EasyToDos

This app is a simple cloud-based task tracker. Users can create an account, login/logout, add tasks that they need to complete, check off those tasks as completed, edit those tasks, and delete those tasks.

The app consists of six screens:

1. Welcome/landing/login
2. New account registration
3. Success/account registration confirmation
4. Task list
5. Task list empty state (for new users, or users who have deleted all of their tasks)
6. Error handler

The app will leverage Google Firebase on the backend to manage account information and store task data. Firebase is a set of cloud services meant to make development faster and easier. Key among these services, and the one we’ll be using most heavily in this project, is Cloud Firestore, a real-time NoSQL database. Firestore allows you to “subscribe” to certain subsets of data. When that data changes, Firebase automatically pushes the updated data to your app.

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.

Let’s get started!

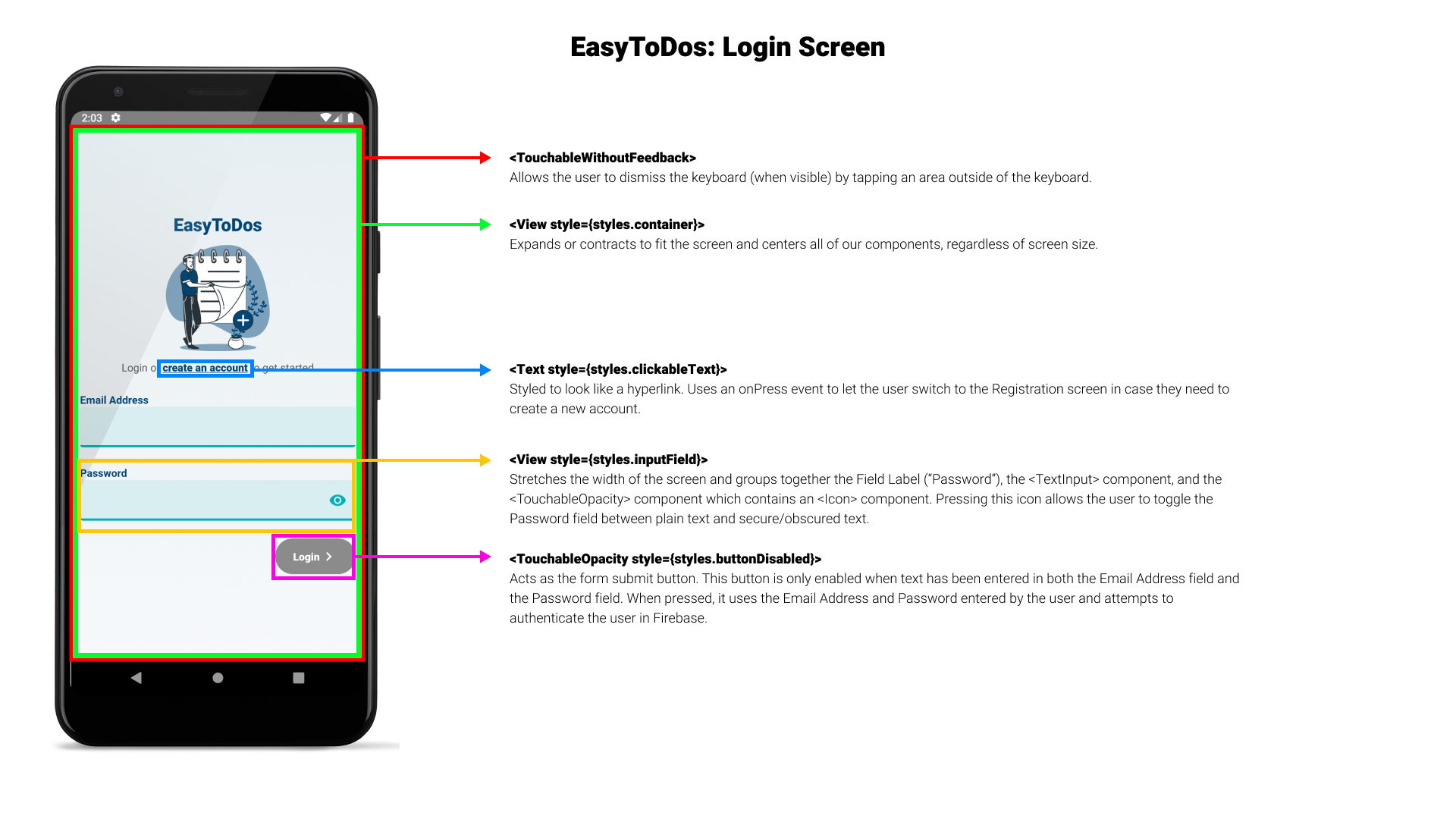
**Setting up Firebase**

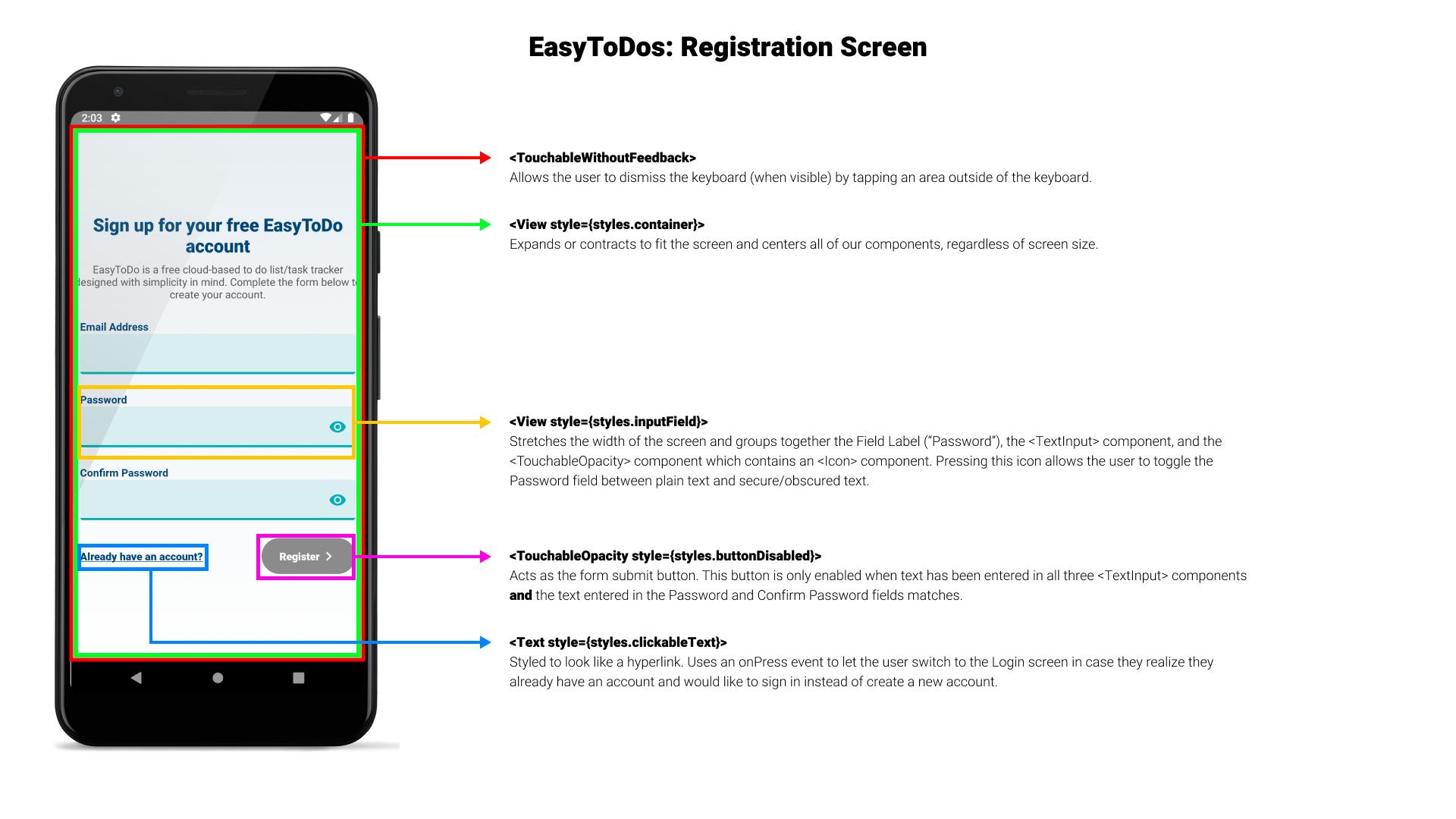
1. Once your Firebase account has been created, login to the Firebase console at <https://console.firebase.google.com>. In my experience, the layout of Firebase can change quite frequently. Because of this, I will not be including screenshots of the Firebase pages and the exact steps, text, and button and link names might be different by the time you follow this guide.
2. Click “Add project”
3. Enter “EasyToDos” as the project name
4. Enable Google Analytics, if asked. While we won’t specifically use this functionality for this project, it may be something you’d like to play around with later.
5. On the “Configure Google Analytics” step, choose “Default Account for Firebase” as your Google Analytics account and “Automatically create a new property” as your Analytics property.
6. Click “Create project”
7. When Firebase has finished creating your project, you should be directed to the Project Overview page. There are several options here. For now, click Cloud Firestore in the left menu under the Develop section.
8. On the Cloud Firestore page, click “Create database”
9. Select “Start in production mode” if it’s not already selected. This will help ensure our data remains secured so that only users who have accounts can access the data.
10. Next, select a region where your Cloud Firestore should be located. Let’s choose a region closest to your physical location where you’ll be doing most of your development work to help minimize any latency as we read and write data. Since I’m currently located in Philadelphia, I chose “us-east1.”
11. Finally, click “Enable” to provision the Cloud Firestore.
12. Once provisioned, you’ll be redirected to your database. You’ll notice there’s not much to see yet.
13. Next, we’ll set up authentication to allow people to register new accounts and login.
14. On the left menu, under Develop, click Authentication.
15. On the Authentication page, click “Get started”
16. You’ll be taken to the “Sign-in method” page where you’ll see a list of Sign-in providers that you can enable. As you look through the list, you’ll notice that Firebase can accept login credentials from various third party providers such as Google, Facebook, Twitter, and Apple. For now, we want our users to be able to register and login using an email address and password. Click the “Email/Password” row and click the top “Enable” switch. Leave the bottom “Email link (passwordless sign-in)” switch disabled.
17. Click the Save button. Firebase is now set up to store user credentials as an email address and password combo.
18. Click the Templates tab at the top, under the “Authentication” section. While we won’t be changing the templates in this project, this is the place where you can change the text of various registration and password reset emails that can be sent to your users.

**Setting up the React Native project**

1. You will be provided a shell project that contains the various screens and layouts to help speed up your development. Your focus for this project will be the data operations: reading and writing data to Firebase.
2. Download the EasyToDos app.  
   npx react-native init EasyToDos
3. Open a command prompt or Terminal window and use the cd command to change to the EasyToDos directory.
4. We will be using a set of packages called React Native Firebase to work with Firebase inside of React Native. We use these packages because Firebase is not natively written to work with React Native. Instead, it has separate Android, iOS, and web APIs. React Native Firebase wraps these APIs into something we can use inside of React Native. This set of packages is open source. More information, including detailed documentation, can be found at <https://www.rnfirebase.io>
5. In your command prompt/terminal window, make sure you’re inside the EasyToDos directory and run the npm install command. This will download all of the Node package dependencies that will be used in this project.
6. Next, issue the following command to install React Native Firebase for your app:  
   npm install --save @react-native-firebase/app
7. Next, we need to register our Android app with our Firebase project. Back in Firebase, click “Project Overview” in the left menu. Here, you should see the option to “Add an app to get started.”
8. Click the Android logo
9. Firebase is now asking us for various information about our Android app. The most important piece of information we’ll need to provide is the Android Package Name. **It is important that the package name we provide exactly matches the package name that was created when we initialized the React Native project.** Mismatches here, even though they may simply be typos, will result in errors later on.
10. Using your favorite IDE or text editor, open your EasyToDos project. For example, if you’re using Atom, click File, Add Project Folder, and navigate to and select EasyToDos.
11. Open the /android/app/src/main/AndroidManifest.xml file in your IDE/text editor. Look for the <manifest xmlns:android="<http://schemas.android.com/apk/res/android>" package="com.easytodos"> tag.
12. Copy the package variable. In this case, com.easytodos. Do not include the quotation marks.
13. Paste this value in the “Android package name” field in the “Add Firebase to your Android app” screen in your Firebase console.
14. In the “App nickname” field, enter EasyToDos.
15. Certain functionality of Firebase requires a debug signing certificate to work correctly while developing and testing. While not exactly necessary for this project, it’s useful to know how to generate such a certificate.
16. In your command line/terminal window, navigate to the android directory of your React Native project and generate a signing report:  
    cd android && ./gradlew signingReport
17. Copy the SHA1 value and paste it into the “Debug signing certificate” field in Firebase Console.
18. Click “Register app”
19. Download the “google-services.json” file and place it inside the /android/app directory of your EasyToDos project.
20. Click “Next.” We don’t need to follow these steps since we’re using React Native Firebase. Click “Next” again and then finally “Continue to console.”
21. Next, we need to enable the “google-services” plugin in our project to allow Firebase to use the credentials we downloaded in the “google-services.json” file
22. Back in your text editor/IDE, open up the /android/build.gradle file
23. You should see something that looks like this:  
    buildscript {  
     ext {  
     buildToolsVersion = “29.0.2”  
     minSdkVersion = 16  
     compileSdkVersion = 29  
     targetSdkVersion = 29  
     }  
     repositories {  
     google()  
     jcenter()  
     }  
     dependencies {  
     classpath(“com.android.tools.build:gradle:3.5.3”)  
     }  
    }
24. In the dependencies section add a new line directly under the existing classpath line:  
    classpath ‘com.google.gms:google-services:4.3.3’
25. Your dependencies section should now read as follows:  
    dependencies {  
     classpath(“com.android.tools.build:gradle:3.5.3”)  
     classpath ‘com.google.gms:google-services:4.3.3’  
    }
26. Next, we need to tell Android to execute this plugin when the app builds. Open the file /android/app/build.gradle and add the following line at the top directly under the first line:  
    apply plugin: “com.google.gms.google-services”
27. We have now set up our Firebase project, our React Native project, and connected the two. The “google-services.json” file contains the information React Native Firebase will need to connect to your Firebase project, and the changes we just made to the Android configuration files will allow Android to access these credentials and make them available to React Native Firebase. Please note that if you’re interested in also connecting Firebase to an iOS app, you’ll need to follow a similar process: create the iOS app in Firebase, download the credentials file, and edit iOS configuration files to read the credentials. Those detailed steps can be found on the React Native Firebase website, <https://rnfirebase.io>
28. Android has a limit where any single app cannot contain more than 64,000 methods. Unfortunately, when we add the @react-native-firebase node modules to our project, we start to exceed that limit. We can get around this limitation by enabling multidex support.
    1. Open the /android/app/build.gradle file. Look for the android section and then the defaultConfig section. Add a new property to the defaultConfig:  
       defaultConfig {  
        //other properties here…  
        multiDexEnabled true  
       }
    2. Look for the dependencies section. Add a new dependency implementation:  
       dependencies {  
        //other implementations here…  
        implementation ‘androidx.multidex:multidex:2.0.1’  
       }
    3. Open your android/app/src/main/java/com/easytodos/MainApplication.java file and make the following changes:
       1. Add a new import at the top after all the other import statements:  
          import androidx.multidex.MultiDexApplication
       2. Edit the public class MainApplication implements React Application { line so that it reads as follows:  
          public class MainApplication extends MultiDexApplication implements ReactApplication {

**Allowing users to Register and Log in**

****

****

The Login screen is the first screen a new user will see when they open the application. Here, they’ll be able to enter their email address and password and login if they already have an account, or they’ll be able to choose to create an account if they do not already have one. Above, you’ll find screenshots of our Login and Registration screens with a high level overview of the various components that make up the screens.

The Login and Registration screens are already completed for you. They will accept input from the user and then attempt to execute the props.doLogin or props.doRegister functions. We need to create these functions in our App.js file so our users can complete the registration and login processes.

1. In your command prompt or terminal window, make sure you’re in the EasyToDos directory and issue the following command:  
   npm install --save @react-native-firebase/auth
2. Open up your App.js file and import the @react-native-firebase/auth module:  
   import auth from ‘@react-native-firebase/auth’;
3. Create a new function called doRegister. Place it under your state variable declarations:  
   const doRegister = (email, password) => {  
    auth()  
    .createUserWithEmailAndPassword(email, password)  
    .then(() => {  
    switchScreen(‘registration-success’);  
    })  
    .catch(error => {  
    if (error.code === ‘auth/email-already-in-use’) {  
    setErrorMessageText(“An account already   
    Exists with that email address. Why not  
    try logging in instead?”);  
    setErrorModalVisible(true);  
    } else if (error.code === ‘auth/invalid-email’) {  
    setErrorMessageText(“The email address you  
    entered is invalid. Make sure it’s   
    formatted correctly. For example,   
    [me@myemail.com](mailto:me@myemail.com)”);  
    setErrorModalVisible(true);  
    }  
    });  
   };
4. This function will be called when the user completes the Registration process. It uses Firebase authentication to create a new account using the email address and password provided by the user. If the user enters an invalid email address, or an email address already being used by someone else, an error message is displayed informing the user. If the registration process is successful, we call the switchScreen function to update our currentScreen state variable, which will then render a screen that informs the user that they have successfully created an account.
5. Next, let’s create the doLogin function. Create this function below your doRegister function:  
   const doLogin = (email, password) => {  
    auth()  
    .signInWithEmailAndPassword(email, password)  
    .then(() => {  
    console.log(“User signed in!”);  
    })  
    .catch(error => {  
    setErrorMessageText(“We couldn’t log you in.   
    Either we couldn’t find an account with that   
    email address or the password you entered was   
    incorrect.”);  
    setErrorModalVisible(true);  
    });  
   };
6. Just like the doRegister function, the doLogin function will take the email address and password entered by the user and use the Firebase auth module to attempt to authenticate the user. If there’s an error, we inform the user. Otherwise, we don’t do anything else. We’ll use the power of React’s useEffect hook to continue the log in process for us.
7. Sometimes, we need to perform certain actions when a component loads or reloads. React’s useEffect hook helps us perform those actions. Code inside of a useEffect hook will be executed once React has finished rerendering the screen. We will use a useEffect hook to create a listener when our app first loads. This listener will listen to the Firebase auth module for any changes to the user’s authentication state. To get started, update the very first import statement in App.js so that it reads as follows:  
   import React, { useState, useEffect } from ‘react’;
8. Next, below your doLogin function, create your useEffect hook:  
   useEffect(() => {  
    const subscriber = auth().onAuthStateChanged(onAuthStateChanged);  
    return subscriber;  
   }, []);
9. Our useEffect hook is an anonymous function that creates a constant named subscriber. This constant holds a reference to the Firebase auth module’s onAuthStateChanged function. We pass to this function our own function, to be defined in the next step, also called onAuthStateChanged. When our app closes (or reloads) we want to destroy any subscriber listeners that may still be active to help prevent memory leaks and creating duplicate listeners that could cause havoc with our app. To do this, we use a return statement. Anything you return from a useEffect hook is considered “clean-up” and will be executed when the hook reruns or the component unmounts (is removed). Finally, you’ll notice we pass an empty array as a second argument of the useEffect hook. This is where the magic for useEffect occurs. An empty array means we only want this piece of code to run one time: when the component loads. We could also pass state variables or props, and the useEffect code will only run when the values of those variables or props changes.
10. Now, let’s create our onAuthStateChanged function:  
    const onAuthStateChanged = user => {  
     setUser(user);  
     if ((user) && (currentScreen === ‘login’)) {  
     setCurrentScreen(‘tasks’);  
     }  
     if (initializing) {  
     setInitializing(false);  
     }  
    };
11. To recap: When our app loads, we create a listener that listens for changes to the Firebase auth module’s auth state. If the user has logged in before, Firebase will remember their details and automatically log them in again, triggering the onAuthStateChanged event from the Firebase auth module. This then executes our own onAuthStateChanged function which sets our user state variable to the value of the user object received from the Firebase auth module’s onAuthStateChanged event. When a user manually logs in, the same process occurs. When a user completes the registration process, they are automatically logged in and so, again, the same process occurs. Once our user state variable has a value, and our initializing state variable has been set to false, we display the appropriate screen: either the Tasks screen if the user has an account and is logged in, or the Login screen if the user is not logged in.
12. Finally, we want to add a function that logs out the user as well. Create the following function:  
    const logoutUser = () => {  
     auth()  
     .signOut()  
     .then(() => setCurrentScreen(‘login’));  
    };

**Structuring Our Data**

Firebase Firestore is a NoSQL database. Instead of tables with rigidly structured and typed columns of data, Firestore instead stores its data as documents in collections. If you need to make comparisons to a more conventional SQL environment, think of a collection as a table and a document as a row in that table. Documents store data as a JavaScript object and you can add and remove fields of data on the fly. This helps us work faster, but can also make us more likely to run into problems because unlike the more traditional table with its well-defined columns, there’s nothing to make sure all of our documents are structured the same or contain the same types of data.

The structure for our EasyToDos app will be fairly straightforward. We will have a collection called tasks. Each document in that collection will describe a task that needs to be completed. Each task document will contain four fields:

1. completed (a boolean value that describes whether the task has been completed or not)
2. date (the date and time the task was added to the list)
3. task (the text of the task)
4. uid the unique identifier of the user who created the task

An example might be:

{  
 tasks:  
 [  
 Document:   
 {  
 completed: false,  
 date: December 7, 2020 at 10:47:07 PM UTC-5  
 task: “Write Chapter 1 of Somehow I Manage”  
 uid: “RkeTOJr8NZT1rjWpITRGRwDDBRRq2”  
 },  
 ]  
}

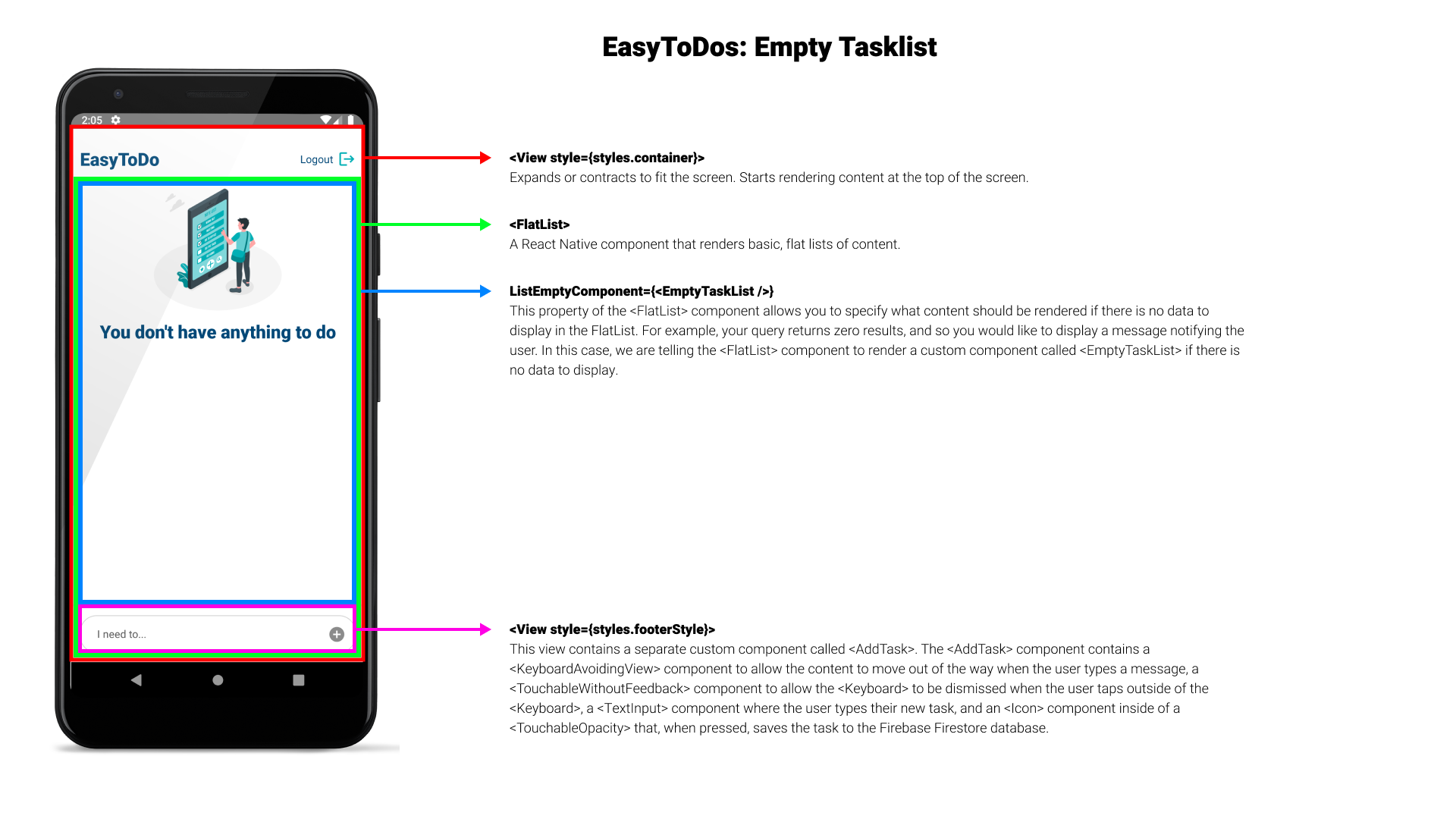
When we attempt to add a document, a collection will automatically be created if it doesn’t already exist.

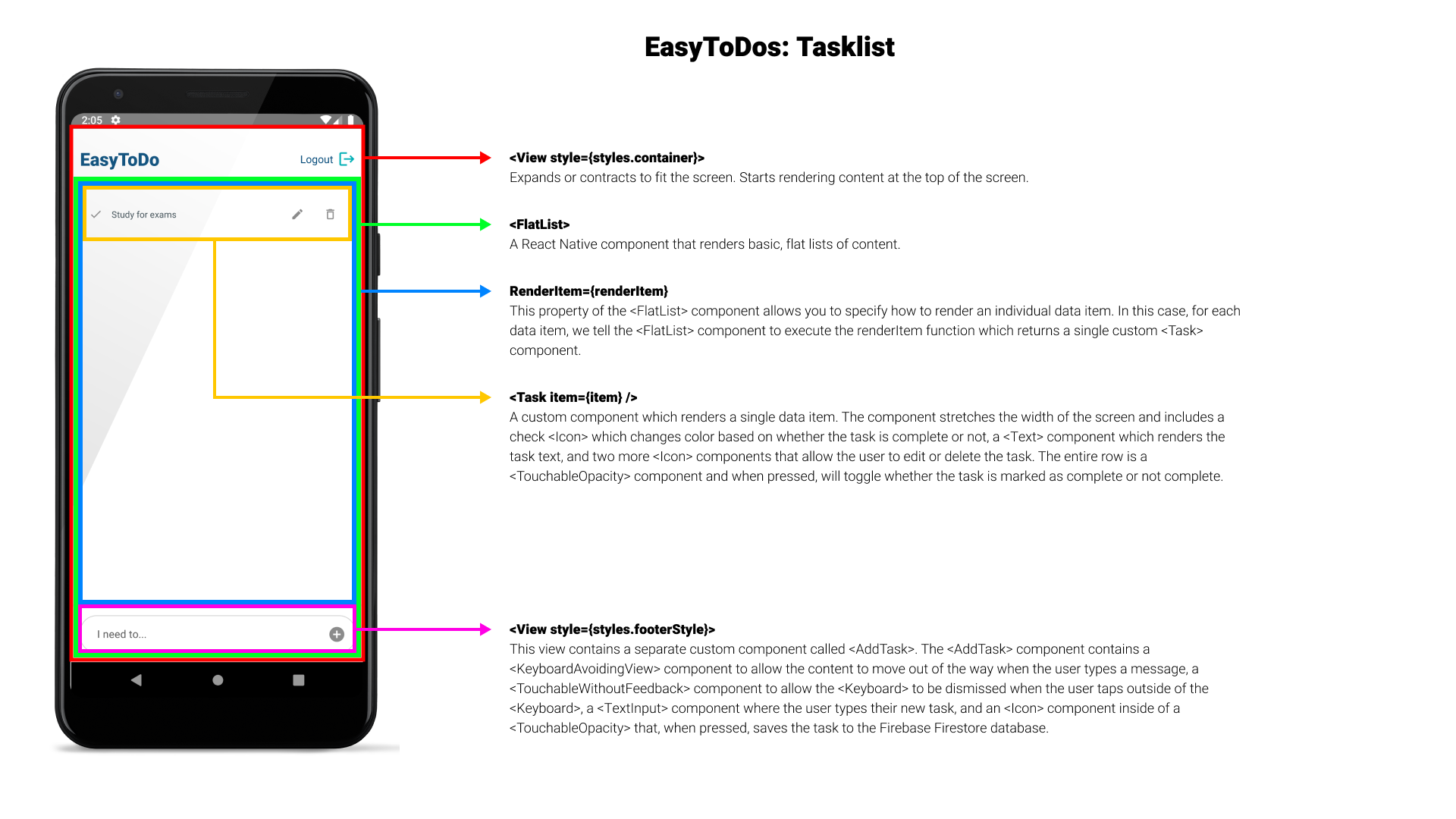
**Securing Our Data**

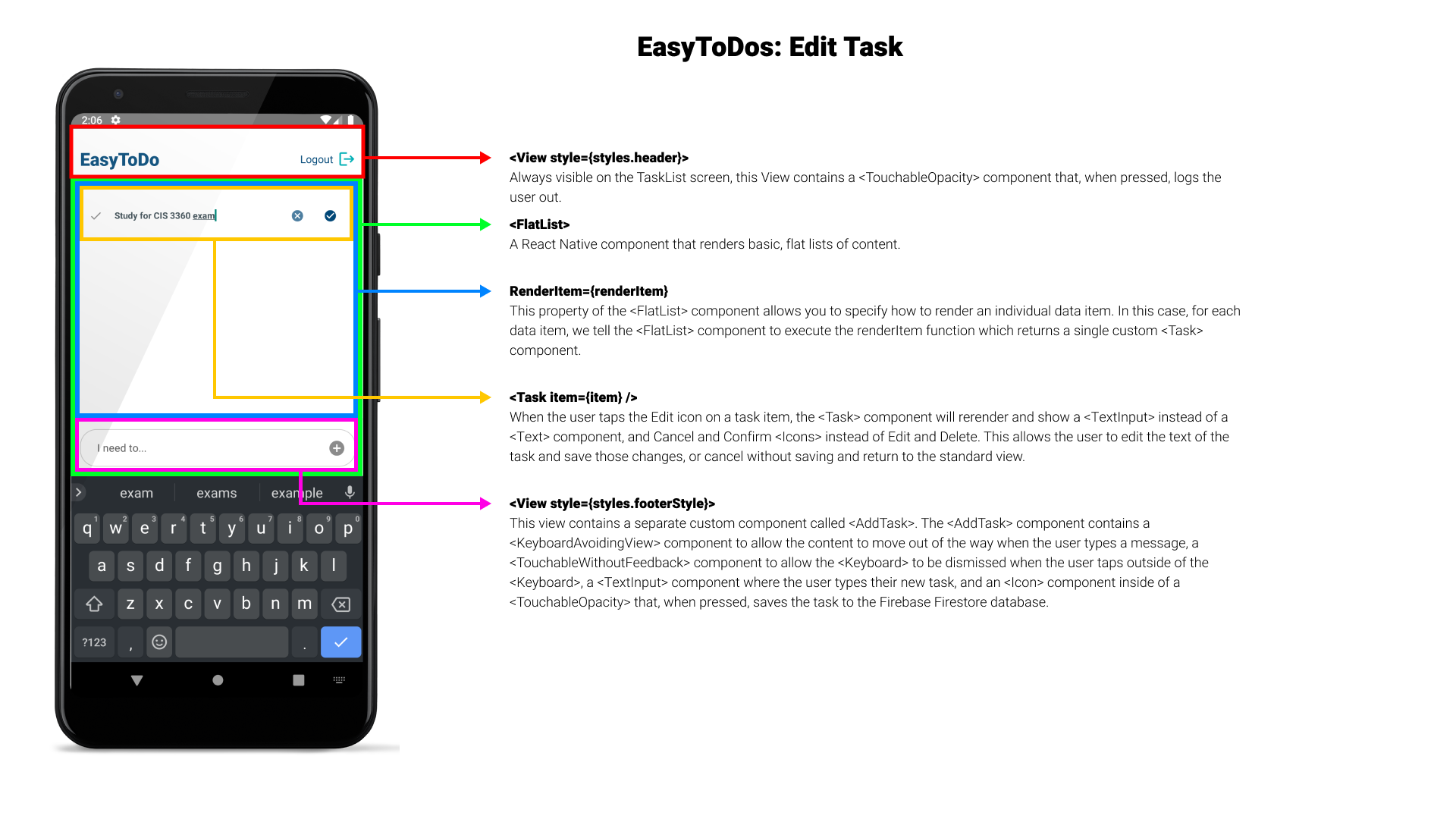
Our app will be used by multiple people and individuals may be tracking personal to dos that they don’t want others to see. Our queries will of course filter on tasks where the uid is equal to the uid of the currently logged in person. However, it’s always good practice to make our data as secure as possible to avoid potential leaks. Let’s provide an additional layer of security:

1. In your Firebase Console, click Cloud Firestore in the left menu if you’re not already in the Cloud Firestore module.
2. Click the Rules tab.
3. Update your rules so they look like the following:  
   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;  
    }  
    }  
   }
4. Save your changes.
5. We’ve locked down our data so that users can only read existing tasks or create new tasks if they’re logged in. We’ve also created a rule so that a user can only read, update, or delete a task when it was created by them. If we mess up our query later and started pulling back more data than we intended, these rules would prevent users from being able to see or modify tasks that don’t belong to them.

**Building the Tasklist**

****

****

****

The tasklist is the heart of the app. It consists of three major parts: a header at the top with the name of the app and a Logout button, the list of tasks in the middle, and a textbox at the bottom that allows users to enter a new task. You have been provided with the basic framework of the Tasklist, including the header row and the new task textbox and button. We’ll build the actual list of tasks together.

1. Open up components/add-task/index.js. Our Add Task button has already been wired up to execute the addTask function when the button is pressed. Now, we just need to write the function! First, let’s install the React Native Firebase Firestore module using npm. In your terminal/command line, make sure you’re in the EasyToDos directory and execute the following command:  
   npm install -- save @react-native-firebase/firestore
2. Now, let’s import the module into components/add-task/index.js:  
   import firestore from ‘@react-native-firebase/firestore’;
3. Unlike relational databases such as SQL Server, NoSQL databases such as Firestore do not require fixed, pre-defined data structures. We can dynamically create collections and documents, as well as document fields, as we need them. Our tasks data structure will look something like the following:  
   tasks: {  
    uid: string containing the logged in user’s id,  
    task: the task’s name/description as provided by the user,  
    date: the timestamp of when the user created the task,  
    completed: boolean flag indicating whether or not the task has been completed,  
   }
4. Create the addTask function:  
   const addTask = () => {  
    firestore()  
    .collection(‘tasks’)  
    .add({  
    uid: props.User.uid,  
    task: taskText,  
    date: firestore.FieldValue.serverTimestamp(),  
    completed: false,  
    })  
    .then(() => {  
    console.log(‘Task added!’);  
    setTaskText(‘I need to…’);  
    });  
   };
5. This function creates a new document in the tasks collection in our Firestore database. Remember when a user signs in to the app, we set the value of the user state variable to the value of the user object we receive from the Firebase auth module. Then, we pass that value as a prop called User to the Tasks component, which in turn passes it as a prop called User to the AddTask component. We then access the uid field in this function to associate our task with a specific user. We also use a Firestore FieldValue to grab the current date and time from the Firestore server and use that as the value for the date field.
6. Let’s hook up our task data to our app. Open up components/tasks/index.js
7. First, let’s import the useState and useEffect hooks into components/tasks/index.js:  
   import React, { useState, useEffect } from ‘react’;
8. Also import firestore:  
   import firestore from ‘@react-native-firebase/firestore’;
9. Next, declare a state variable:  
   const Tasks = (props) => {  
    const [taskData, setTaskData] = useState();  
    //the rest of your component…  
   };
10. Now, set up the useEffect hook so that when the <Tasks> component is loaded, we create a subscription to Firebase for our task data and so that when the <Tasks> component is unloaded, we destroy the subscription:  
    useEffect(() => {  
     const subscriber = firestore()  
     .collection(‘tasks’)  
     .where(‘uid’, props.User.uid)  
     .onSnapshot(tasks => {  
     if (tasks) {  
     let tasksArray = [];  
     tasks.forEach(task => {  
     tasksArray.push({  
     ...task.data(),  
     key: task.id,  
     });  
     });  
     setTaskData(tasksArray);  
     }  
     });  
      
     //stop listening for updates when no longer required  
     return () => subscriber();  
    }, []);
11. The useEffect hook is the way we can cause certain actions to happen when specific pieces of data have updated as well as when certain React lifecycle events occur. Here, the useEffect hook will run when the component is first loaded. Anything in the return block will be executed when the component is destroyed and is typically where cleanup methods are executed. Here, we’re destroying our subscription to Firebase so we don’t waste resources.
12. The onSnapshot method of the firestore() object creates a listener that will allow your app to access changes to documents returned by the query in realtime. This is a powerful feature of Firebase, but sometimes we don’t need realtime changes. In those instances, we can use the .get() method instead. Each time Firebase notifies us of new data, we’re refreshing the data stored in our taskData state variable with the new set of fresh data.
13. Despite being the core functionality of the app, the Tasklist will actually be relatively easy to set up. Let’s add a React Native FlatList component. The project with which you’ve been provided already imports all the necessary components, so we just need to add the component to our return statement. Below your headerRow View and above your footerStyle View, add the FlatList:  
    <FlatList  
     ListEmptyComponent={<EmptyTaskList />}  
     style={{width: “100%”}}  
     renderItem={renderItem}  
     data={taskData}  
     keyExtractor={item => item.key}  
     keyboardShouldPersistTaps=”handled”  
    />
14. Here’s what we’re doing: First, we’re telling the FlatList component to render a custom component named EmptyTaskList if there is no actual data to show. If there is data to show, the array of data will be passed to the function specified in the renderItem property (in this case our function is also called renderItem). This function will loop through the data and render each item. Next, we tell the FlatList component that it can find the data we want to render in our taskData state variable. Remember, this variable gets updated via our Firebase onSnapshot subscription we created in step 10. Next, FlatList needs each data item to have a unique key field, so we use the keyExtractor property to tell the FlatList component it can find the unique key in our item.key field. Finally, we set the keyboardShouldPersistTaps property to “handled”. Because our FlatList will be on the same screen as our AddTask component, which will cause the keyboard to be displayed when a user tries to add a task, we need to make sure the keyboard doesn’t remove the ability for our FlatList to detect taps. If we don’t set this property correctly, our FlatList may not recognize taps and so may not scroll and may not allow us to mark tasks as complete, edit them, delete them, etc.
15. Finally, let’s create the renderItem function which will take care of rendering each data item in the FlatList. Below your useEffect hook, create a function named renderItem:  
    const renderItem = {{ item, index }) => (  
     <Task item={item} />  
    );
16. The final piece of the puzzle is to create our Task component. This component is used by the renderItem function and contains the instructions for how to render a single item. Open up components/task/index.js.
17. Import firestore:  
    import firestore from ‘@react-native-firebase/firestore’;
18. We need to create three functions in order to handle our database operations: one to mark a task as complete (or mark a completed task as incomplete), one to delete a task, and one to save an edited task.
19. Create the completeTask function:  
    const completeTask = () => {  
     firestore()  
     .collection(‘tasks’)  
     .doc(props.item.key)  
     .update({  
     completed: props.item.completed ? false : true,  
     });  
    };
20. Remember that the renderItem function in components/tasks/index.js returns a <Task> item with a property named item with a value of task document from our tasks collection in Firebase. Here, we use that property to access various pieces of data from our document, including the unique document ID which we have stored in the key field. We set the completed field on the database to the opposite of whatever its current value is. This allows us to easily mark an incomplete task as complete and vice versa. That’s all there is to it! Now, when a user taps a task, the related document in our Firebase Firestore database will update accordingly.
21. Next, let’s add the deleteTask function:  
    const deleteTask = () => {  
     firestore()  
     .collection(‘tasks’)  
     .doc(props.item.key)  
     .delete();  
    };
22. Again, because we’re storing the document’s unique identifier in a field called key in our item object, we can use this field to grab the appropriate document and then delete it.
23. Finally, let’s create the saveChanges function:  
    const saveChanges = () => {  
     firestore()  
     .collection(‘tasks’)  
     .doc(props.item.key)  
     .update({  
     task: taskText,  
     })  
     .then(() => {  
     setEditMode(false);  
     setTaskText(‘’);  
     Keyboard.dismiss();  
     });  
    };
24. Simple enough! We tell Firestore which document we want to update, then inside of the update method we create an object and pass in the fields we want to update along with their new values. When Firestore is finished performing the update, we then set our editMode state variable back to false, clear out the taskText state variable, and hide the keyboard so the TaskList is returned to its original read-only state… except now our task has updated text!
25. You’ve now completed the EasyToDos app. Save all your files, launch the app on your emulator, and try creating an account, logging in, logging out, creating a task, updating a task, and deleting a task.