instead, we will make use of Firebase’s Anonymous Authentication option to automatically create a basic user object for each person who opens the app.

**Configuring Firebase**

Let’s start by configuring Firebase to accept Anonymous authentication. Log in to your Firebase console and select the same Firebase project you used for the EasyToDos and TrashTalk apps.

1. Click the Authentication link in the left menu.
2. Click the Sign-in method tab.
3. Click the Anonymous row in the Sign-in providers tab
4. Click Enable if it’s not already enabled and then click Save.
5. Click the Cloud Firestore link in the left menu
6. Click the Rules tab
7. Add the following rules:  
   rules\_version = ‘2’;  
   service cloud.firestore {  
    match /databases/{database}/documents {  
    match /ao-games/{game} {  
    allow read, write: if request.auth != null;  
    }  
    match /ao-avatars/{avatar} {  
    allow read: if request.auth != null;  
    }  
    match /ao-questions/{question} {  
    allow read: if request.auth != null;  
    }  
    match /ao-answered-questions/{question} {  
    allow read, write: if request.auth != null;  
    }  
    }  
   }
8. Because we’ve enabled Anonymous authentication, anyone connecting to our Firestore through our app will automatically be authenticated. Thus, request.auth will not be null as long as the user is connected via our app. We need our players to be able to read and write to documents in the ao-games collection, read from the ao-avatars and ao-questions collections, and read and write to the ao-answered-questions collection.
9. Publish the rules.
10. Click the Data tab.
11. In the left most Data pane that lists your various collections, click Start collection
12. Enter ao-questions as the Collection ID and click Next.
13. In the Document ID field, click Auto ID.
14. Type questionText in the Field textbox.
15. Type in a question in the Value textbox. For example, What should I buy with my stimulus money?
16. Leave the Type as string and click Save.
17. You should see your new ao-questions collection show up in your list of collections. Click on ao-questions and you should see your first document show up in the documents pane.
18. With ao-questions selected in the collections pane, click Add document in the documents pane and repeat steps 13-16, making sure questionText is entered in the Field textbox exactly as it is written here, but typing a new question in the Value textbox. . Repeat this process until you have a decent number of question documents in the ao-questions collection (**Make sure you at least have 3**).
19. Next, we’ll use an online Image to Base64 converter to create our player avatars. Find a few pictures you’d like to use as player avatars and save them to your computer. Make sure they’re less than 1MB in file size. Then, open a browser window and go to <https://base64.guru/converter/encode/image>. Upload your first picture and click the “Encode image to Base64” button.
20. In your Firebase console, create a new collection called ao-avatars. Click the Auto-ID button. Name the Field avatarText. Copy the enormous text string that was generated by the Base64 encoder and paste it in the Value textbox and click Save.
21. Select the ao-avatars collection and then click Add document. Back in your Base64 encoder window, upload your next photo, copy the generated Base64 text string, and paste it as the Value for the avatarText field. Repeat this process until you have half a dozen or so ao-avatars documents, each with a different picture.
22. In your Firebase console, click Project Overview in the left menu.
23. At the top, click the plus sign to add a new app and select Android.
24. In the Android package name textbox, enter com.askouija
25. In the App nickname textbox, enter Ask Ouija
26. Click Register app and download the config file
27. Place the config file in the /AskOuija/android/app folder. Make sure it’s named google-services.json. If you already had an existing google-services.json file in your Downloads directory, it may have been automatically renamed. Once the file is in its final directory, it needs to be named exactly google-services.json.
28. We’ve now completed the setup necessary to connect Firebase and Firestore to the Ask Ouija app, as well as created some of the base data objects that will be used by our app. Next, we’ll get started coding the app.

**Finishing the App**

You’ll be provided with a shell app which will have a lot of the code already written for you. Download the shell app, unzip its contents, and then open your command prompt or terminal window, navigate to the shell app director, and issue the npm install command. Once all of the node module dependencies have been installed, open the project folder in your code editor. We’ll start by completing App.js.

1. Open the App.js file
2. We need to create a useEffect hook that runs once on app startup. This hook will authenticate the user.
3. Import the firebase and auth modules:  
   import firebase from ‘@react-native-firebase/app’;  
   import auth from ‘@react-native-firebase/auth’;
4. Directly before the return block, add a new useEffect hook:   
   useEffect(() => {  
    const authSubscriber = auth().onAuthStateChanged(onAuthStateChanged);  
     
   auth()  
    .signInAnonymously()  
    .then(() => {  
    console.log(‘User signed in anonymously’);  
    })  
    .catch(err => {  
    let friendlyError = {  
    friendly: “We couldn’t authenticate you with the game service.”,  
    technical: err.toString()  
    };  
    setError(() => { throw friendlyError });  
    });  
     
    return (() => {  
    if (subscriber !== null) {  
    subscriber();  
    authSubscriber();  
    }  
    });  
   }, []);
5. Here, we create a listener on the auth object’s onAuthStateChanged event and tell it to execute a custom method also called onAuthStateChanged when it detects a change. Then, we use the auth object’s signInAnonymously method to sign in the user without forcing them to create an account or log in with specific credentials. If the user authenticates successfully, we log a message to the console. If the process encounters an error, we create an object that contains a friendly error message as well as detailed error data. We then force the app to throw an error using this custom error object. We have a custom component called ErrorBoundary that, through some React Native magic, will detect that an error has been thrown and display an error screen with our custom error message. Finally, we tell the component to execute some clean up code when the App.js component is unmounted.
6. Next, let’s write the function that will allow our app to get a list of players who have joined a game.
7. Import the firestore module:  
   import firestore from ‘@react-native-firebase/firestore’;
8. Create a new function called getPlayers:  
   const getPlayers = gameDocID => {  
    return firestore()  
    .collection(“ao-games”)  
    .doc(gameDocID)  
    .get()  
    .then(doc => {  
    if (doc.exists) {  
    return doc.data().players;  
    } else {  
    return {   
    hasError: true,  
    value: ‘doc-not-found’ };  
    }  
    }  
    })  
    .catch(err => {  
    return { hasError: true, value: err };  
    });  
   };
9. This function queries the ao-games collection for a document with a specific document ID. If it finds the document, it returns the list of players who have joined the game so far. Otherwise, it returns an error object.
10. Next, we’ll create a function that allows our app to update the list of players when a new player joins a game. Create a new function called updatePlayers:  
    const updatePlayers = (gameDocID, playersArray) => {  
     return firestore()  
     .collection(“ao-games”)  
     .doc(gameDocID)  
     .update({  
     players: playersArray,  
     })  
     .then(() => {  
     return { hasError: false, value: null }  
     })  
     .catch(err => {  
     return { hasError: true, value: err }  
     });  
    };
11. This function accepts a game document ID and an array of players as its parameters. Then, it updates the document with the new array of players and then returns an object which indicates if it encountered an error not.
12. Next, we’ll create a function that will assign a random avatar to a user when they join a game. Firestore does not have a native method for selecting a random document, so we’ll need to create a work around to fake our way into grabbing a random document. The workaround is as follows: First, we generate a new document ID. Then, we query for a document in the collection that has an ID of equal or greater value. If we get a document back, then we have our random document. If we don’t get a document back, we execute a second query that looks for a document in the collection with an ID of lesser value than the ID we generated. If we have any documents in our collection at all, this method should ensure we find at least one random document.
13. Create a new function called getAvatar:  
    const getAvatar = () => {  
     let key = firestore().collection(“ao-avatars”).doc().id;  
     return firestore()  
     .collection(“ao-avatars”)  
     .where(firebase.firestore.FieldPath.documentId(), “>=”, key)  
     .limit(1)  
     .get()  
     .then(snapshot => {  
     if (snapshot.size > 0) {  
     let avatar = “”;  
     snapshot.forEach(doc => {  
     avatar = doc.data().avatarText;  
     });  
     return avatar;  
     } else {  
     Return firestore()  
     .collection(“ao-avatars”)  
     .where(firebase.firestore.FieldPath.documentId(), “<”, key)  
     .limit(1)  
     .get()  
     .then(snapshot => {  
     let avatar = “”;  
     snapshot.forEach(doc => {  
     avatar = doc.data().avatarText;  
     });  
     })  
     .catch(err => {  
     return { hasError: true, error: err };  
     });  
     }  
     })  
     .catch(err => {  
     return { hasError: true, error: err };  
     });  
    };
14. Now, we’ll put together these three functions to create a process by which when a new player joins a game, a Firestore listener is created on the ao-games document, the player is assigned a random avatar, we conduct a quick check to make sure the player hasn’t already joined the game once before, and if not, we update the list of players to include the player who just joined.
15. Create a new function called joinGame:  
    const joinGame = gameDocID => {  
     setGameID(gameDocID);  
     subscriber = firestore()  
     .collection(“ao-games”)  
     .doc(gameDocID)  
     .onSnapshot(doc => {  
     updateGameData(doc.data());  
     });  
      
     return getAvatar()  
     .then(avatar => {  
     if (!avatar.hasError) {  
     return getPlayers(gameDocID)  
     .then(players => {  
     if (!players.hasError) {  
     let alreadyThere = false;  
     let newPlayers = players;  
     players.forEach(player => {  
     if (player.uid === auth().currentUser.uid) {  
     alreadyThere = true;  
     }  
     });  
     if (!alreadyThere) {  
     newPlayers.push({  
     uid: auth().currentUser.uid,  
     displayName: auth().currentUser.displayName,  
     avatar: avatar,  
     });  
     return updatePlayers(gameDocID, newPlayers)  
     .then(response => {  
     if (!response.hasError) {  
     setCurrentScreen(‘lobby’);  
     } else {  
     let friendlyError = { friendly: “Something has gone terribly wrong.”, technical: response.value !== undefined ? response.value.toString() : ‘’};  
     setError(() => { throw friendlyError });  
     }  
     });  
     } else {  
     setCurrentScreen(‘lobby’);  
     }  
     } else {  
     let friendlyError = { friendly: “Something has gone terribly wrong.”, technical: players.value !== undefined ? players.value : ‘’ };  
     setError(() => { throw friendlyError });  
     }  
     });  
     }  
     });  
    };
16. Wow! That’s a lot. Let’s take a moment to look at what’s happening here. First, we set the gameID state variable to the value of the gameDocID parameter being passed to this function. The gameID state variable will be passed as a prop to other components so each component can know which game document we’re working with. Then, we create a Firestore subscription so that any time the game document changes, those changes are immediately pushed to our players. Next, we execute the getAvatar function and make sure we got back an avatar and not an error. If there was no error, then we execute the getPlayers function. Again, we check to make sure there was an error, and if not, we loop through the list of players and check to make sure the current user hasn’t previously joined this game. If they haven’t, we add them to the array of players and execute the updatePlayers function to update the game document with the new list of players. If the user had previously joined the game, we dump them into the game screen.  
      
    We’re chaining together three different Firestore queries and any one of them could return an error, so as an extra layer of user friendliness, we check the return value of each function to make sure there are no errors. If there are errors at any step in the process, then we throw a custom error which will be picked up by the ErrorBoundary component and display an error screen to the user.
17. Next, we’ll focus on completing the piece of the app that will let a user create or join a game. Open /AskOuija/components/greeting/index.js
18. Create a function called generateGameCode:   
    const generateGameCode = () => {  
     const letters = [‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, ‘Q’, ‘R’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, ‘X’, ‘Y’, ‘Z’];  
     let newGameCode = ‘’;  
     for (var x=0; x<4; x++) {  
     newGameCode+= letters[Math.floor(Math.random() \* 26)];  
     }  
     return checkGameCode(newGameCode);  
    };
19. Create the function that will check to see if the generated game code is already in use:  
    const checkGameCode = newGameCode => {  
     return firestore()  
     .collection(“ao-games”)  
     .where(“gameCode”, “==”, newGameCode)  
     .get()  
     .then(results => {  
     if (results.size > 0 {  
     return generateGameCode();  
     } else {  
     return saveGame(newGameCode);  
     }  
     })  
     .catch(err => {  
     return { hasError: true, value: err };  
     });  
    };
20. This function will take a generated game code and look for any documents in the ao-games collection that are already using that game code. If it finds one, it will generate a new code and perform the check again. If it doesn’t find a document already using that game code, it will create a new game document with the generated code.
21. Next, let’s create the saveGame function:  
    const saveGame = newGameCode => {  
     return firestore()  
     .collection(“ao-games”)  
     .add({  
     gameCode: newGameCode,  
     status: “waiting”,  
     players: [],  
     owner: auth().currentUser.uid,  
     answer: “”,  
     question: “”,  
     })  
     .then(gameDoc => {  
     return gameDoc.id;  
     })  
     .catch(err => {   
     return { hasError: true, error: err };  
     });  
    };
22. Now, let’s write the function that will allow a user to join a game with a specific game code:  
    const joinGame = () => {  
     if (gameCode.length === 4) {  
     return firestore()  
     .collection(“ao-games”)  
     .where(“gameCode”, “==”, gameCode.toUpperCase())  
     .get()  
     .then(results => {  
     if (results.size > 0) {  
     let thisGameDocID = null;  
     results.forEach(game => {  
     thisGameDocID = game.id;  
     });  
     setGameDocID(thisGameDocID);  
     setCurrentStep(‘get-name’);  
     } else {  
     setCurrentStep(‘error-game-not-found’);  
     }  
     })  
     .catch(err => {  
     let friendlyError = { friendly: “Something has gone terribly wrong.”, technical: err.toString() };  
     setError(() => { throw friendlyError });  
     });  
     }  
    };
23. This function will query the ao-games collection for a document with a specific gameCode. If it finds the document, it will set the gameDocID state variable to the ID of the document it found and then show the user the screen that asks them to input a name. If it does not find a document, it will show a screen informing the user that it could not find a game with that game code.
24. Next, let’s write the function that will allow a user to create a new game:  
    const createGame = () => {  
     generateGameCode()  
     .then(response => {  
     if (!response.hasError) {  
     setGameDocID(response;  
     setCurrentStep(‘get-name’);  
     } else {  
     let friendlyError = { friendly: “Something has gone terribly wrong.”, technical: response.value !== undefined ? response.value.toString() : ‘’ };  
     setError(() => { throw friendlyError });  
     }  
     })  
     catch(err => {  
     let friendlyError = { friendly: “Something has gone terribly wrong.”, technical: err.toString() };  
     setError(() => { throw friendlyError });  
     });  
    };
25. Finally, when a user enters their name, we need to update their profile with that name. This data is not stored in Firestore, but rather on the auth object. Create a new function called updateName:  
    const updateName = () => {  
     if (userName.length > 0) {  
     return auth().currentUser.updateProfile({  
     displayName: userName,  
     })  
     .then(r => {  
     props.joinGame(gameDocID);  
     })  
     .catch(err => {  
     let friendlyError = { friendly: “Something has gone terribly wrong.”, technical: err.toString() };  
     setError(() => { throw friendlyError });  
     });  
     }  
    };
26. Next, we’ll complete the code for the Lobby component, which is shown when the app is waiting for enough players to join the game. When enough players have joined, the person who created the game will see a Start Game button at the bottom of their screen. When they press that button, the app will randomly pick a person that will ask the question and a person to provide the first letter of the answer. Then, we’ll update the game document with these roles and change the status to “playing.” When the app detects the status has changed from “waiting” to “playing”, it will then show the game screen instead of the lobby.
27. To get started, open /AskOuija/components/lobby/index.js.
28. Create a new function called startGame:  
    const startGame = () => {  
     if ((props.GameData !== undefined) && (props.GameData.owner === props.auth.uid) && (props.GameData.players.length > 2)) {  
     let questionAsker = { displayName: ‘’, uid: ‘’, avatar: ‘’ };  
     let answeringSpirit { displayName: ‘’, uid: ‘’, avatar: ‘’ };  
     while (questionAsker.uid === answeringSpirit.uid) {  
     questionAsker = getRandomPlayer(props.GameData.players);  
     answeringSpirit = getRandomPlayer(props.GameData.players);  
     }  
     return setRoles(questionAsker, answeringSpirit)  
     .then(response => {  
     if (response.hasError) {  
     let friendlyError = { friendly: “Something has gone terribly wrong”, technical: response.value.toString() };  
     setError(() => { throw friendlyError });  
     }  
     });  
     }  
    };
29. Next, we need to write the function that will select a random player from an array of players:  
    const getRandomPlayer = players => {  
     return players[Math.floor(Math.random() \* players.length)];  
    };
30. Finally, let’s create the setRoles function to update our game document with our new player roles:  
    const setRoles = (questionAsker, answeringSpirit) => {  
     return firestore()  
     .collection(“ao-games”)  
     .doc(props.GameID)  
     .update({  
     status: “playing”,  
     questionAsker: questionAsker,  
     answeringSpirit: answeringSpirit,  
     question: ‘’,  
     answer: ‘’,  
     })  
     .then(() => {  
     return { hasError: false, value: null };  
     })  
     .catch(err => {  
     return { hasError: true, value: err };  
     });  
    };
31. Now, let’s work on letting our Question Asker, well… ask a question! Open up /AskOuija/components/question-asker/index.js.
32. We need to create a useEffect that will fire off a series of queries to our Firestore database when our player is chosen as the new Question Asker:  
    useEffect(() => {  
     if ((props.GameData !== undefined) && (props.GameData.questionAsker.uid === props.auth.uid) && (props.GameData.question === “”)) {  
     setSelectedQuestion({ id: -1, text: ‘’ });  
     setCustomQuestionText(‘’);  
     const getQuestions = async() => {  
     let someQuestions = [];  
     while (someQuestions.length < 3) {  
     let alreadyPicked = false;  
     let key = firestore()  
     .collection(“ao-questions”)  
     .doc()  
     .id;  
     let potentialQuestion = “”;  
     let dbQuestion = await firestore()  
     .collection(“ao-questions”)  
    .where(firebase.firestore.FieldPath.documentId(), “>=”, key)  
    .limit(1)  
    .get();  
      
     if (dbQuestion.size > 0) {  
     dbQuestion.forEach(q => {  
     potentialQuestion = q.data().questionText;  
     });  
     } else {  
     dbQuestion = await firestore()  
     .collection(“ao-questions”)  
     .where(firebase.firestore.FieldPath.documentId(), “<”, key)  
     .limit(1)  
     .get();  
     dbQuestion.forEach(q => {  
     potentialQuestion = q.data().questionText;  
     });  
     }  
     someQuestions.forEach(question => {  
     if (question === potentialQuestion) {  
     alreadyPicked = true;  
     }  
     });  
     if (!alreadyPicked) {  
     someQuestions.push(potentialQuestion);  
     }  
     }  
     setQuestions(someQuestions);  
     }  
     getQuestions();  
     }  
    }, [props.GameData]);
33. Here, we use our hack to workaround the fact that Firestore doesn’t have a native method for retrieving a random document. We use a loop that will continue to execute queries against our Firestore collection until we’ve picked 3 random questions. For each question we get back from Firestore, we do a check against the questions we already have to make sure we haven’t accidentally selected the same question multiple times. If we haven’t already selected that question, it’s added to the list of questions and the loop continues until we’ve selected all 3 questions.
34. Next, once the player has selected their question, we need to update the game document with the question. Create a new function called askQuestion:  
    const askQuestion = () => {  
     let questionText = ‘’;  
     if (customQuestionText.length > 0) {  
     questionText = customQuestionText;  
     } else if (selectedQuestion.text.length > 0) {  
     questionText = selectedQuestion.text;  
     }  
      
     if (questionText.length > 0) {  
     firestore()  
     .collection(“ao-games”)  
     .doc(props.GameID)  
     .update({  
     question: questionText,  
     })  
     .catch(err => {  
     let friendlyError = { friendly: “Something has gone horribly wrong.”, technical: err.toString() };  
     setError(() => { throw friendlyError });  
     });  
     }  
    };
35. Finally, let’s allow our Answering Spirits the ability to provide letters for the answer. Open up /AskOuija/components/answering-spirit/index.js.
36. When a player selects a letter, we need to add it to the answer so far and then find a new player to provide the next letter. Create a new function called addLetter:  
    const addLetter = () => {  
     return firestore()  
     .collection(“ao-games”)  
     .doc(props.GameID)  
     .update({  
     answer: props.GameData.answer + selectedLetter,  
     })  
     .then(response => {  
     let newAnsweringSpirit = getRandomPlayer();  
     while((newAnsweringSpirit.uid === props.GameData.answeringSpirit.uid) || (newAnsweringSpirit.uid === props.GameData.questionAsker.uid)) {  
     newAnsweringSpirit = getRandomPlayer();  
     }  
     updateAnsweringSpirit(newAnsweringSpirit);  
     })  
     .catch(err => {  
     let friendlyError = { friendly: “Something has gone horribly wrong.”, technical: err.toString() };  
     setError(() => { throw friendlyError });  
     });  
    };
37. Next, let’s create the function that will update the game document with the new Answering Spirit:  
    const updateAnsweringSpirit = newAnsweringSpirit => {  
     return firestore()  
     .collection(“ao-games”)  
     .doc(props.GameID)  
     .update({  
     answeringSpirit: newAnsweringSpirit,  
     })  
     .catch(err => {  
     let friendlyError = { friendly: “Something has gone horribly wrong.”, technical: err.toString() };  
     setError(() => { throw friendlyError });  
     });  
    };
38. If the player chooses to finalize the answer instead of adding a new letter, we need to update our game document with a new Question Asker and Answering Spirit as well as reset the current question and answer text. Create a new function called finalizeAnswer:  
    const finalizeAnswer = () => {  
     let newQuestionAsker = getRandomPlayer();  
     let newAnsweringSpirit = getRandomPlayer();  
     while (newQuestionAsker.uid === newAnsweringSpirit.uid) {  
     newQuestionAsker = getRandomPlayer();  
     }  
     return firestore()  
     .collection(“ao-games”)  
     .doc(props.GameID)  
     .update({  
     questionAsker: newQuestionAsker,  
     answeringSpirit: newAnsweringSpirit,  
     question: ‘’,  
     answer: ‘’,  
     })  
     .catch(err => {  
     let friendlyError = { friendly: “Something has gone horribly wrong”, technical: err.toString() };  
     setError(() => { throw friendlyError });  
     });  
    };
39. There you have it! You’ve completed the Ask Ouija app. What’s shown on each player’s screen is determined by the ao-games document, specifically the questionAsker and answeringSpirit objects in conjunction with the status field. Using these three fields together, we determine if a player should see a list of questions, the answer board, or if they should just be observing what everyone else is doing.