

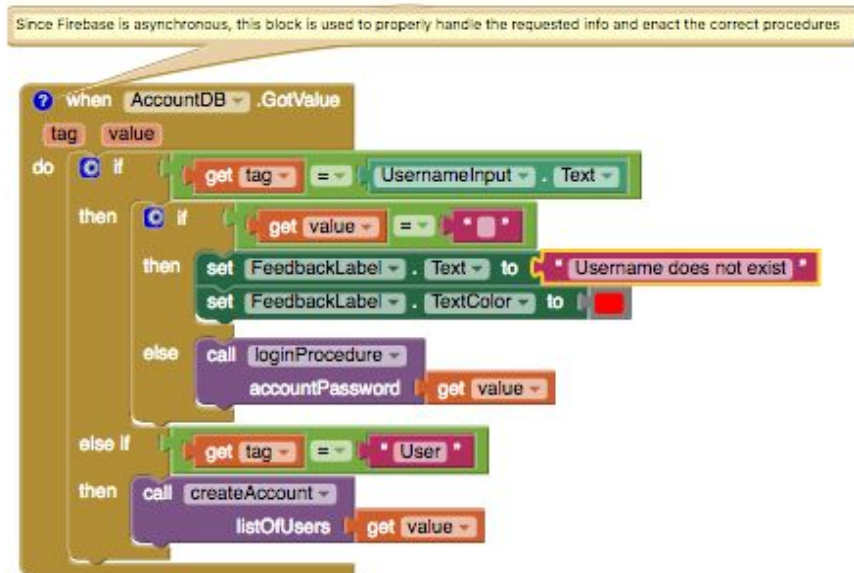
2a. Narration in video.

2b.

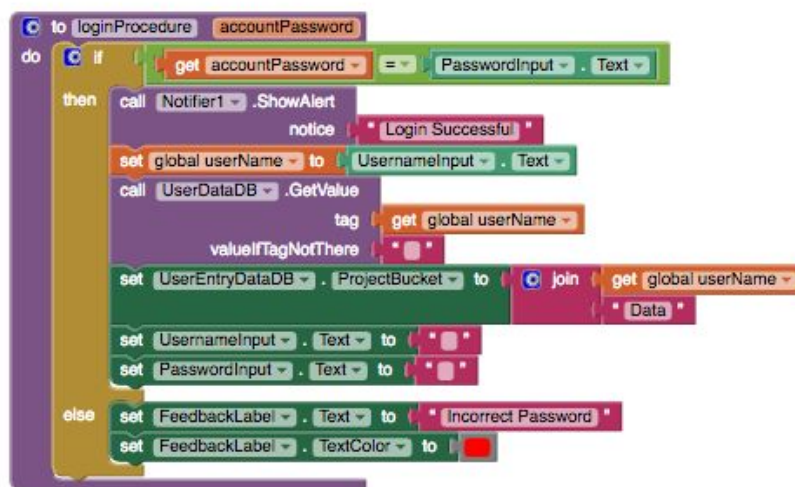
Being unfamiliar with Firebase's structure, I encountered a problem while programming when I tried to include a 3rd Firebase database. Upon the addition of the component and the corresponding coding elements, my app could no longer be packaged or loaded onto a device for testing. My app would always crash while loading. Unable to find a clear syntax error, I resolved the issue by debugging and deleting portions of the code until the app would finally successfully load and then reprogramming the deleted portions of code.

Another difficulty I encountered was transferring variables across screens in order to access the correct user's data. Opening a new screen in App Inventor would clear the values of the variable on the device, which would render them unusable on the next screen. I resolved this independently by assembling the contents of each screen into its own arrangement, and utilizing the `.visible` property of these arrangements to make them appear and disappear, providing the illusion of multiple screens and allowing the accessed variable values to be consistent across all "screens".

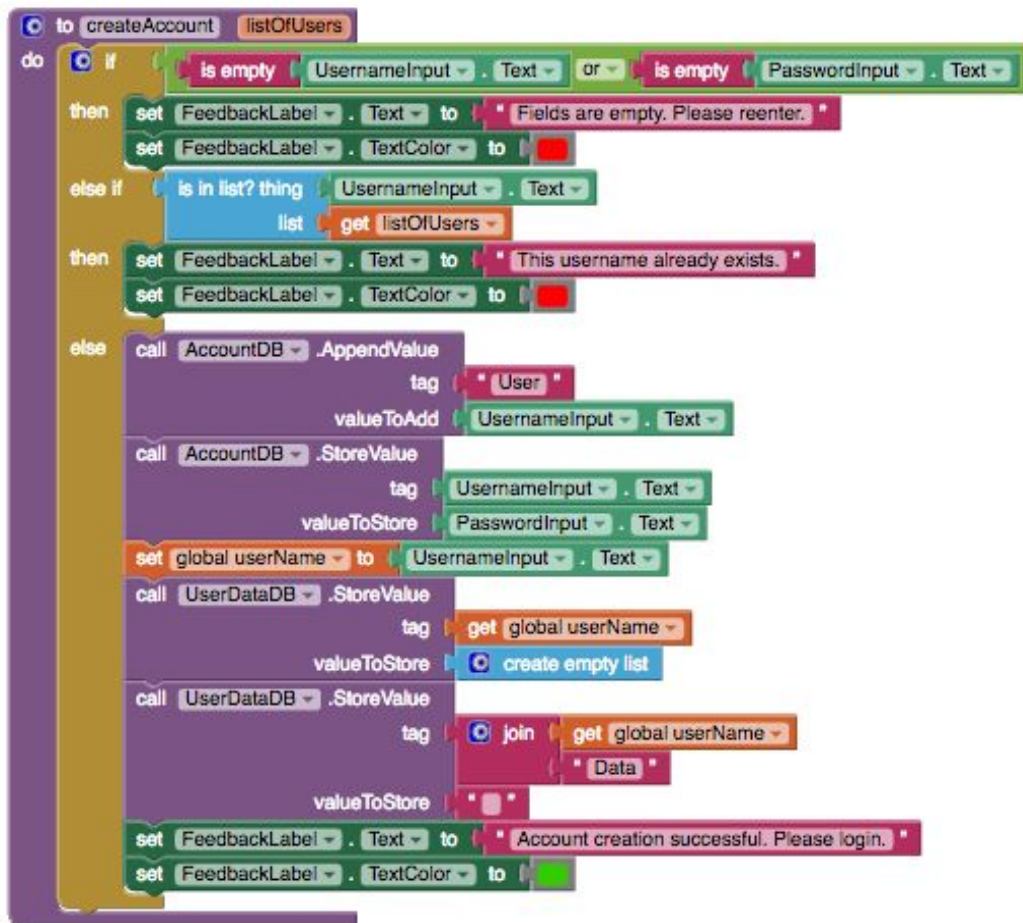
2c.



As my program uses Firebase databases to store user data, `AccountDB.GotValue` is an important algorithm as it handles all data retrieved from the account database such as users and passwords. Because Firebase data requests are handled asynchronously to the program, it is necessary that when data is sent back from Firebase, the algorithm examines the tag and values sent back in order to properly redirect the program to either proceed with a login or create account procedure.



One of the integrated algorithms is the procedure called `loginProcedure` (above). When called, the procedure `loginProcedure` will login in the user and load up the user's diary entries if the correct password is entered. Otherwise, an error message will appear and the user will have to try again.

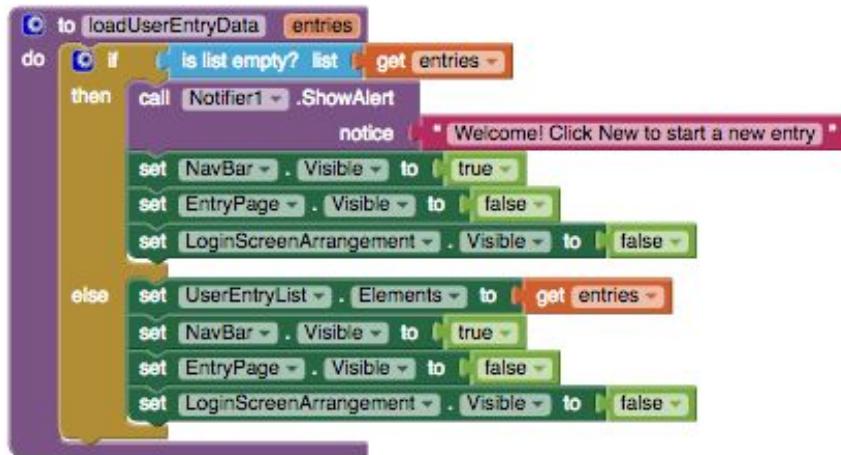


```

to createAccount listOfUsers
do
  if (is empty UsernameInput . Text or is empty PasswordInput . Text)
  then
    set FeedbackLabel . Text to "Fields are empty. Please reenter."
    set FeedbackLabel . TextColor to red
  else if (is in list? thing UsernameInput . Text list get listOfUsers)
  then
    set FeedbackLabel . Text to "This username already exists."
    set FeedbackLabel . TextColor to red
  else
    call AccountDB .AppendValue tag "User" valueToAdd UsernameInput . Text
    call AccountDB .StoreValue tag UsernameInput . Text valueToStore PasswordInput . Text
    set global userName to UsernameInput . Text
    call UserDataDB .StoreValue tag get global userName valueToStore create empty list
    call UserDataDB .StoreValue tag join get global userName "Data" valueToStore ""
    set FeedbackLabel . Text to "Account creation successful. Please login."
    set FeedbackLabel . TextColor to green
  end
end

```

The procedure `createAccount` shown above is another integrated algorithm that helps create a user's account and mark the designated locations for the user's data in Firebase given that they had provided a valid password and an unique username. The integration of the two procedures `createAccount` and `loginProcedure` helps the overall algorithm perform and regulate the core functions of the login screen of creating accounts and logging in.

2d.

```
to loadUserEntryData entries
do
  if is list empty? list get entries
  then
    call Notifier1 .ShowAlert
    notice "Welcome! Click New to start a new entry"
    set NavBar . Visible to true
    set EntryPage . Visible to false
    set LoginScreenArrangement . Visible to false
  else
    set UserEntryList . Elements to get entries
    set NavBar . Visible to true
    set EntryPage . Visible to false
    set LoginScreenArrangement . Visible to false
```

One abstraction I developed to manage the complexity of my code was the procedure `loadUserEntryData`. `loadUserEntryData` helps populate a list of user's entries and is called multiple times throughout the program using different (albeit only slightly) parameters. Implementing this abstraction improves the readability of the code by reducing redundancy and the overall line count. Instead of repeating the nine lines of code in every place, I would only need to call the procedure `loadUserEntryData`. In addition, this abstraction manages complexity as any future changes that need to be made to loading user entry data can be done in a single place. Overall, this abstraction was a helpful in managing redundancy, length of code, editability, and overall complexity.