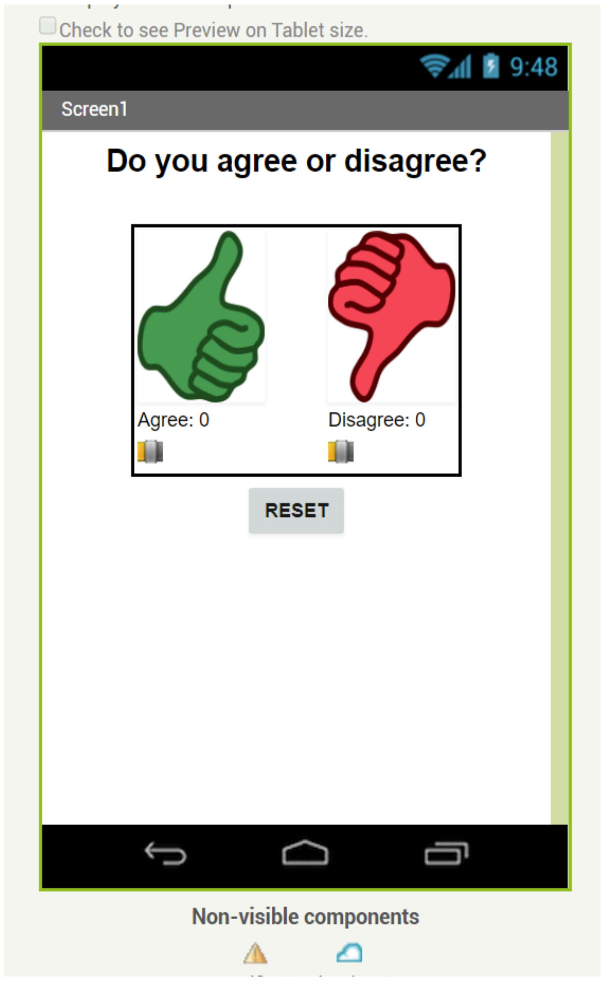
In earlier apps that we designed in this course, we used TinyDB to store and retrieve data on our physical device (phone or tablet). But in this lesson, we will build a simple Clicker App that will store and retrieve data from a cloud database on the web.

Imagine a teacher asking the class a question and students voting on it. We want to design an app that can not only store the results from each student in one central place but also allow the teacher and the students to view the results in real time.

Objectives: In this lesson you will learn to:

- create an app that can be used to poll individuals and store responses on the web in a cloud database;
- understand the concept of ***centralizing and sharing Web data;***
- grasp the difference between ***synchronous*** and ***asynchronous*** operations;
- use a **CloudDB** database.

Short Handout


Check to see Preview on Tablet size.
9:48
Screen1
Do you agree or disagree?
Agree: 0
Disagree: 0
RESET
Non-visible components

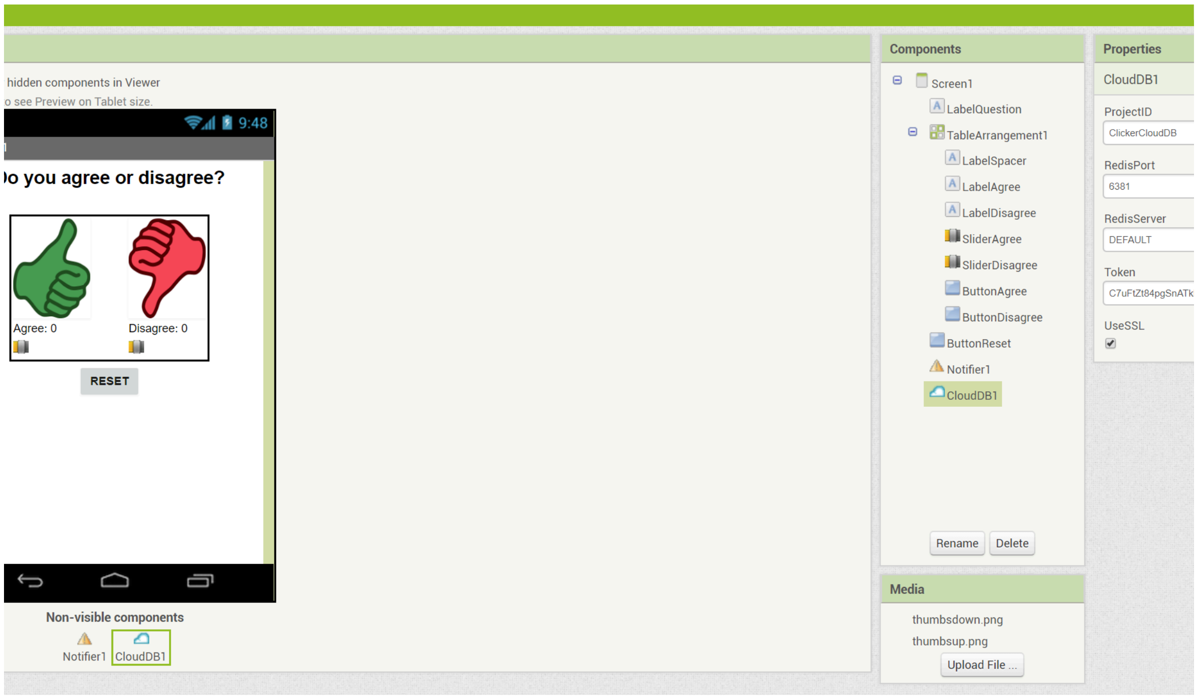
Click to watch video

Introduction: The CloudDB Database

In today's Internet connected world users of mobile devices can benefit by accessing data stored external to the device. In this Clicker App we will poll a set of users and store the results of the poll in a centralized CloudDB database. We will then display the results of that poll in real-time on our mobile devices.

CloudDB is a non-visible component (indicated by the blue cloud logo) that can be used to store and retrieve data values in a database located on the Web. It can be found in the Palette's *Storage* drawer. Whereas TinyDB stores data only on the device running the app, CloudDB can be shared among multiple users and multiple devices running the same app because it is online in the cloud.

CloudDB is a free, experimental, web-based database that is integrated into App Inventor 2. Data can be shared between users by making sure that the *ProjectID* and *Token* of the applications are the same when the CloudDB component is selected (see image, below). This happens automatically if two users load the same .apk file from the same QR code.



Screen1
LabelQuestion
TableArrangement1
LabelSpacer
LabelAgree
LabelDisagree
SliderAgree
SliderDisagree
ButtonAgree
ButtonDisagree
ButtonReset
Notifier1
CloudDB1

Properties
CloudDB1
ProjectID: ClickerCloudDB
RedisPort: 6381
RedisServer: DEFAULT
Token: C7uFtZt84pgSnATk
UseSSL: ✓

Note that the tags are case sensitive in a CloudDB.

Because this app is more easily tested using .apk files, we recommend it be built (and tested) on Android devices until iOS .apk files become available in App Inventor.

Getting Started

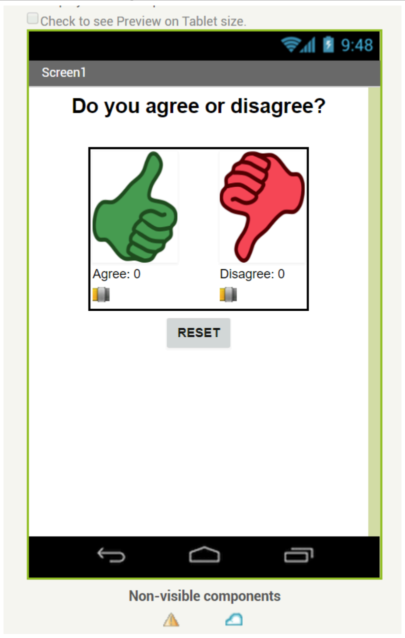
Start App Inventor with the Clicker App Template. Once the project opens, use *Save As* to rename your project *CloudDBstudent.* You should see a User Interface similar to the one shown above.

Note: CloudDB sometimes has connection problems due to server overload. If you get a socket connection error, switch to using the Experimental/FirebaseDB and its associated blocks instead of CloudDB in this tutorial!

Designing the User Interface

Most of the UI is already built for you. Add two new non-visible components: a Notifier and a CloudDB.

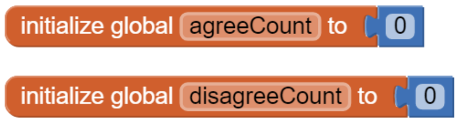
UI Component	Name	Properties
Notifier	Notifier1	● Default settings used
CloudDB	CloudDB1	● Default settings used.


Check to see Preview on Tablet size.
9:48
Screen1
Do you agree or disagree?
Agree: 0
Disagree: 0
RESET
Non-visible components

Note the two non-visible components shown above, the Notifier and CloudDB will be used in the App.

Coding the App

This app uses two variables agreeCount and disagreeCount that count the number of votes when the thumb up or thumb down buttons are clicked. Create these two variables.

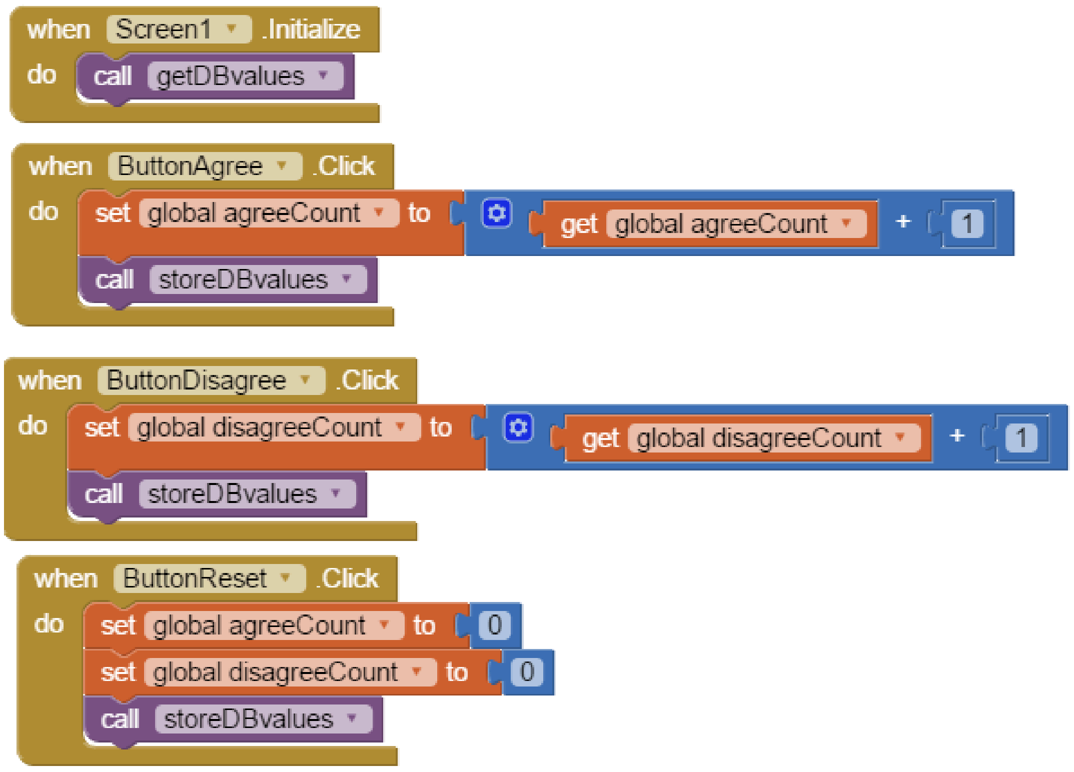


```
initialize global agreeCount to 0

initialize global disagreeCount to 0
```

Variables	Values
agreeCount	0
disagreeCount	0

When the user presses the “Thumbs Up” or “Thumbs Down” buttons, we want to add 1 to the appropriate variable and store it in the database. The RESET button sets the variables back to 0 and stores them in the database. The reset is mostly used for debugging purposes since a real version of the Clicker app would likely not offer this feature to a student user. However, in a later enhancement, we will build a “Teacher” version of this app. For that enhancement, the Reset button will become a functional and important feature for the Teacher user.

Create two new procedures **getDBvalues and storeDBvalues**. You can leave them blank for now. And create the following event handlers. When the app first starts with the Screen1.initialize event handler, you will call getDBvallues. When the user clicks on any of the buttons, you will add 1 to the appropriate variable and call storeDBvalues.



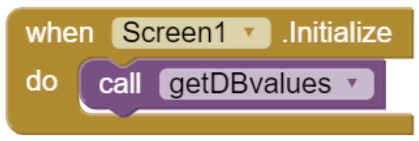
```
when Screen1 .Initialize
do  call getDBvalues

when ButtonAgree .Click
do  set global agreeCount to ( get global agreeCount + 1 )
    call storeDBvalues

when ButtonDisagree .Click
do  set global disagreeCount to ( get global disagreeCount + 1 )
    call storeDBvalues

when ButtonReset .Click
do  set global agreeCount to 0
    set global disagreeCount to 0
    call storeDBvalues
```

In summary, here are the event handlers:

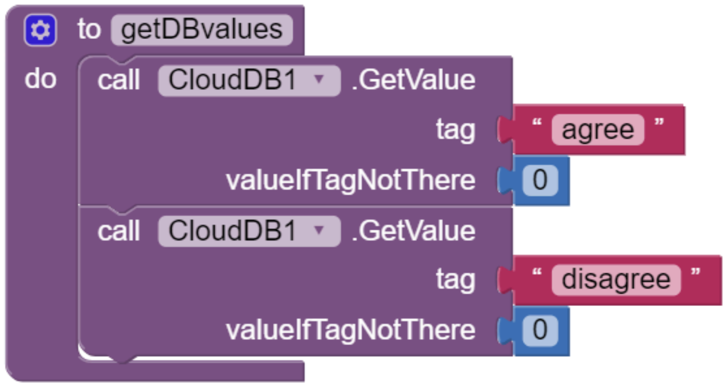
Event Handlers	Algorithms
Screen1.Initialize	Call a new procedure getDBvalues.
ButtonAgree.Click	Add 1 to agreeCount. Call a new procedure storeDBvalues.
ButtonDisagree.Click	Add 1 to disagreeCount. Call procedure storeDBvalues.
ButtonReset.Click	Set agreeCount to 0. Set disagreeCount to 0. Call procedure storeDBvalues.

Get and Store CloudDB Values

When the screen first initializes in our clicker app, we need to fetch data immediately from the database to initialize the screen with whatever values are currently stored in the database.

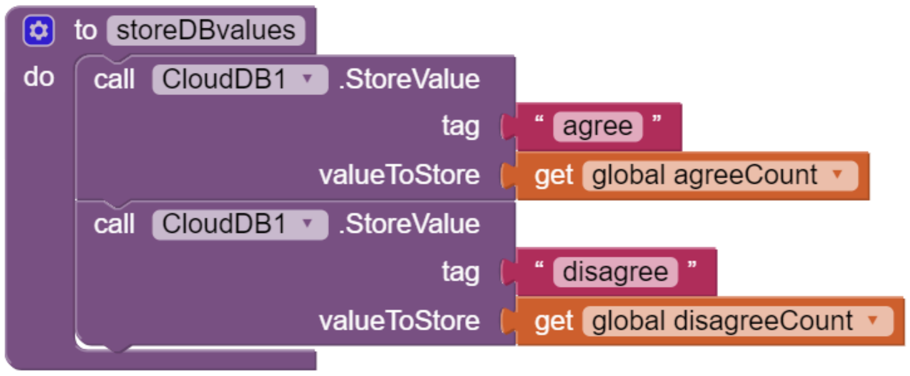


```
when Screen1 .Initialize
do  call getDBvalues
```



```
to getDBvalues
do  call CloudDB1 .GetValue
                     tag  " agree "
       valueIfTagNotThere  0
    call CloudDB1 .GetValue
                     tag  " disagree "
       valueIfTagNotThere  0
```

When the users click on the voting buttons and change the variable value, these must be stored in the database using CloudDB1.StoreValue blocks.

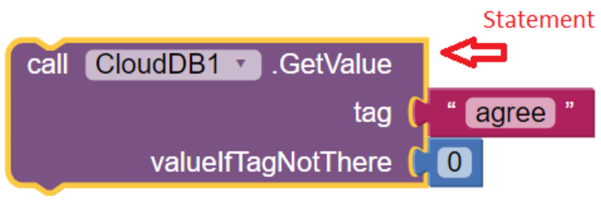


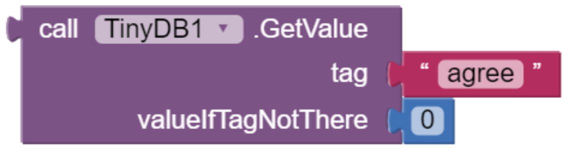
```
to storeDBvalues
do  call CloudDB1 .StoreValue
                tag  " agree "
         valueToStore  get global agreeCount
    call CloudDB1 .StoreValue
                tag  " disagree "
         valueToStore  get global disagreeCount
```

Like the TinyDB, data stored in CloudDB is associated with a unique ***tag***. **Note that this tag is case-sensitive and must match exactly in GetValue and StoreValue!** In this app, we use the tags ***"agree"*** and "**disagree**" as the tags for storing the two variables that keep track of the votes.

Synchronous vs. Asynchronous Retrieval

Notice the difference between the *CloudDB.GetValue* block and the *TinyDB.GetValue:*





The *CloudDB.GetValue* is a ***statement*** *block* whereas the *TinyDB.GetValue* is an ***expression block.*** Notice the puzzle-piece appendage on the top left of the TinyDB expression block, which is missing from the statement block. This difference is crucial to understand. In the case of TinyDB when you retrieve a value you would use the expression block to immediately assign it to a variable, as we did in a previous lesson.

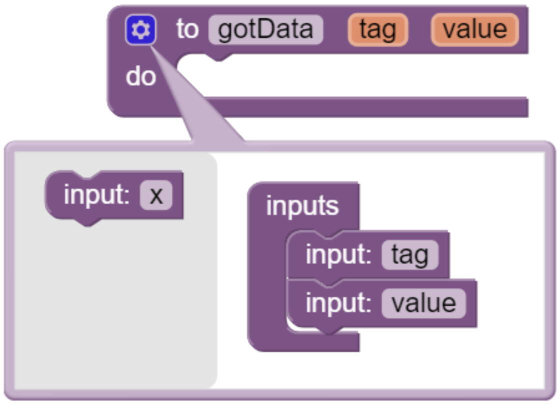
TinyDB.GetValue is an example of ***synchronous retrieval***. This means the retrieval happens immediately. *Synchronous* means *at the same time.* So the retrieval happens at the same time as when the *GetValue* block executes. The TinyDB is stored on the device's permanent storage (i.e., flash drive) and the retrieval is nearly instantaneous. While the retrieval of data from TinyDB is many milliseconds slower than retrieving data from the device's RAM memory (which would take microseconds), the difference would not be noticeable to the user.

However, synchronous retrieval is impossible over the Web. In order to retrieve something from the Web your app must send a request over the Internet. This takes time, usually a few milliseconds but that's not the same as *at the same time.* Therefore, retrieval over the Web is ***asynchronous retrieval*** -- i.e., *not at the same time.*

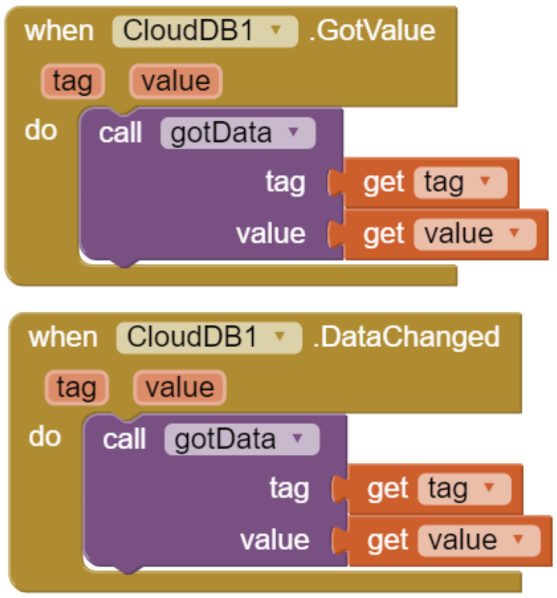
Moreover, things could go wrong when retrieving data over the Internet. For example, your Wifi connection could be slow or dropped. Or, the web site storing the data could be down. Or, any number of other ***I/O problems*** could result***.*** Because of this, the app cannot wait for the data to be received. Instead, when the device receives the requested data it will trigger an event and it will notify your app.

GotValue and DataChanged Events

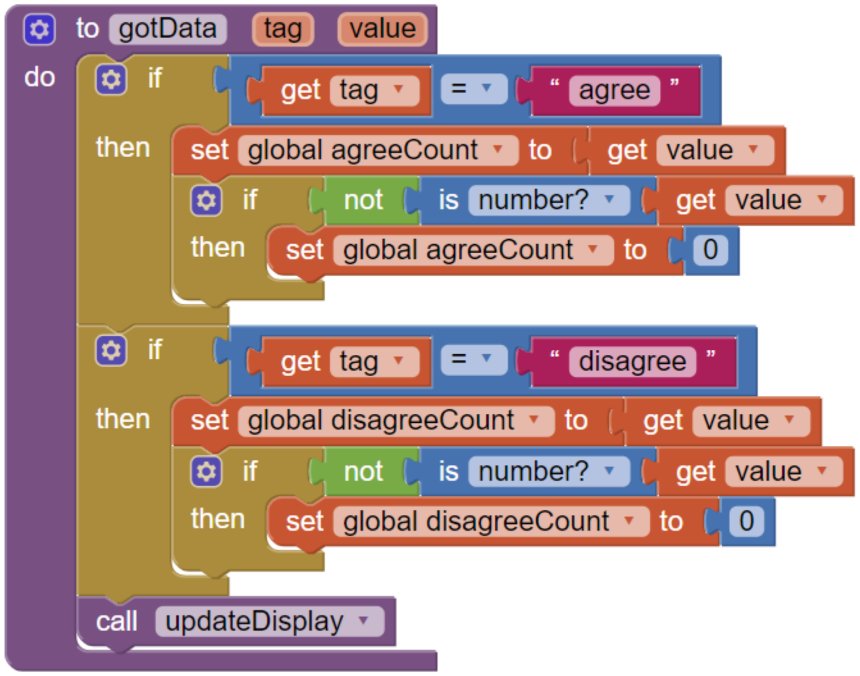
When the data returns from the database, it will call the ***CloudDB.GotValue*** event handler. Or if the data is changed by another user, it will call the ***CloudDB.DataChanged*** event handler. The GotValue event handler is used to respond to the GetValue commands that run when the app first initializes whereas the DataChanged event handler is triggered whenever data in the database changes while the app is running. We can have both of these event handlers call a new procedure called gotData. Create this new procedure and use the blue mutator button to give it two parameters tag and value.



Have both the **CloudDB1.GotValue** and the **CloudDB1.DataChanged** event handlers call this new procedure gotData.

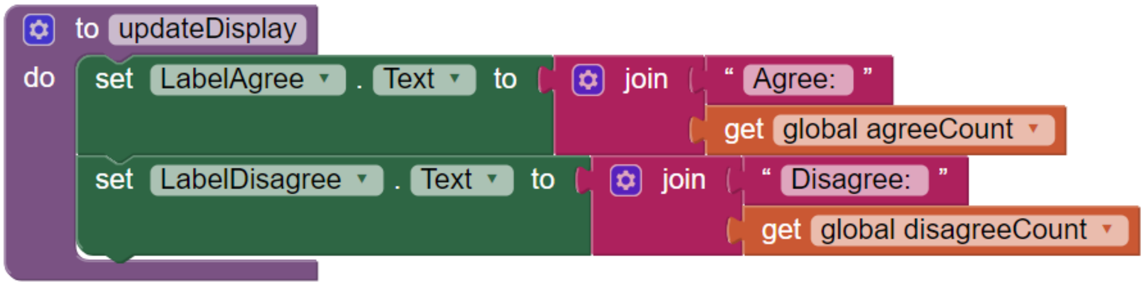

```
when CloudDB1.GotValue
  tag  value
do  call gotData
         tag    get tag
         value  get value

when CloudDB1.DataChanged
  tag  value
do  call gotData
         tag    get tag
         value  get value
```

The *gotData* procedure figures out which query was answered -- i.e., which tag has been requested -- and updates the internal data variable associated with the corresponding response. If we were expecting a number but no number was returned (as may happen the first time the app is run or in the case of an error), we can set the data element to a safe value of zero with the use of the "if not is number ..." statements shown (this part is optional but good error checking). This procedure can also call another new procedure called **updateDisplay.**



```
to gotData tag value
do  if    get tag = "agree"
    then  set global agreeCount to get value
          if    not is number? get value
          then  set global agreeCount to 0
    if    get tag = "disagree"
    then  set global disagreeCount to get value
          if    not is number? get value
          then  set global disagreeCount to 0
    call updateDisplay
```

The updateDisplay procedure, puts the variable values in the labels to show the votes so far:



```
to updateDisplay
do  set LabelAgree . Text to join "Agree: "
                                  get global agreeCount
    set LabelDisagree . Text to join "Disagree: "
                                     get global disagreeCount
```

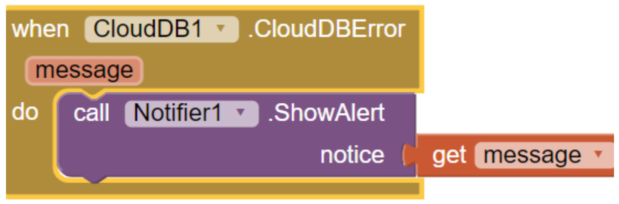
In summary, you will need to develop the following event handlers and procedures to use CloudDB:

Event Handlers	Algorithms
Screen1.Initialize	Call a new procedure *getDBvalues*.
ButtonAgree.Click	Add 1 to agreeCount. Call a new procedure *storeDBValues*.
ButtonDisagree.Click	Add 1 to disagreeCount. Call procedure *storeDBValues*.
ButtonReset.Click	Set agreeCount to 0. Set disagreeCount to 0. Call procedure *storeDBValues*.
CloudDB1.GotValue	Call a new procedure *gotData* with 2 parameters tag and value.
CloudDB1.DataChanged	Call a new procedure *gotData* with 2 parameters tag and value.

Abstraction: Procedures	Algorithms
getDBvalues	Calls CloudDB1.getValue for the tag “agree” and 0 for the valueIfTagNotThere. Calls CloudDB1.getValue for the tag “disagree” and 0 for the valueIfTagNotThere.
storeDBvalues	Call CloudDB.storeValue with tag “agree” and value agreeCount. Call CloudDB.storeValue with tag “disagree” and value disagreeCount.
gotData(tag, value)	If the tag is “Agrees” then set agreeCount to the value returned from the database. Else if the tag is “Disagrees” then set disagreeCount to the value returned from the database. Call new procedure updateDisplay.
updateDisplay	Set labelAgree to “Agree:” joined with agreeCount. Set labelDisagree to “Disagree:” joined with disagreeCount.

Error Handling

If CloudDB returns an error message of some sort (perhaps an Internet issue, a server problem, etc.), we want to display the error message that is received by the app to the user. We, therefore, encode the *WebServiceError* handler as shown using a *Notifier*.


```
when CloudDB1 .CloudDBError
  message
do  call Notifier1 .ShowAlert
                   notice  get message
```

Note: CloudDB sometimes has connection problems due to server overload. If you get a socket connection error, switch to using the Experimental/FirebaseDB and its associated blocks instead in this tutorial!

Testing the App

This app is best tested by forming a group of students where everyone in the group loads one student's app using **Build/App (provide QR code for apk).** Or the whole class could load 1 student’s app projected on the screen. When one of student in your group votes, the latest data should update on everyone's screen. Because this app is more easily tested using .apk files, we recommend it be built (and tested) on Android devices until iOS .apk files become available in App Inventor.

Inputs	Expected Outputs	Actual Outputs
Test clicking agree and disagree buttons with other people in your group.	All apps in the group should update whenever you click on a button and show the agree and disagree counts.	?

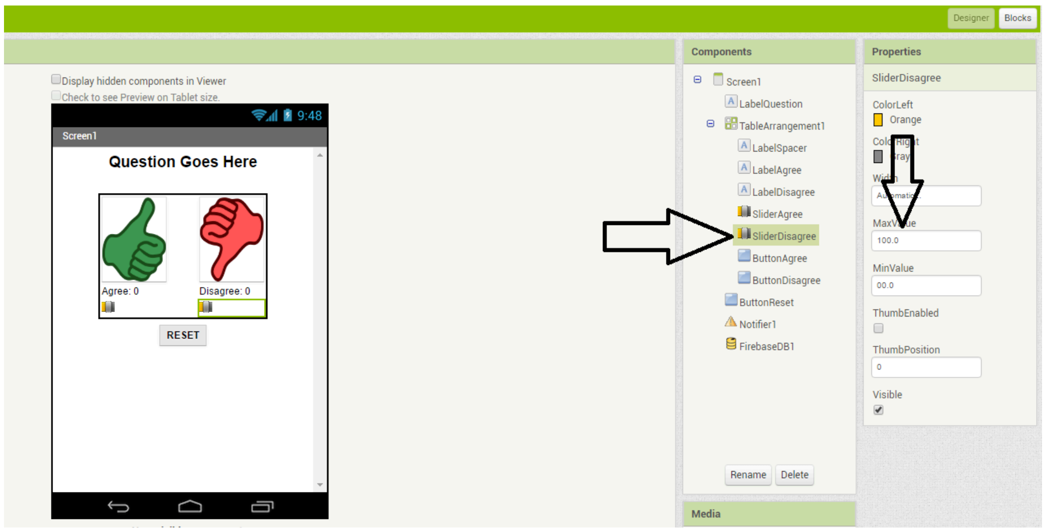
Enhancements

Enhancement #1: Create a Bar Chart Using the Thumb Switches

Read this documentation or watch this video on sliders. Sliders or thumb switches are most frequently used to allow the user to set the value of some property by moving their thumb on a sliding scale. For our Clicker app, we will be using this component in reverse - to create a ***bar chart*** based on the ratio of "Agree" and "Disagree" votes recorded by the app.

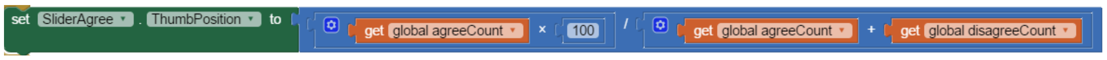
Programming the Sliders

When we add the two Slider components to the User Interface, we have to be sure to set the MaxValue property to 100 since our display will be based on a percentage:



SliderDisagree
ColorLeft: Orange
ColorRight: Gray
Width: Automatic
MaxValue: 100.0
MinValue: 00.0
ThumbEnabled: ☐
ThumbPosition: 0
Visible: ☑

The general idea for setting the Slider ***ThumbPosition*** property is to divide the number of "Agree" or "Disagree" votes by the total number of votes and then multiplying by 100 to determine the percentage of "Agree" and "Disagree" votes.



```
set SliderAgree . ThumbPosition to ( get global agreeCount × 100 ) / ( get global agreeCount + get global disagreeCount )
```

We have to be careful not to divide by zero (which will cause a run-time error) when the app is first started, when the number of “Agree” and “Disagree” votes are both zero. Using IF statements can allow us to set the value of the slider position to zero when the total votes is zero. We have not shown these “IF” blocks; you have to figure them out for yourself.

Enhancement #2: Allow Users to Vote Only Once
Modify the app so that the Clicker only allows the user to vote once (hint: there is an Enabled property for buttons). Votes will still be updated by the DataChanged procedure which is called automatically when the data in the database is updated. Add re-enabling the voting buttons when the user hits reset. You may want to consider turning off this feature when it comes time for you to demonstrate your app to your instructor.

Enhancement #3: Build a Teacher Version of the App
Add a feature that will allow a special version of the app, the “Teacher” version, to update the question displayed on the screen in real time. First in the student app,

- Change the student version of the app to accept new questions while the app is running. This will involve adding code to the ***CloudDB.DataChanged*** event handler to see if the question was changed in the database (use the “question” tag) and changing the question label accordingly and re-enabling the voting buttons. Note that the Question data will consist of a string, whereas the agree and disagree data were numbers.
- Remove the ***RESET*** button from the UI of the student side so that only the teacher can reset the counters.

Build a separate version of the app called "ClickerTeacher" (use File/Save As). Allow only this version to change the questions. Note that when you use File/Save As, the CloudDB ***token*** and ***ProjectID*** will both stay the same, so the student app and the teacher app can share the same database. Also, when testing the app, it may be easier to use QR codes to load the two versions of the app instead of trying to use the Companion.

Note: If using Projects/Save As does not copy the CloudDB token, you may need to copy and paste the token from the student version into a text editor (e.g. a Google doc) and then copy and paste the token from the text editor into the teacher version.

- Replace the Question Label in the teacher version of the app with a TextBox to allow the teacher to update the question field in real time.
- Add an “Update Question” button to the teacher app that will store the new question into the CloudDB database from where it will get pushed to all the users. Remember the tag name you used! Also, reset the counters and store them in the database too.
- Test with your group with one student using the teacher app and the rest using the

corresponding student apps.

Summary

Vocabulary Review

Review the following new vocabulary in this lesson:

- Synchronous data operations
- Asynchronous data operations
- Cloud or web database

Nice work! Complete the Self-Check Exercises and Portfolio Reflection Questions as directed by your instructor.