TrashTalk

TrashTalk is a simple chat app that allows a user to create a private room and share a unique room code with their friends, who can then use that code to join the chat room. When the app starts, it will use the device’s geolocation features to record the user’s current position. Once other users have joined the chat room, the app will then calculate the distance in miles (or feet, if the distance is less than one mile) between the user and their friends in the chat room. When a user creates a new account, they will be asked for their name. The name they provide is then sent to a Wikimedia API to pull back a random image that contains the provided name. That image is then set as the user’s avatar.

TrashTalk consists of 8 screens. You will be provided with a partially completed project. The Sign Up and Sign In flows are completed for you. You will be focused on adding the Geolocation functionality and the Firestore functionality that will allow the chat features to work as expected.

For this project, you can use your existing Firebase project that you set up for the Simple ToDos app. This will decrease the amount of time you need to spend setting up Firebase. We will create a new app in Firebase which will allow us to connect TrashTalk to your Firebase project. The app will use a new set of collections and documents so it will not interfere with your Simple ToDos data.

As mentioned, the app consists of eight screens:

1. Welcome/landing
2. New account registration
3. Login
4. Registration/Login error screen
5. Profile Setup - What’s your name?
6. Account Creation Success - Avatar review
7. Create/Join a chat room
8. Chat room

To test this app, you will need to run multiple Android emulators at once. The process for starting the app on your emulators will change slightly from how we’ve done it previously. For this project, you will need to issue commands to start the emulators and then, once they are started, issue the command to build and run the app. If both emulators are started correctly, the app will automatically build and run on each emulator simultaneously. We will go into more details on how to achieve this later.

Prerequisites:

1. You should already have a React Native development environment set up on your local machine. For more information on how to set up your own React Native development environment, check out <https://reactnative.dev/docs/environment-setup>. I recommend using React Native CLI Quickstart. The rest of this material will be written assuming you used these instructions. Expo is nice to get started quickly, but adds more overhead in the long-term, so at this time I recommend against using it.
2. You’ll need a free Google account. Use that account to set up a free Firebase account at <https://firebase.google.com>. Once your Firebase account is set up, you can view your Firebase console at <https://console.firebase.google.com> Note that there are paid tiers of Firebase, but the free tier will be sufficient for what we’re doing.
3. You will need to have multiple Android emulators installed and ready to go.
4. You will need to download the TrashTalk app provided to you.

Let’s get started!

**Setting up a Second Android Virtual Device (AVD)**

1. Start Android Studio
2. Click the Configure dropdown and then AVD Manager
3. Click the “Create Virtual Device…” button at the bottom of the screen
4. Choose a Pixel device, but make it different from the virtual device you already have installed
5. Click Next
6. Click Download next to the “Q” System Image and then click Next
7. Finish the AVD setup
8. Open a command prompt or Terminal window and execute the following command:  
   emulator -list-avds
   1. If this command doesn’t work, you may need to cd your way into the Android/sdk/tools folder on your computer. For example, on Mac:  
      cd Library/Android/sdk/tools
   2. You’ll need to find the installation directory for your Android Studio so you know what your directory structure is
   3. If you’re in the right directory and the command still won’t execute, try:  
      ./emulator -list-avds
   4. You should see a list of the installed AVDs on your machine. For example, mine shows:  
      Pixel\_3\_API\_28  
      Pixel\_3a\_API\_29
9. If you see multiple AVDs show up in the step above, you’ve successfully installed multiple Android AVDs and can proceed to the next section.

**Setting up Firebase**

1. Log in to your Firebase console and go to the project you set up for the Simple ToDos app.
2. At the top of the screen, click the “Add app” button
3. Select Android
4. On the Register App screen, enter com.trashtalk in the Android package name field if you downloaded the provided project. You can double check this value by opening the android/app/build.gradle file and searching for the applicationId.
5. Enter TrashTalk in the App nickname field and skip the Debug signing certificate field this time. Click the “Register app” button
6. Download the Firebase config file and place it in the android/app folder of your project.
7. Skip the “Add Firebase SDK” step and finish setup.
8. Click the Cloud Firestore link in the left menu and then click the Rules tab
9. We’re going to add a new rule to allow authenticated users to read and write to a new collection called chats. Under the match rule for your tasks collection, add a new rule:  
   match /chats/{document=\*\*} {  
    allow read, write: if request.auth != null;  
   }
10. Your entire Rules document should now read as follows:  
    rules\_version = ‘2’;  
    service cloud.firestore {  
     match /databases/{database}/documents {  
     match /tasks/{document=\*\*} {  
     allow read, create: if request.auth != null;  
     allow read, update, delete: if resource.data.uid == request.auth.uid;  
     }  
     match /chats/{document=\*\*} {  
     allow read, write: if request.auth != null;  
     }  
     }  
    }
11. Save your changes
12. For this project, we’ve already included the @react-native-firebase package and enabled MultiDex support for you to speed up development time. We’ve also enabled the Google Services in the Gradle build files. Just remember, if you were building your own project, you would need to perform these steps to complete the connection to Firebase.

**Setting up the React Native project**

1. Open a command prompt or Terminal window and use the cd command to change to the TrashTalk directory.
2. Run npm install to download the Node dependencies.

**Getting the user’s current location**

1. We will be using a node module called @react-native-community/geolocation to tap into the mobile device’s native geolocation capabilities. Open a command prompt or Terminal window and use the cd command to navigate to the TrashTalk project folder.
2. Once in the TrashTalk folder, issue the command to install the Geolocation module:  
   npm install @react-native-community/geolocation --save
3. We will need to request permission to access the device’s location. Open up the TrashTalk/android/app/src/main/AndroidManifext.xml file and add a new <uses-permission> tag under the existing tag:  
   <uses-permission android:name=”android.permission.ACCESS\_FINE\_LOCATION />
4. Open TrashTalk/App.js and import the Geolocation package:  
   import Geolocation from ‘@react-native-community/geolocation’;
5. Add a new state variable that will be used to store the user’s location:  
   const [currentLocation, setCurrentLocation] = useState();
6. Create a new function called getLocation that will tap into the Geolocation package and return the user’s current location:  
   const getLocation = () => {  
    Geolocation.getCurrentPosition(position => {  
    setCurrentLocation(position);  
    console.log(position);  
    },  
    error => console.error(JSON.stringify(error)),  
    {enableHighAccuracy: true, timeout: 20000, maximumAge: 1000},  
    );  
   };
7. This function will execute the getCurrentPosition method of the Geolocation module, returning an object we’re calling position and which we are storing in our currentLocation state variable. We’re also printing the value of the position object to the console so you can see what kind of data is being returned. In the case of an error, the error message will be printed to the console as well. Finally, we pass a few configuration options to use GPS instead of WiFi location, set the maximum amount of time the device is allowed to take in order to return a position, and the maximum age allowed for the device to use a cached value instead of returning a fresh value.
8. Adding the permissions to the manifest file is just one piece of the permissions puzzle. Next, we need to check to make sure the user has agreed to allow us to access the device’s location. If not, we need to request those permissions. We’ll achieve this using a useEffect hook. When the app loads, it will check to make sure it has permission to access the device’s location. If it doesn’t, it will ask the user to grant those permissions. Once the permissions have been granted, we will execute the getLocation function to grab the user’s current location. Import the PermissionsAndroid and Platform packages from the react-native module:  
   import {  
    SafeAreaView,  
    StyleSheet,  
    View,  
    StatusBar,  
    PermissionsAndroid,  
    Platform  
   } from ‘react-native’;
9. Look for the useEffect hook where we’re setting up the authSubscriber listener and create a function that checks for permissions:  
   useEffect(() => {  
    const authSubscriber = auth().onAuthStateChanged(onAuthStateChanged);  
     
    const checkPermissions = async() => {  
    //check for geolocation permissions on Android  
    if (Platform.OS == ‘android’) {  
    let granted = await PermissionsAndroid.check(PermissionsAndroid.PERMISSIONS.ACCESS\_FINE\_LOCATION);  
    if (!granted) {  
    granted = await PermissionsAndroid.request(  
    PermissionsAndroid.PERMISSIONS.ACCESS\_FINE\_LOCATION,  
    {  
    title: “TrashTalk would like to access your location”,  
    message: “TrashTalk needs access to your location so your friends can see how far away you are.”,  
    buttonNeutral: “Ask Me Later”,  
    buttonNegative: “Cancel”,  
    buttonPositive: “OK”  
    }  
    );  
    if (granted === PermissionsAndroid.RESULTS.GRANTED) {  
    getLocation();  
    }  
    } else {  
    getLocation();  
    }  
    }  
    };  
     
    checkPermissions();  
     
    //destroy the authSubscriber when the component unmounts  
    return authSubscriber;  
   }, []);
10. We’re creating an asynchronous function inside of our useEffect hook. First, this function will check to see if the app is running on an android device. If so, we’ll use the PermissionsAndroid package to check whether or not permission has already been granted to access the device’s location. Here, we use the await keyword to stop execution of the rest of the function until PermissionsAndroid has returned a value. If permission has not been granted, we’ll use PermissionsAndroid to ask the user to grant permission to access the user’s fine location. We configure the message we want to be displayed, as well as the types of buttons we want to show and their corresponding labels. Android permissions boxes can have up to three buttons: A positive button that indicates the user is granting permission, a negative button that indicates the user does not grant the permission, and a neutral button that indicates the user is not granting access at this time, but also is not explicitly denying access. The button the user presses will be returned to PermissionsAndroid, which will then return a value to us that is either GRANTED, DENIED, or NEVER\_ASK\_AGAIN. If the return value is GRANTED, then we execute the getLocation function we wrote earlier. Otherwise, we do nothing. If the user had previously granted permission, then we execute the getLocation function immediately on app load. Finally, after we declare the asynchronous function, we immediately execute it so the permissions check will occur.   
      
    Why are we using an asynchronous function? We need to be able to stop execution of the rest of the function while we wait for PermissionsAndroid to do some work and return to us a value. This is made possible through the use of the await keyword. But await can only be used when we explicitly define a function as asynchronous. This is done by using the async keyword when creating a function. We cannot use the async keyword when declaring a useEffect hook, so we create a function inside of the useEffect that uses the async keyword and includes the await in the places we need to pause execution. We then immediately execute that function in the useEffect.  
      
    Another way to achieve similar execution is if PermissionsAndroid utilized a promise structure. In this scenario, our function would remain synchronous-- that is, execution wouldn’t be paused, and we would instead create a callback function that would take the data from PermissionsAndroid when it’s available and execute the next set of instructions. You can find this type of structure in the doRegister function in App.js. You’ll notice that we execute the createUserWithEmailAndPassword method and then use a .then() to create an anonymous function that executes once createUserWithEmailAndPassword is done doing its thing.  
      
    PermissionsAndroid does not use a promise structure, so we use the async and await approach instead.
11. Now, any time the app is loaded or reloaded, we’ll check for permissions and then execute the getLocation function if we have access to the device’s location. That updated location information is then stored in the currentLocation state variable. Now, let’s pass that information on to the chat screen. Scroll down to the return block of App.js. Look for the line that renders the <ChatRoom> component. Modify it so that it sends on the currentLocation state variable as a prop named CurrentLocation:  
    { currentScreen === ‘messages’ ? <ChatRoom Auth={auth} CurrentLocation={currentLocation} updateScreen={updateScreen} RoomCode={roomCode} RoomUsers={roomUsers} Messages={roomMessages} /> : null }
12. Open TrashTalk/components/chat-room/index.js. When new users join a chat room, we want to calculate the distance between our location and their’s. We’ll use a complicated distance calculation I found on Stack Overflow and assume it’s the right way to calculate an accurate distance between two geolocations. Create a new function called calculateDistance:  
    const calculateDistance = (lat1, lon1) => {  
     let earthRadiusMiles = 3958.8;  
     let lat2 = props.CurrentLocation.coords.latitude;  
     let lon2 = props.CurrentLocation.coords.longitude;  
     let distanceLat = degreesToRadians(lat2-lat1);  
     let distanceLon = degreesToRadians(lon2-lon1);  
      
     lat1 = degreesToRadians(lat1);  
     lat2 = degreesToRadians(lat2);  
      
     let a = Math.sin(distanceLat/2) \* Math.sin(distanceLat/2) + Math.sin(distanceLon/2) \* Math.sin(distanceLon/2) \* Math.cos(lat1) \* Math.cos(lat2);  
     let c = 2 \* Math.atan2(Math.sqrt(a), Math.sqrt(1-a));  
      
     let distanceMiles = earthRadiusMiles \* c;  
     if (distanceMiles < 1) {  
     return Math.round(distanceMiles \* 5280) \* “ ft”;  
     } else {  
     return Math.round(distanceMiles) + “ mi”;  
     }  
    };
13. Next, create a new function called degreesToRadians:  
    const degreesToRadians = degrees => {  
     rReturn degrees \* Math.PI / 180;  
    };
14. I’m not going to attempt to explain the mathematical formulas to you. However, the logic of our function is fairly straightforward. We pass in the latitude and longitude of another user as the lat1 and lon1 variables. Then, we use our own latitude and longitude, stored in the CurrentLocation prop, and then use all four of those variables in the complicated formula to calculate a distance between the two locations. If that distance is greater than 1 mile, we return the value as miles. If it’s less than one mile, we convert it to feet instead. We round the values to avoid getting overly precise numbers with a ton of decimal points.
15. We want to use a horizontal ScrollView component to list the users in the chat room. In the return block, above the <FlatList> component, let’s add the ScrollView:  
    <ScrollView  
     horizontal={true}  
     showsHorizontalScrollIndicator={false}  
     style={styles.scrollViewStyles}  
     contentContainerStyle={styles.scrollViewContainer  
    >  
     {getMyself()}  
       
     {props.RoomUsers.map((user, index) => (  
     user.uid !== props.Auth().currentUser.uid ? (  
     <View style={styles.userContainer} key={index}>  
     <Image  
     source={{uri: user.photoURL}}  
     style={styles.participantAvatar}  
     />  
     <Text style={styles.userName}>  
     {user.displayName}  
     </Text>  
     <Text style={styles.distance}>  
     {calculateDistance(user.location.coords.latitude, user.location.coords.longitude)}  
     </Text>  
     </View>  
     ) : (  
     null  
     )  
     ))}  
    </ScrollView>
16. We always want to be the first person shown in the users list, so the first thing we do in our ScrollView is execute a function called getMyself which will render our name and avatar. We will write this function in the next step. Then we use the map function on our array of users that are stored in the RoomUsers prop. The map function allows us to evaluate each item in the array and render a piece of content for each of those items. Here, we make sure that the current user in the array is not us because we don’t want to be listed twice in the user list. If the current user in the array is not us, we render a block of content that includes the user’s avatar, their name, and then this user’s distance from us, by executing the cacluateDistance function we wrote above.
17. We will be the first person displayed in the user list, but there’s no point in calculating a distance from ourselves. By peeling this piece of logic off from the .map function above, we ensure that we are the first person in the user list and that we don’t calculate or render a distance measurement. Let’s create the getMyself function that makes this happen:  
    const getMyself = () => {  
     for (var x=0;x<props.RoomUsers.length;x++) {  
     if (props.RoomUsers[x].uid === props.Auth().currentUser.uid) {  
     return (  
     <View  
     style={{...styles.userContainer,  
     marginLeft: 6  
     }}  
     key={props.RoomUsers[x].uid}  
     >  
     <Image  
     source={{uri props.RoomUsers[x].photoURL}}  
     style={styles.participantAvatar}  
     />  
     <Text style={styles.userName}  
     {props.RoomUsers[x].displayName}  
     </Text>  
     <Text style={{...styles.distance,  
     color: “#434187”  
     }}  
     >  
     {“N/A”}  
     </Text>  
     </View>  
     )  
     }  
     }  
    }
18. This function will loop through all the users in the props.RoomUsers array and check each user’s uid against our own. If the uids match, we’ll return a View component that includes our user avatar, name, and a “N/A” in place of a distance. The “N/A” text will be the same color as the background so it isn’t visible, but including it will allow our elements to stay nicely aligned on the screen.
19. With that, we are done with the location piece of the app. We’ve managed Android permissions, we’ve tapped into the device’s geolocation features, we’re getting our user’s current location, and we’re using that location to calculate how far away we are from other users of the app that join our chat room. In the next section, we’ll implement the logic behind creating and joining a chat room.

**TrashTalk’s data structure**

Like Simple ToDos, we’re using a NoSQL Firestore database composed of collections and documents to store the data and we use listeners so all connected users see messages as soon as they’re saved to the database. Unlike Simple ToDos, we will use a slightly more advanced data structure composed of collections, documents, and… subcollections. Subcollections are simply collections that are created on a document. Let’s take a look at our data structure:

1. First, we have a collection called chats that will be used to store all data associated with TrashTalk. Each “chat room” will be a document inside the chats collection. The document itself will not contain any fields, but will instead consist of a 4 letter document ID that we specify. That chat room document will then have two subcollections: a messages subcollection, which will consist of documents that contain data for individual chat messages, and a users subcollection that contains a document that stores data for each user in the chat room. The structure looks like this:  
   chats: {  
    [chat document: {  
    users: [{  
    displayName: the name of the user,  
    location: {  
    coords: {  
    accuracy: some number,  
    altitude: some number,  
    heading: some number,  
    latitude: some number,  
    longitude: some number,  
    speed: some number,  
    },  
    mocked: boolean,  
    timestamp: a timestamp when the location was obtained  
    },  
    photoURL: the photo URL of the user,  
    uid: the unique identifier of the user  
    }],  
    messages: [{  
    author: {  
    displayName: the user’s name who wrote the message,  
    photoURL: the user’s photo who wrote the message,  
    uid: the unique identifier of the user  
    },  
    messageDate: A timestamp of when the message was written,  
    messageText: The text of the message,  
    }],  
    }]  
   }
2. In a relational database, we would need to create multiple tables and manage foreign key relationships to ensure all the data was related appropriately. In this NoSQL format, we store the data where it’s relevant. Subcollections may seem confusing, but they work the same way as normal collections, they’re just attached to a specific document.

**Creating and Joining Chat Rooms**

A key part of our app architecture is the creation of a random, 4 letter chat room code that can be shared with friends. Your friends can then use that code to join your chat room and let the good times roll. In this section, we’ll create the functions that generate our codes, create the associated chat rooms, and let your friends join you.

1. Open up TrashTalk/components/create-or-join/index.js and create the following function:  
   const createRoom = generatedCode => {  
    firestore()  
    .collection(‘chats’)  
    .doc(generatedCode)  
    .set({  
    roomCode: generatedCode.toUpperCase(),  
    })  
    .then(() => {  
    props.joinRoom(generatedCode.toUpperCase());  
    });  
   };
2. This function will take a randomly generated room code and create a chat room document using that code as the document’s unique identifier. Once the document is created, it will execute a function defined in App.js that places the user in the chat room. Because we’re no longer relying on Firestore to create a unique identifier for us, we also need to make sure that the code we generate isn’t already being used. Let’s create a function that will do that:  
   const checkIfRoomExists = generatedCode => {  
    firestore()  
    .collection(‘chats’)  
    .doc(generatedCode)  
    .get()  
    .then(doc => {  
    If (doc.exists) {  
    generateCode();  
    } else {  
    createRoom(generatedCode);  
    }  
    });  
   };
3. This function will take a generated chat room code and attempt to get a document from Firestore that has that code as it’s document id. If it succeeds and a document exists, it means our generated code is already being used and we need to generate a new one. If it fails and no document is returned, it means that code is not being used and we’re free to create a new document with that id.
4. Next, we need to write the function that will actually randomly generate our code. Create a new function:  
   const generateCode = () => {  
    let generatedCode = ‘’;  
    while (generatedCode.length < 4) {  
    let randomNumber = Math.round(0 + Math.random() \* ((letters.length -1) - 0));  
    generatedCode+= letters[randomNumber]  
    }  
    checkIfRoomExists(generatedCode);  
   }
5. This function grabs 4 random letters from our array of letters and pushes them together to make one string of 4 letters. This is our generatedCode that is then passed to checkIfRoomExists and then ultimately createRoom.
6. Next, we’ll create a function called joinRoom that will be triggered when you or a friend inputs a room code and tries to join that room.  
   const joinRoom = () => {  
    if (enteredCode.length === 4) {  
    firestore()  
    .collection(‘chats’)  
    .doc(enteredCode.toUpperCase())  
    .get()  
    .then(doc => {  
    if (doc.exists) {  
    Keyboard.dismiss();  
    props.joinRoom(enteredCode.toUpperCase());  
    } else {  
    Alert.alert(  
    “That room doesn’t exist”,  
    “We couldn’t find a room with that room code. Are you sure you entered it correctly?”,  
    [  
    {  
    text: “OK”,  
    onPress: () => {},  
    }  
    ],  
    { cancelable: false }  
    }  
    });  
    }  
   }
7. As a user types a room code in the textbox, we store their input in a state variable called enteredCode. When they tap the Join button, we attempt to get a document from Firestore with a document id that equals the room code entered by the user. If we’re successful and the document exists, we hide the keyboard and then execute the joinRoom function from App.js to actually place the user into the chat room. If the document does not exist, we use the React Native Alert component to display a native alert box on the user’s device telling them we couldn’t find that chat room. The Alert component takes several properties: A title, the message, an array of buttons (in this case, one button that says OK and that when tapped dismisses the alert), and an option that indicates whether the user can “cancel” out of the alert by tapping an X instead of one of the buttons. Here, we’ve indicated that the user cannot tap an X and must tap the OK button.
8. Next, let’s create the joinRoom function in App.js. This function will accept a room code, update the users subcollection on the chats document to add a new users subdocument, and update the roomCode state variable. Write the function as follows:  
   const joinRoom = thisRoomCode => {  
    setRoomCode(thisRoomCode);  
    firestore()  
    .collection(‘chats’)  
    .doc(thisRoomCode)  
    .collection(‘users’)  
    .doc(auth().currentUser.uid)  
    .set({  
    displayName: auth().currentUser.displayName,  
    photoURL: auth().currentUser.photoURL,  
    location: currentLocation,  
    uid: auth().currentUser.uid,  
    })  
    .then(() => {  
    setCurrentScreen(‘messages’);  
    });  
   };
9. Notice how we access a subcollection. First, we access the collection (‘chats’), then the specific document we want, and then we use the .collection method again to specify the subcollection we want (‘users’), and finally the specific user subdocument we’re looking for.
10. Note that since we’re using the .set command, we will overwrite the users document if it exists or create a new one if it doesn’t. This saves us some work of determining whether or not a user has already previously joined the room and then trying to prevent a user from showing up twice in the users list on the chat screen. With this method, a user will only ever have one document in the users collection.
11. Let’s say we join a chat room, but then another friend invites us to a different chat room. We exit out of the chat screen, input the new chat room code sent to us by a friend, and join that room instead. To accomplish this, we need to destroy any Firestore listeners that were created on the old chats document and its subcollections and recreate them on the new chats document. We can accomplish this using a new useEffect hook. Note that in the joinRoom function above, we update a state variable called roomCode. When this state variable updates, we want a new useEffect hook to execute that will clean up the old listeners and create the new ones. This is a long block of code, but we’ll examine it in more detail after we write it. Let’s get started:  
    useEffect(() => {  
     let usersSubscriber = null;  
     let messagesSubscriber = null;  
     if ((roomCode !== null) && (roomCode !== ‘’)) {  
     usersSubscriber = firestore()  
     .collection(‘chats’)  
     .doc(roomCode)  
     .collection(‘users’)  
     .onSnapshot(usersSnapshot => {  
     let usersArray = [];  
     if (usersSnapshot !== null) {  
     usersSnapshot.forEach(user => {  
     usersArray.push(user.data());  
     });  
     }  
     setRoomUsers(usersArray);  
     });  
      
     messagesSubscriber = firestore()  
     .collection(‘chats’)  
     .doc(roomCode)  
     .collection(‘messages’)  
     .orderBy(‘messageDate’, ‘desc’)  
     .limit(20)  
     .onSnapshot(messagesSnapshot => {  
     let messagesArray = [];  
     if (messagesSnapshot !== null) {  
     messagesSnapshot.forEach(message => {  
     messagesArray.push({key: message.id, data: message.data()});  
     });  
     }  
     setRoomMessages(messagesArray);  
     });  
     }  
      
     //stop listening for updates when the roomCode changes or when the component unmounts  
     return (() => {  
     if ((usersSubscriber !== undefined) && (usersSubscriber !== null)) {  
     console.log(‘unsubscribing from users’);  
     usersSubscriber();  
     }  
     if ((messagesSubscriber !== undefined) && (messagesSubscriber !== null)) {  
     console.log(‘unsubscribing from messages’);  
     messagesSubscriber();  
     }  
     })  
    }, [roomCode]);
12. We can have as many useEffects as we need. In this app, we now have two: one that creates a listener to the onAuthStateChanged event when the app first loads so we can handle user authentication, and a second that will be executed any time the roomCode state variable is updated.
13. Let’s break down what’s happening in the new useEffect hook. First, we’re declaring a couple of local variables that will be used to hold references to our Firestore subscriptions. Next, we make sure we actually have a roomCode and that we’re not dealing with a blank or null variable. Then, we create the subscription to the users subcollection on the chats document that has an ID of the chat room code we entered. The subscription will return to us an array of users subdocuments any time it detects a change to the users subcollection. We loop through each of those users subdocuments, store each one in a temporary array called usersArray, and then set our roomUsers array to the value of that temporary array. The messagesSubscriber works the same way, but we’re doing a few things differently. First, we’re ordering the messages by messageDate so we can show the most recent messages first, and we’re also limiting the number of messages we show at one time to 20 so we don’t crash the app by attempting to show hundreds or thousands of messages at once. Finally, when we push each message into our temporary messagesArray, we’re pushing an object that contains a key field that is set to the document id of the message, and a data field that will contain an object which has all the actual message data we’re storing in Firestore. The key field is used by the FlatList component on the chat screen to render properly.
14. The final piece of this useEffect hook is the return block. Remember, any code placed in the return block of a useEffect will be executed when a component is “unmounted”, or removed from the screen, so this is typically where we place any “clean up” code. Here, we’re checking to make sure our usersSubscriber and messagesSubscriber variables actually contain references to Firestore listeners, and if so, destroying them so we don’t leave old listeners out there consuming resources and potentially causing bugs.
15. All done! Now, our users will be able to create and join chat rooms. When they enter a chat room, we’ll display the list of people in the chat room along with their distance from our user, and the messages that are being sent back and forth.
16. In the next section, we’ll learn how to use an API to grab a random image that will be used for a user’s avatar.

**Leveraging third party APIs**

Third party APIs are a great way to save time while adding cool new functionality to your app. You can use them to access data hosted outside of your app. In TrashTalk, we use an API provided for free by the WikiMedia foundation. When a user creates a new account, we ask them for their name. We then take that name and send it to the WikiMedia API, asking it to return a list of pictures that contain that person’s name in the picture’s filename. We then choose one of those pictures at random and use it as the user’s profile picture/avatar. If no picture is found, we hard code a default image to use. The WikiMedia API is easy to use because it doesn’t require you to create an account or manage security tokens. As long as you know the endpoint, you can tap into the API.

1. Open up TrashTalk/components/configure-profile/index.js and create a new function called getRandomPhoto:  
   const getRandomPhoto = searchTerm => {  
    let pic = “”;  
    fetch(‘[https://en.wikipedia.org/w/api.php?action=query&list=allimages&aiprop=user|timestamp|url|canonicaltitle&aisort=name&aiprefix=](https://en.wikipedia.org/w/api.php?action=query&list=allimages&aiprop=user%7Ctimestamp%7Curl%7Ccanonicaltitle&aisort=name&aiprefix=)’ + searchTerm + ‘&origin=\*&format=json’)  
    .then(results => results.json())  
    .then(data => {  
    if (data.query.allimages.length <= 0) {  
    getRandomPhoto(userName.substr(0,1));  
    } else {  
    for (var x=0;x<data.query.allimages.length;x++) {  
    pic = data.query.allimages[x].url;  
    if ((pic.indexOf(‘.jpg’) > 0) || (pic.indexOf(‘.jpeg’) > 0) || (pic.indexOf(‘.png’) > 0)) {  
    break;  
    } else {  
    pic = “”;  
    }  
    }  
     
    if (pic === “”) {  
    pic = “https://upload.wikimedia.org/wikipedia/commons/6/69/June\_odd-eyed-cat\_cropped.jpg”;  
    }  
     
    props.Auth().currentUser.updateProfile({  
    displayName: userName,  
    photoURL: pic,  
    });  
    setPhotoURL(pic);  
    }  
    });  
   };
2. We use the fetch keyword to access the API’s endpoint, passing in a lot of parameters and options in the URL along the way. We use the aiprefix=seachTerm format in the URL to tell the API to return a list of images where the filename starts with the searchTerm we’ve specified-- in this case, the name the user entered. When we get our results, we convert them to JSON and then get to work. If we didn’t get any images returned, we execute the function again, but this time the seachTerm is just the first letter of the name entered by the user. Hopefully, this expands the search criteria enough that in the function’s next execution, we’ll get a good list of images.  
     
   Once we have a list of images, we loop through them and look for an image that’s in a format we can use in our app-- in this case, a .jpg/.jpeg or .png. Certain file formats, such as .svg, can cause our app to bomb out because React Native doesn’t have good support for those formats. Once we find a suitable image, we break out of the loop so we don’t waste resources looping through the rest of the results. If we loop through everything and still don’t find a good image, we set the pic variable to a default image of a cat.  
     
   Finally, we use the updateProfile method of the currentUser object on the Firebase Auth module to set the user’s displayName and photoURL. Then, we update the photoURL state variable of the component so the user sees the photo on their screen.

**Test the app by running multiple emulators**

1. Open a new command line or terminal window. We will need to execute various versions of the emulator command like we did when we set up our second AVD at the beginning of this project. Whatever you needed to do to get the command to work then-- be in the Android/sdk/tools directory, use ./emulator instead of emulator, etc., you’ll need to do here as well.
2. Execute the emulator -list-avds command to get the list of AVDs installed on your computer. You should see a list of at least two AVD names.
3. For each emulator, execute the emulator -avd <AVD\_NAME> command. For example:  
   emulator -avd Pixel\_3\_API\_28
4. Once you have your two emulators up and running, open a new command line or terminal window and navigate to the TrashTalk project folder. Run the TrashTalk project like you would normally:  
   npx react-native run-android
5. With any luck, the project will build, install, and launch on **both** running emulators.
6. Now, on one emulator, create a new account and verify that a random image is selected as your avatar. On the second emulator, you can create another new account or log in as one of the accounts you created for the Easy ToDos app. Just note that if you use an account created for Easy ToDos, you will not see a display name or photo for that account in TrashTalk.
7. Try creating a new room on one emulator and joining that room on the other. Send messages back and forth and confirm that a message sent by one user shows up in the app for the other user.
8. On one of the emulators, go back to the main landing (Login/Register/Get Started) screen. On the vertical gray toolbar that is next to your emulator, click the three dots at the bottom to open the emulator settings.
9. Select Location in the left menu if it is not already selected. By default, your location is set to some place in California. Search for a new location and then click the SET LOCATION button. Return to the emulator and refresh the app by hitting R twice on your keyboard or, if that doesn’t work, close the app on the emulator and reopen it.
10. Rejoin the chat room and check the users list at the top to confirm that the distance calculation updated appropriately.

Congratulations! You’ve completed the TrashTalk app, and now you can talk trash with your closest friends using your own homemade app!