**Memo**

To: Professor Pisano

From: Jacob Dansey, Changshuo Fu, Christine Low, Urvashi Mohnani, Neil Sanghrajka

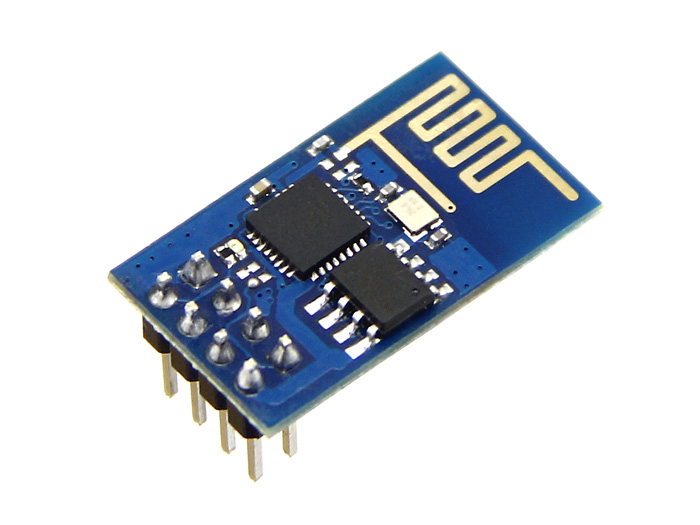
Team: 24

Date:

Subject: Second Deliverable Test Plan



1. **Connecting to Wifi**



* 1. **Description & Goal**

The wifi module used is the ESP8266, which is compatible with the Arduino and most wifi networks. The wifi module works as two different devices. It acts as a bluetooth between the toy and the phone, as well as connecting to the internet to get facts from the server. The initiative behind using just a wifi module instead of both wifi and bluetooth is to save energy and cost. The Arduino acts as an interface between the user and ESP8266.

* 1. **Procedure**

The wifi module and Arduino talk to each other over UART. The Tx and Rx pins on the wifi module is connected to pins on the Arduino that have been defined as Rx and Tx using the SoftwareSerial library. The ESP8266 can handle a maximum voltage of 3.3V only, thus it’s powered by the 3.3V pin on the Arduino. Figure 1.2.1 below shows the circuitry of this connection. The Arduino has a ESP8266 library that supports connection and command transfer between the two. The ESP8266 has to be set to a specific baud rate depending on the firmware and the ESP8266 library has a function called autoSetBaud(), which attempts to connect to the ESP over the different baud rates and sets it to the specific one that worked.

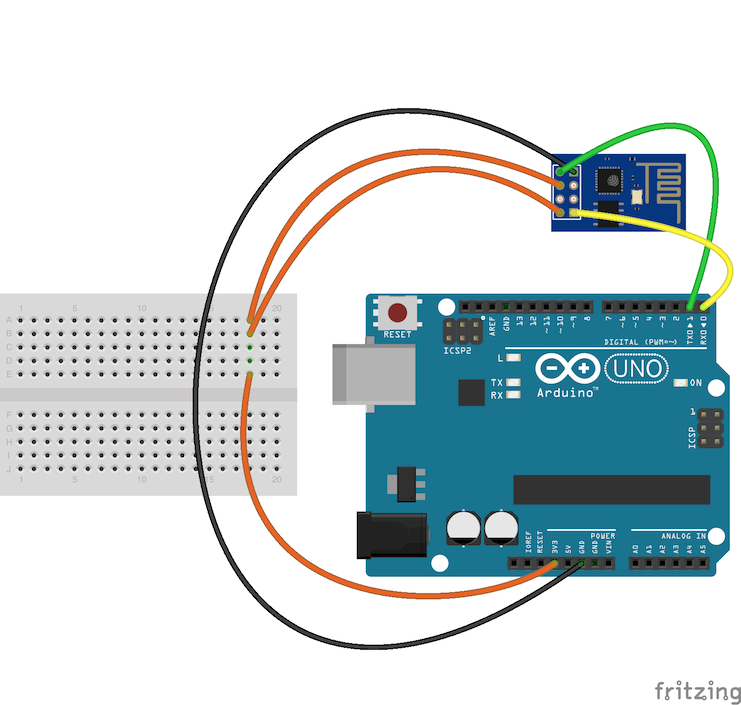


Figure 1.2.1 Connection between Arduino and ESP8266

The wifi module then needs to be told whether to be in Station/Client mode, Access Point(AP) mode or both. The Station/Client mode allows the wifi module to only connect to a wifi network. The AP mode allows the wifi module to act as a router. That is, devices can connect to this network and be on their own private LAN network, but will not be connected to the internet. The last mode enables the wifi module to be in both Station/Client mode as well as AP mode. Therefore, the wifi emits its own signal as well as being connected to the internet.

The wifi module is set to start up in AP mode with a TCP server running on port 333 so that a device can connect to it. Once a device connects to it, the wifi module sends a message to the device informing it that it has received the connection. When the app gets this response, it informs the user that the phone and ball are connected. The mobile app scans for available wifi networks and displays it to the user. The user then selects the wifi network and enters in the password, which the app sends to the TCP server running in the format “ssid:password”.

Once the Arduino receives this information, it breaks it and sends it to the wifi module telling it to connect to the specific wifi network. Once this is done, the wifi module is online and can then connect to the server to get the facts and TCP server is stopped. A reset button will be implemented that will allow the user to change the wifi network they want to connect to. Once the wifi credentials are stored, the wifi module connects to the network automatically.

The module uses AT commands, which gives information about devices connected, statuses of devices connected, network connected to etc. As shown in Figure 1.2.2, the Arduino prints out responses from the wifi module. The IP address of the module, the IP device of the device connected and some other things like whether the TCP has been started is shown.

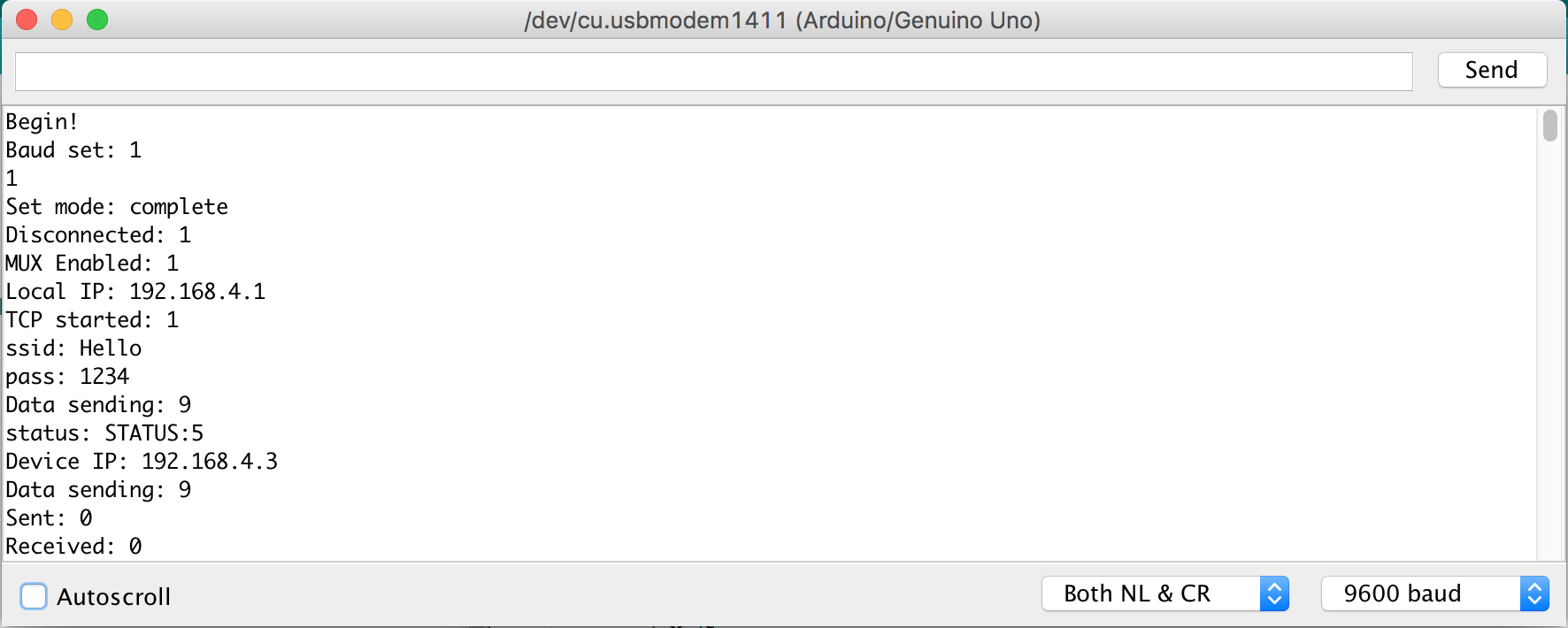
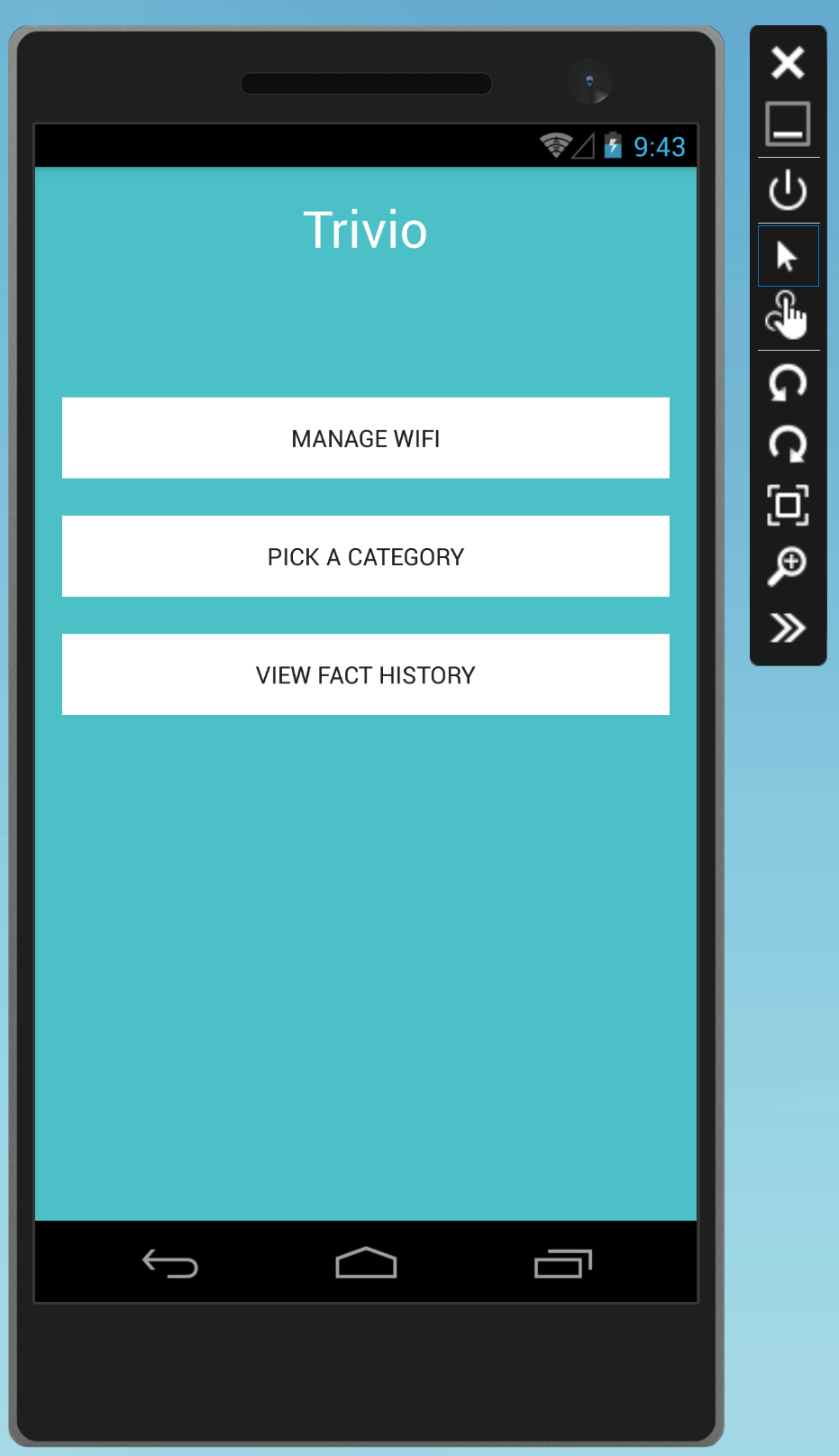


Figure 1.2.2 Arduino Serial Monitor with results from wifi module

* 1. **Verifiable Results**

Connects to the wifi network once credentials are given. The wifi module sends a response once a device is connected to it. However, the wifi module is unable to receive anything from the app so still working on debugging that.

**2.0 Android Mobile Application**

2.1 **Description & Goal**

The main use of the Android mobile application is to scan for the wireless networks available and allow the user to select one and enter the credentials for the Trivia Throw Toy to connect to. The user must create an account and login in order to use the app. The mobile application also serves to enhance the user’s experience by allowing the user to customize the toy by picking fact categories and viewing the history of facts that have been played on the toy. The user will also be able to select a specific fact from the history and upvote/downvote it, assisting the server in gathering data in what is considered a “fun” fact.

* 1. **Procedure**

**2.2.1 User Login and Registration**

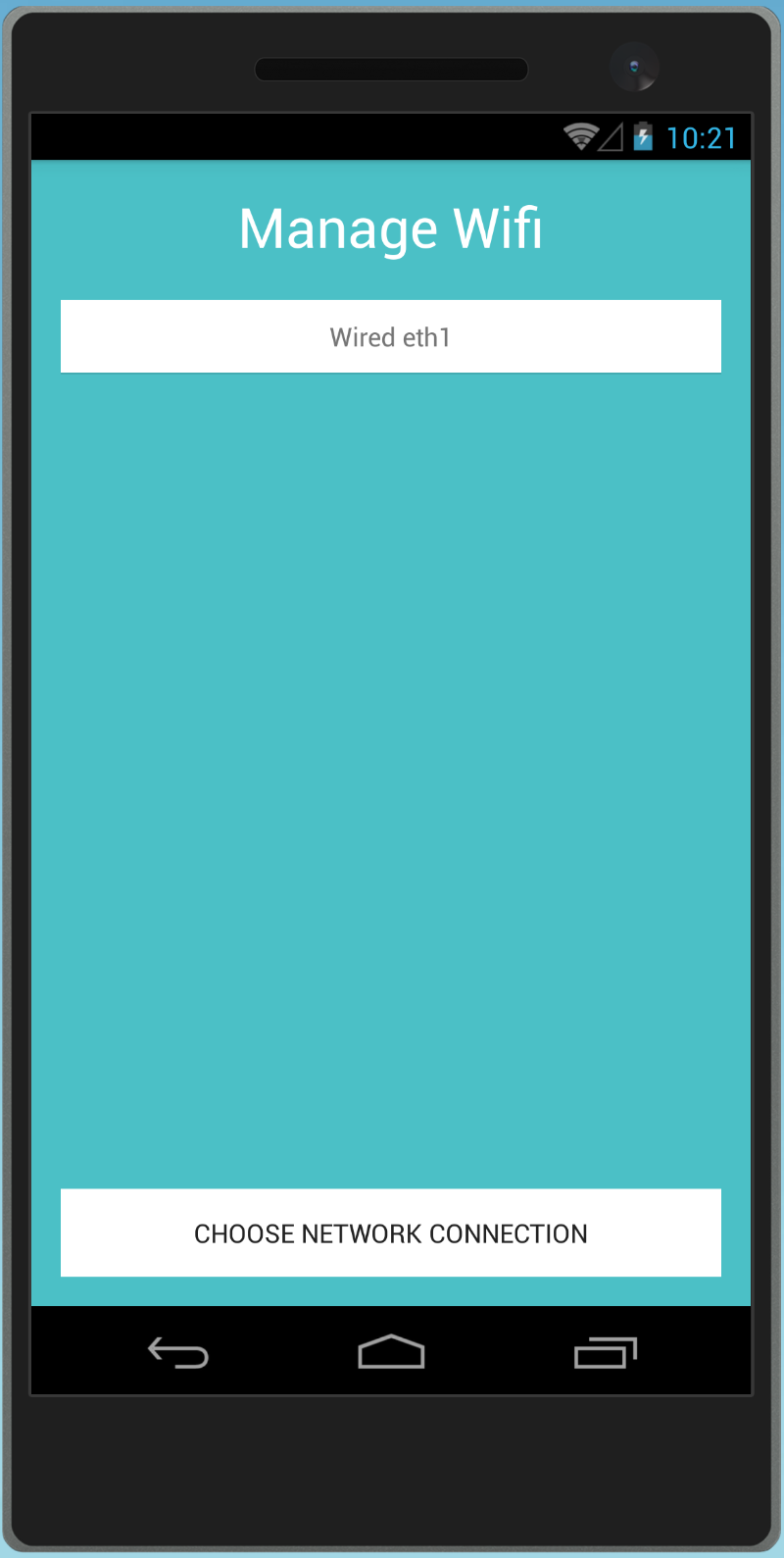
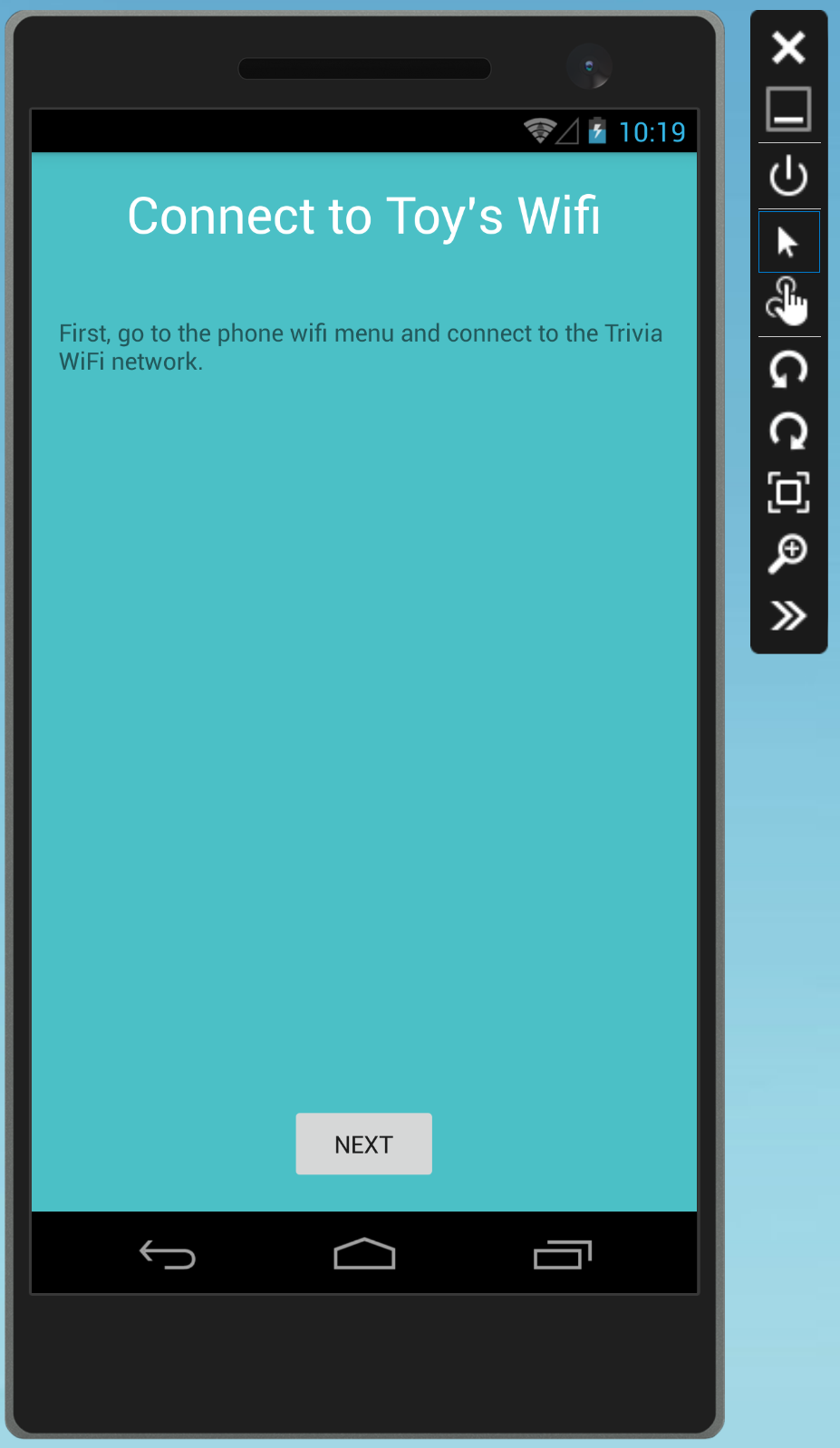
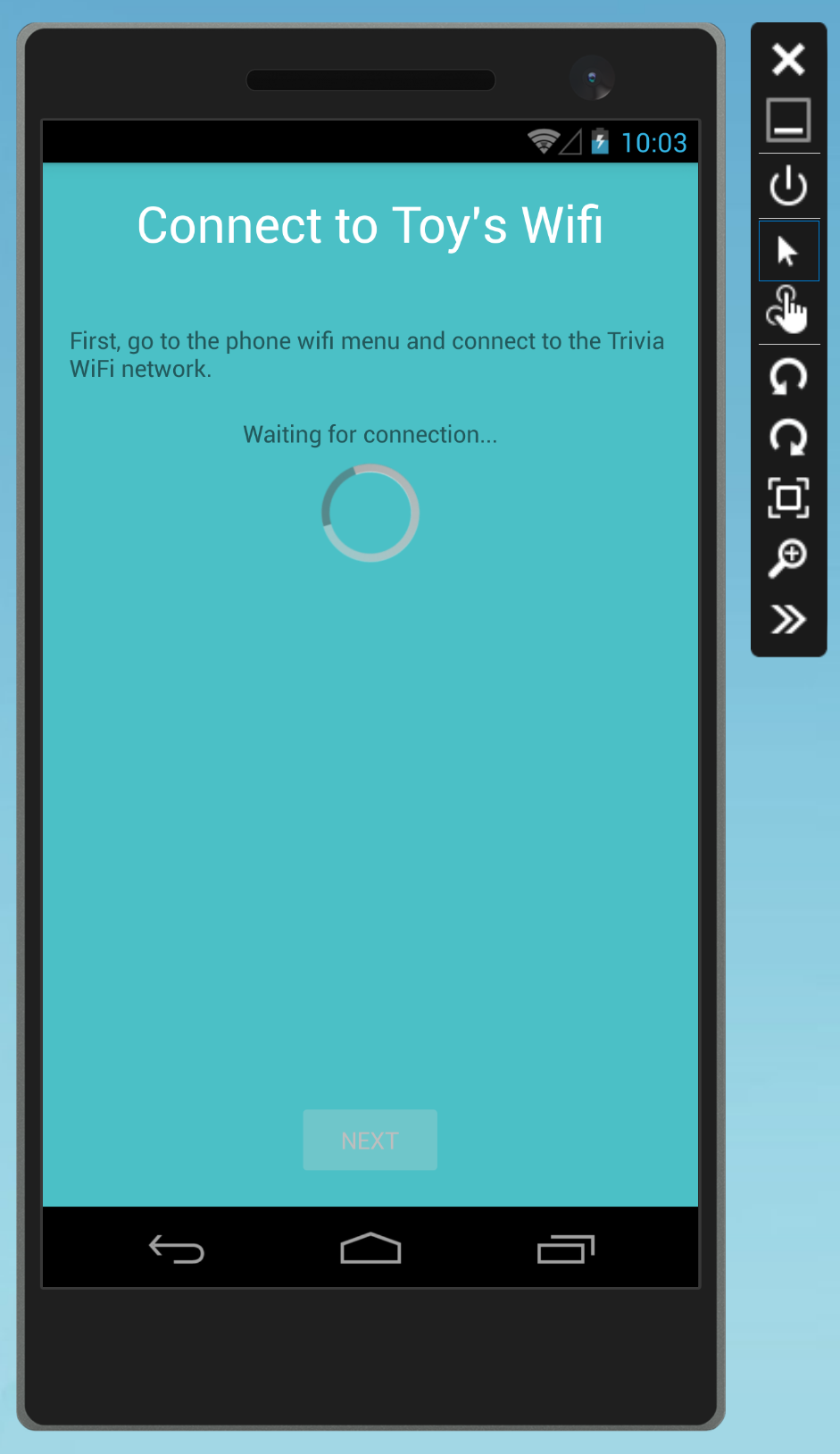
When the android app is launched, it defaults to running *MainActivity* which checks if the user is logged in or not. This is done by creating a class called *SaveSharedPreferences* which contains a function that will save the username to the phone if the user has logged in successfully and a function that checks if there is a username saved, allowing the user to stay signed in between sessions. If there is no username saved, *MainActivity* redirects the user to the sign in page. The sign in page also has the option to register for an account. Once the user fills in the email and password input forms and clicks the sign in button, a POST request is made to the server. For user login, the POST request sends the user email and password in JSON format and will receive from the server, if successful, a JSON object containing the user’s ID. For user registration, the POST request sends the user’s first name, last name, email, and password. Upon successful login, the app redirects the user to the *MainMenuActivity* by starting an intent.

The *MainMenuActivity* consists of three buttons which will allow the user to navigate to the pages of Manage WiFi, Pick a Category, and View Fact History. Clicking on the buttons will start the intents of *CheckPhoneConnectionActivity*, *CategoryActivity*, and *FactHistoryActivity* respectively.

**2.2.2 Managing WiFi Connection for Trivio Toy**

*CheckPhoneConnectionActivity* is responsible for allowing the user to connect the Trivio toy to a wireless network. First, it displays the instruction telling the user to connect the phone to the WiFi the toy emits. Doing this allows the phone application and the toy to communicate through a socket connection. It checks the correct connection by attempting the start the socket connection and if it receives the toy’s socket emitting message that confirms it is the Trivio toy. Once confirmed, it is connected to the toy, it allows the user to continue to the next activity of *ManageWifiActivity* which will scan for the wireless networks available. The user will be able to select a network from the populated list and then enter in the credentials and emits it through the socket connection to the toy. If the toy successfully connects, it will emit a confirmation message that it is connected and the app will display that it was successful.

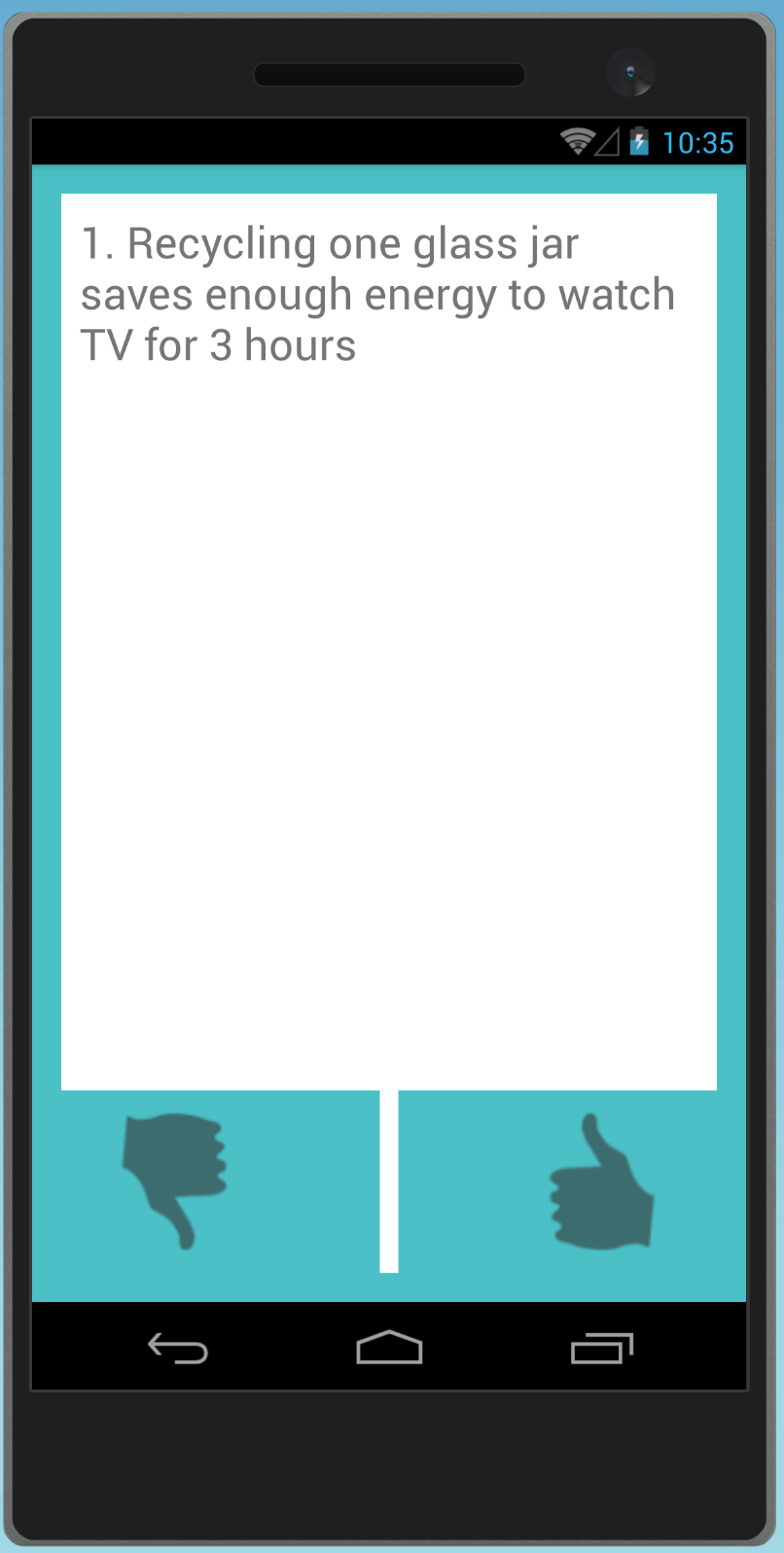
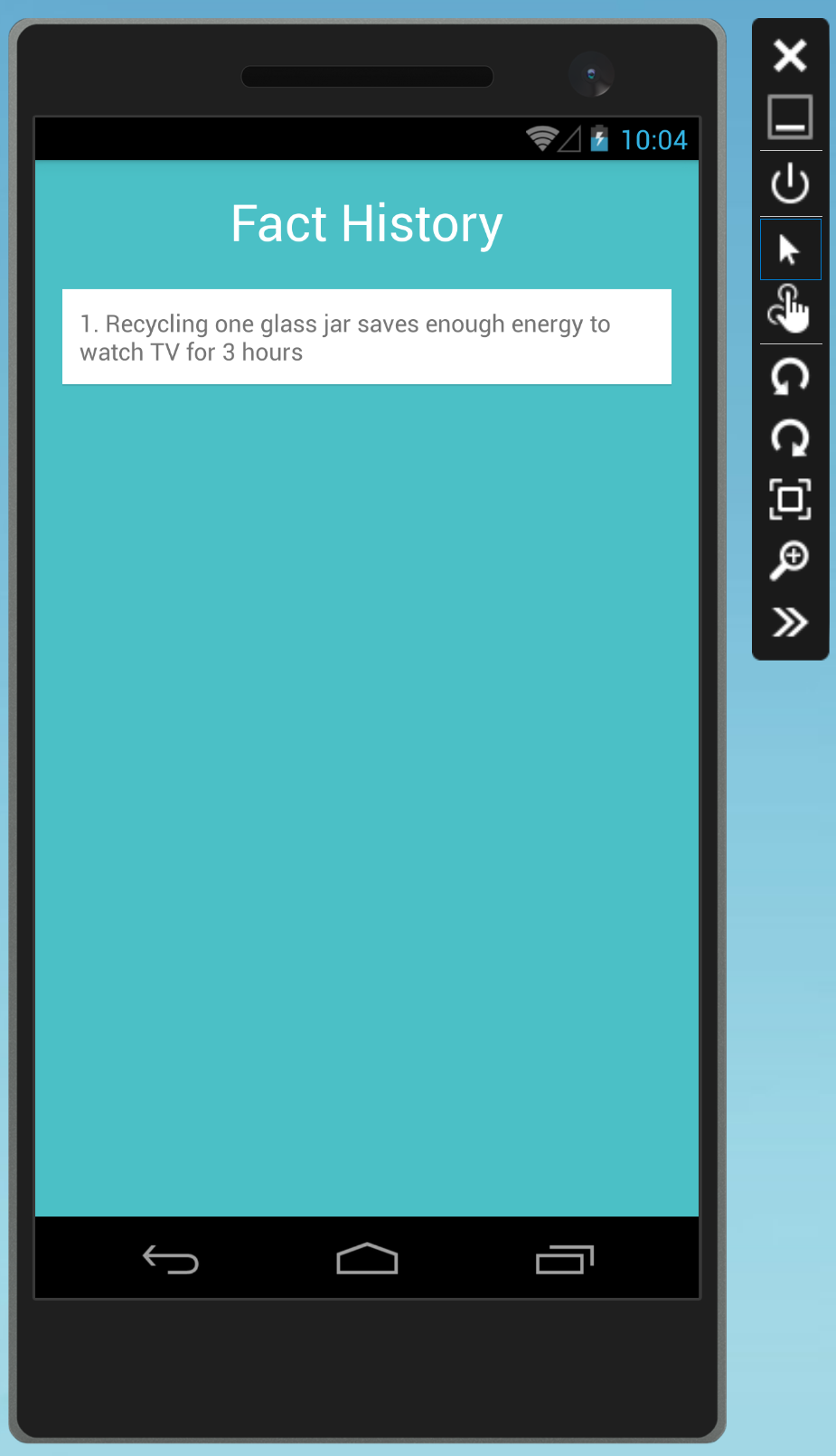
Figure 2.2.2.1 Connecting the Trivio Toy to Wifi via the Android app



**2.2.3 User Fact History Retrieval**

*FactHistoryActivity* is responsible for displaying the facts the user has played on the Trivio toy. It starts an asynchronous task which uses a GET request to the server, giving it the user’s ID. If the GET request is successful, it should receive a JSON object of the list of facts. The activity then parses the list of facts into separate *factItem* objects. It then populates the listview to display them using a custom listview *factItem* arrayadapter which allows the facts to be clickable and redirects the user to the *ViewFactActivity* when clicked, passing the *factItem* to the intent. *ViewFactActivity*  displays the individual fact and allows the user to upvote/downvote it. Clicking on the upvote/downvote button sends a PUT request to the server.

Figure 2.2.3.1 Displaying User Fact History and Voting



**2.2.4 Fact Category**

*CategoryActivity* allows the user to customize what type of facts the toy will play. On the start of the activity, it sends a GET request for the available categories. It receives the JSON object and parses it into a list which displays the categories on the activity. When a specific category is picked, it sends a PUT request to the server, updating what category the user picked.

* 1. **Verifiable Results**

The android app emulator and sdk on the real phone works accordingly with all the POST and GET requests implemented, showing the result on the GUI.

**3.0 IOS Mobile Application**

**3.1 Description & Goal**

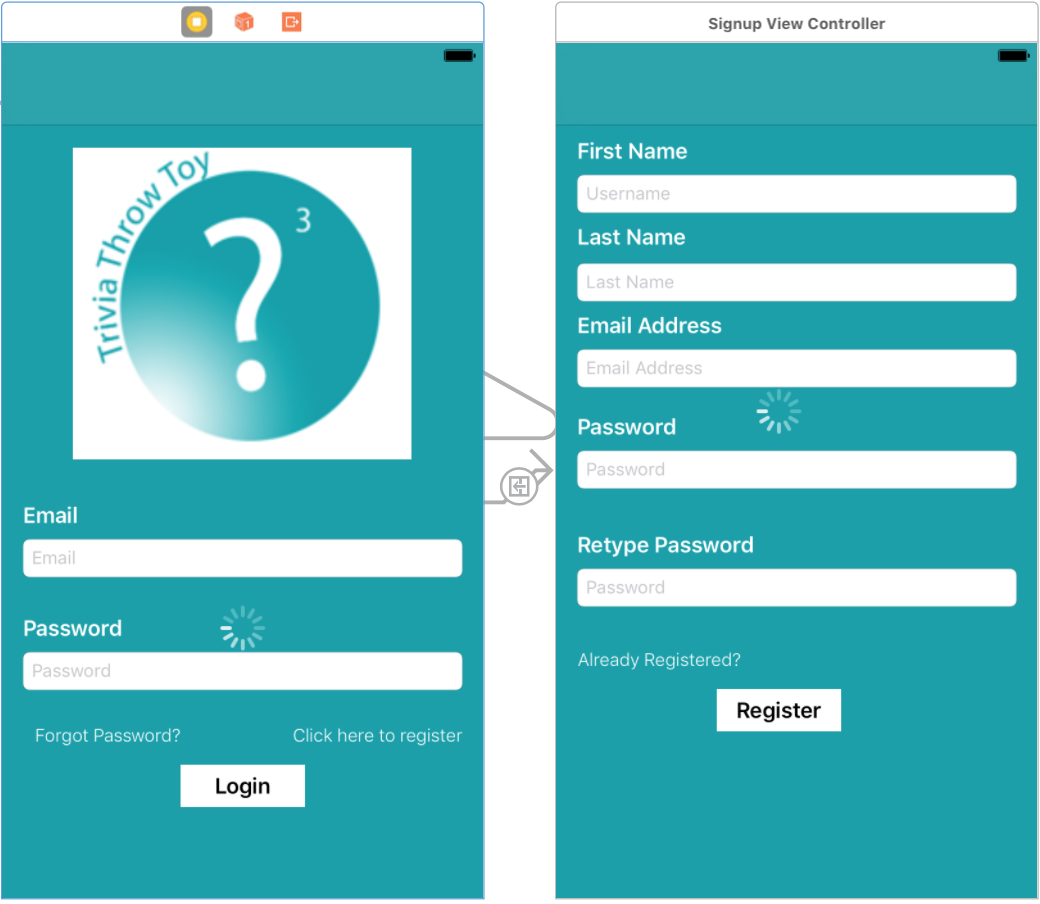
Just like the Android application, the main use of the iOS mobile application is to scan for the wireless networks available and allow the user to select one and enter the credentials for the Trivia Throw Toy to connect to. The user must create an account and login in order to use the app. This account is used to link the toy and the mobile application so that the fact history and category picking are synced. The user will also be able to select a specific fact from the history and upvote/downvote it, assisting the server in gathering data in what is considered a “fun” fact.

**3.2 Procedure**

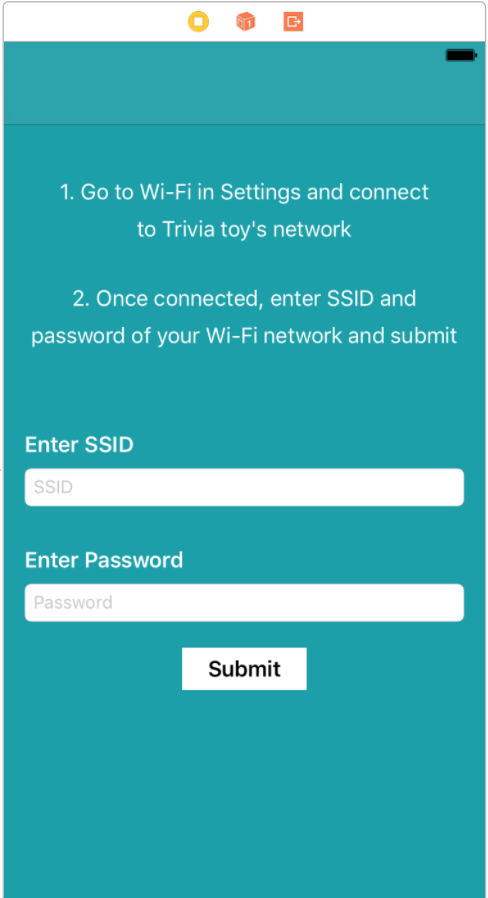
**3.2.1 User Login and Registration**

When a user registers with the application, they input their first name, last name, email and password as unique credentials in the *SignupViewController*. This information is then sent as an asynchronous POST request to the endpoint /Users/Register with a unique userID returned in a successful response JSON. The unique userID is stored on the device in the user defaults location with the value “*userID*”, and a session is initiated. When a unique userID is detected on the login screen, the user is automatically logged in to the application.

If their is no current session going, the user must login to the application with their email and password. The email and password are sent as as asynchronous POST request to the endpoint /Users/Login with their uniqueID returned in a successful response JSON in the *LoginViewController*. This ID is then stored back into the user defaults “*userID*” value to begin a new session and the user is entered into the application. Within the application the unique ID is used in HTTP requests to identify the proper fact history and category selection. When the logout button is pressed, the user defaults for “*userID*” are cleared to end the session and the user is brought to the main login screen.

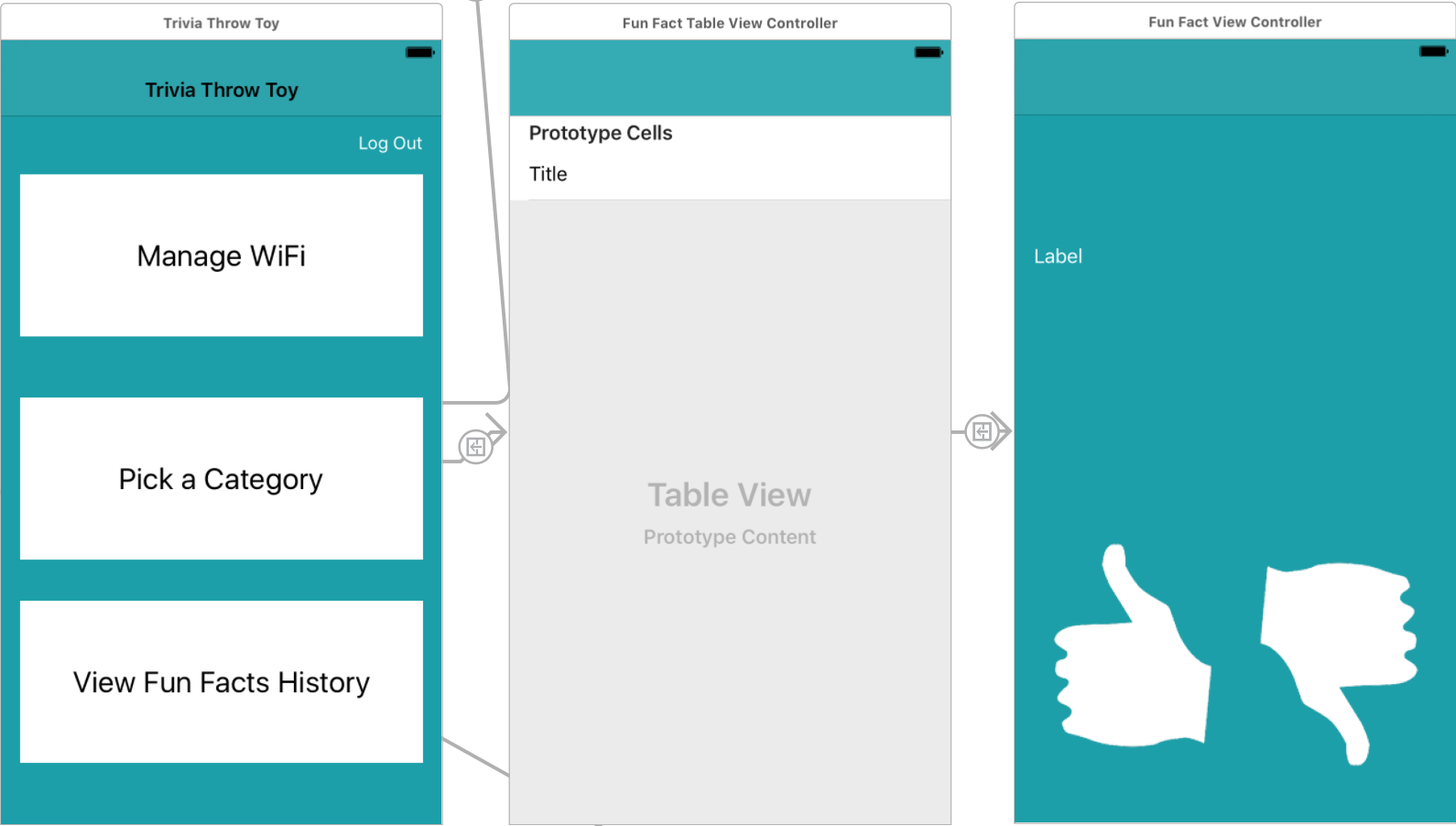


**3.2.2 Manage Wi-Fi Connection**

Managing the Wi-Fi connection is handled by the *WiFiViewController*. Upon entering the view, the user is prompted to go to the settings of the phone and connect to the wi-fi network being emitted by the toy. Doing this allows the phone application and the toy to communicate through a socket connection. It checks the connection by attempting to start the socket connection and then receives the toy’s socket emitting message that confirms it is the Trivio toy. After connecting to the proper network, the submit button is enabled and the user is prompted enter the network credentials of their Wi-Fi network. If the toy successfully connects, it will emit a confirmation message that it is connected and the app will display that it was successful.

**3.2.3 Display Fact History**

The fact history of a user is displayed in a tableview in the *FunFactTableViewController.* Upon entering the table view controller, the userID of the user is pulled from the default preferences on the device. An asynchronous GET request is then sent to the endpoint User/{userID}/history and a JSON response is returned with the user’s fact history. Once the JSON is returned, the tableview has to reload the displayed data due to the nature of asynchronous calls. The JSON is then parsed and displayed in the proper cell of the table view in reverse chronological order. This is accomplished through the use of the function *tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell*. The index path of a cell is related to its location in the tableview and the return value is the cell of that specific index with the information it is displaying. Once a fact cell is selected from the tableview,the current view is segued to the *FunFactViewController* that displays the fact information as well as the ability to up vote/down vote said fact. Prior to this segue, the *FunFactTableViewController* must pass along the fun fact string and unique ID to the next view.



**3.3 Verifiable Results**

The iOS mobile application works properly with all of the GET and POST requests. Users are able to register and log on with a saved user session. When logged in, users are able to check their fact history and log out of the session.

**4.0 Server & Database**

**4.1 Description & Goal**

All the trivia facts are stored on a remote Azure SQL database. The servers acts as an interface between the database and the arduino; hence it is necessary to test whether it's configured correctly.

It is important to ensure that the server is hosted remotely on Azure web services, so that the toy(arduino on toy) can retrieve facts at anytime using internet. Additionally, since most of the Natural Language processing will be done on the server, setting it up now lays foundation for future work.

To achieve the above mentioned functionality, a web based API was created. There are two broad categories of APIs:

* User: These endpoints deal with tasks leading to management of user accounts. This API is primarily used by the smartphone application.
* Trivia: These endpoints deal with modifying and accessing the trivia stored in the database.

Detailed API documentation: <https://app.swaggerhub.com/api/neils95/trivio-api/1.0.0>

This table below provides a summary of API endpoints and expected outcome:

|  |  |  |  |
| --- | --- | --- | --- |
| Endpoint | Description | Parameters | Return values |
| GET:  Trivia/ | Returns JSON collection of all facts stored in database. | None | Status code : 200   * JSON list of Trivia   Status Code: 400 |
| GET:  Trivia/{UserId} | Returns a single Trivia string for user account linked to {UserId} | {UserId:int} in Path | Status code : 200   * Trivia string   Status Code: 400   * Error string |
| GET:  Users/ | Returns JSON collection of all Users stored in database | None | Status code : 200   * JSON list of users   Status Code: 400 |
| POST:  Users/Register | Creates new user account with unique userid. | JSON in request body {password,username,name} | Status code : 200   * Created user object(JSON)   Status Code: 400   * Preexisting account |
| POST: Users/Login | Log in existing user. | JSON in request body {password,username} | Status code : 200   * User object(JSON)   Status Code: 400   * Invalid Login detail |
| GET: User/History/{UserId} | Return array of TriviaId of a particular user’s Trivia history | {UserId:int} in Path | Status code:200   * Array of TriviaId (int)   Status code:400   * Invalid UserId |

**4.2 Procedure**

To test the API and 3 two primary tools are used.

1. Postman: This Chrome Web Application is used to make HTTP requests and view the response from these requests. By testing our server in this manner, and not directly from the smartphone app or arduino we are able to independently test the server code. If we get an error using Postman, we can safely conclude that error is in the server code and not any other component of the app.
2. Visual Studio Server Explorer: This extension inside Visual Studio allows us to view and modify local database tables . Every API call mentioned above, either makes changes to the database or extracts data from a database. By making HTTP requests and then checking the impact on the database using a separate independent tool, we can ensure that the server code functions properly.
3. Azure SQL console: The SQL database is hosted on Microsoft Azure web services. Using the SQL console we can make direct queries to our remote database. This serves a similar function to Visual Studio Server Explorer, however we can now test online databases.

Testing will be performed in the following order.

1. Register a new user: Use Postman to make a call to POST:/User/Register. Request body must contain email and password.

2. Login a new user: Use Postman to make a call to POST:/User/Login. Request body must contain email and password.

3. Save UserId of newly created user: The previous call should return a User object with UserId property.

4. Generate trivia: Use Postman to make a call to POST:/User/Trivia

5. Verify Trivia: Use SQL server tools and visualize the database. Check if Trivia exists in database.

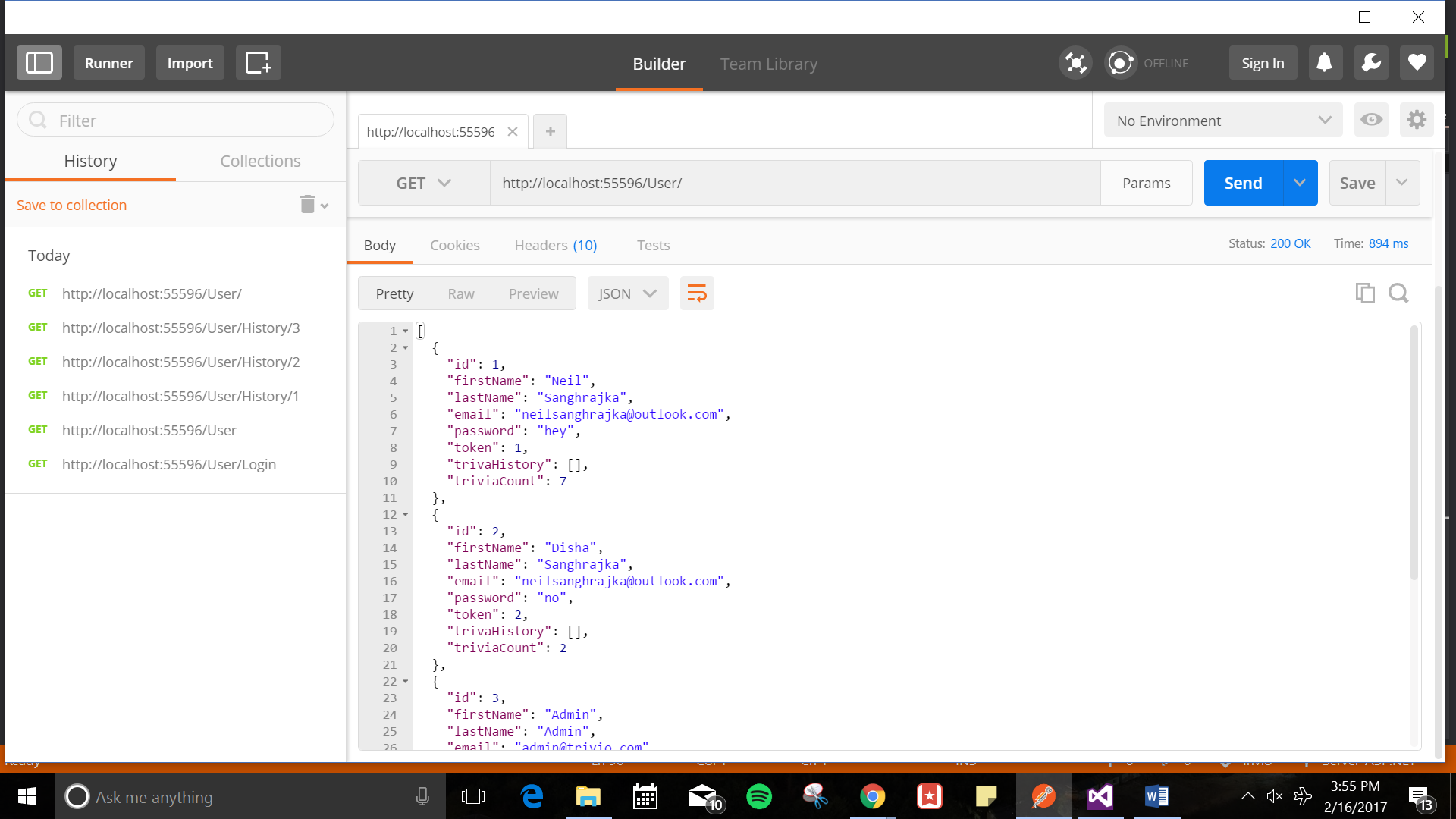
6. Generate trivia again: Repeat steps 3&4. The trivia should be unique..

7. Check User trivia history: Use Postman to make a call to GET:/User/History/{UserId}. The body of return object should contain an array of ids of Trivia heard in step 4 and 6.

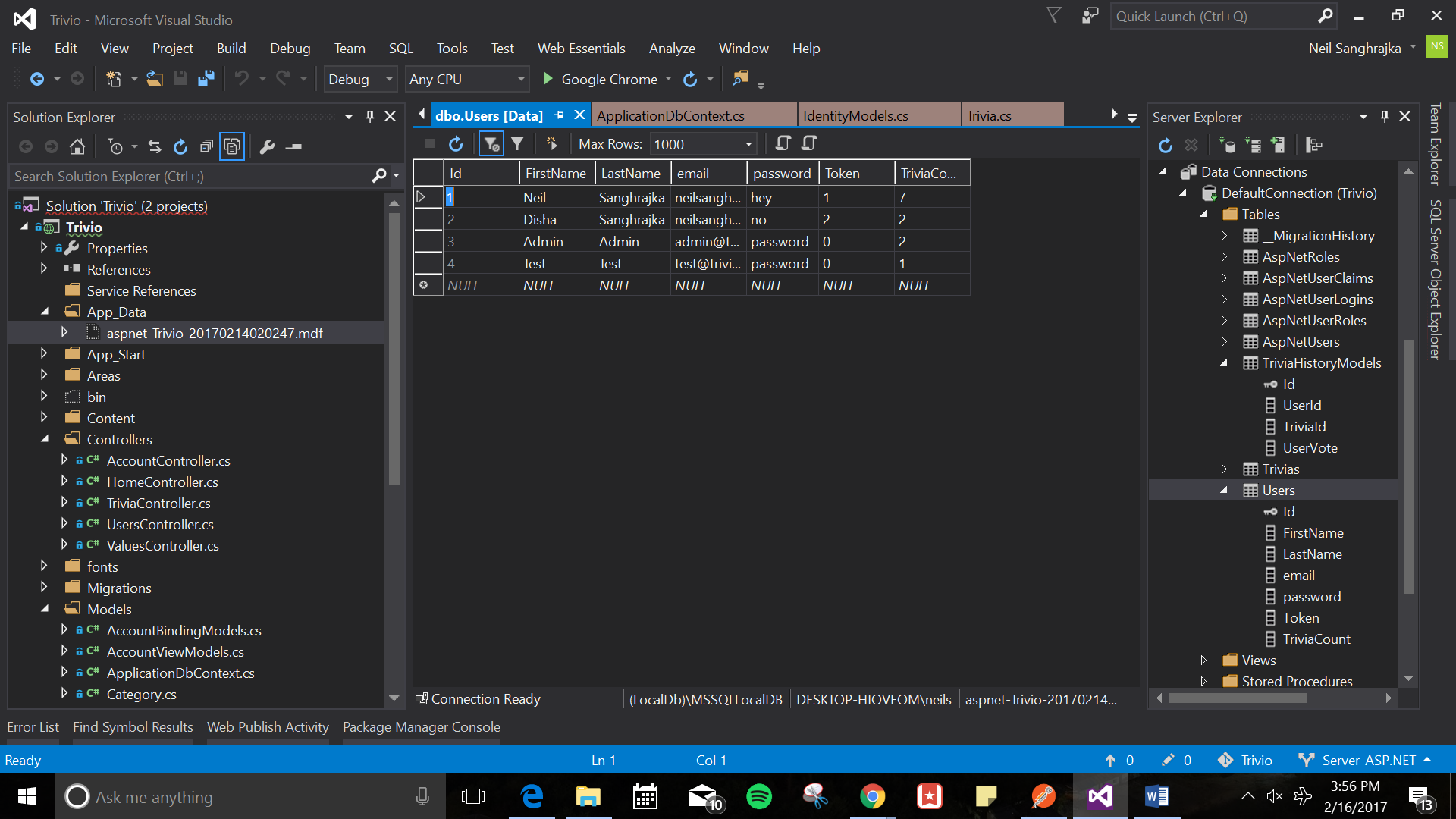
**4.3 Verifiable Results**

The outcome of every HTTP request to the API mentioned above must exactly equal the description. Additionally, the database should show corresponding changes.

This is an example of a GET request as viewed from POSTMAN.



The corresponding database can be viewed using Visual studio.



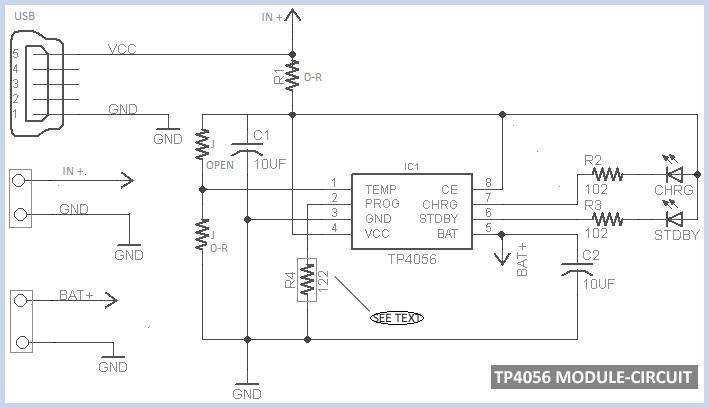
**5.0 Battery and Charging the toy**

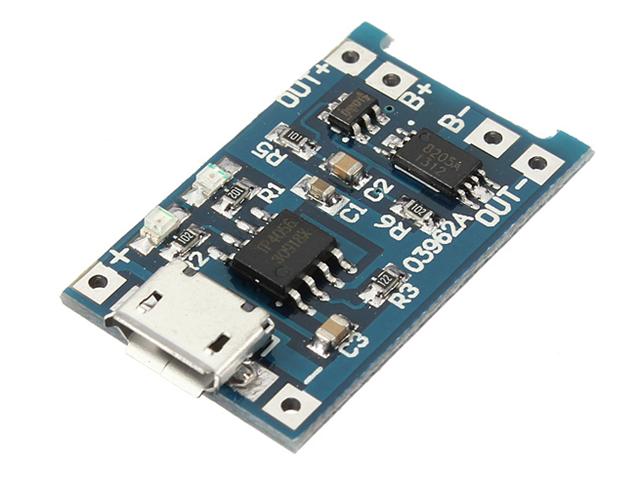
**5.1 Description & Goal**

All hardware within trivia throw toy require lots of charge. In order to make sure every part in the toy works successfully (to detect the motion, the accelerometer must always be on; a high quality speaker is required to ensure outputted sounds are audible and articulate.) a high quality rechargeable battery is needed. The battery life must last at least 3 hours and support charging from standard 120V and 60Hz AC power output.

Since the user need to charge the toy while the toy is running out of battery, the connection between charging port, battery and the entire circuit needs to be established. The system should stop charging the battery automatically when the battery is fully charged to prevent overcharge. Also, the system should show the user that if the power supply is connecting to the battery and the battery is fully charged or not.

**5.2 Procedure**

For battery life that at least can be used for 3 hours and rechargeable without memory effect, cell-phone lithium batteries are the best options. Since these kinds of batteries are thin and light. On one hand they can fit well in the ball and won’t make the toy too heavy. On the other hand, it is very convenient for users to replace if the battery is broken. Anker AK-70SMI9300-S12P1NA are one of the options. The size of Anker AK-70SMI9300 is 2.4x2x0.2 inches; weight: 2.6 ounces. Small and light enough to fit in the ball. The voltage is 3.8V; capacity is 2200mAh. And built-in IC chip to prevents overcharging. Another option is Samsung OEM standard replacement battery(3.8V/ 9.88Wh ) EB-B600BUBEAMZ. The size of this battery is 6 x 0.8 x 9 inches slightly bigger than the first one but it is small enough to fit in the toy. The capacity is 2600mAh but it is lighter than the first one -- 0.8 ounces. All these battery can be used at least 3 hours after determined. The size and the weight also meet the requirement of small enough and light enough.

In order to establish the port to battery, port to circuit and battery to circuit connection, new TP4056 (size:6 x 4 x 0.1 inches, weight: 0.3 ounces) charging module is used. This module is used for single lithium or parallel lithium battery charging. The charging model is linear charging with Micro USB input interface. 

The battery is connected to the B+ and B- terminals and the entrie circuit is connected to the OUT+ and OUT- terminals( Battery to cricuit connection established). If the battery is fully charged, the power will supply by the battery. When there is no charge in the battery which means the toy need to be charged, user need to conncet the Mirco USB port to the power supply. In this case, the red LED on the module should lights up to indicate that the battery is charging. When the battery is fully charged the green LED should lights up that indicate the user to stop charging.( port, power and battery connection established). The entrie syetem should work functionally while charging the battery.

**5.3 Verifiable Results**

With fully charged, the battery works at least 3 hours continuously. Also the battery can be charged normally when it is connected to the power output. The battery -- port-- circuit system works normally when there is no outside power supply connected. When the battery is fully charged, the system will stop charging the battery to prevent the overcharge of the battery and indicate the user stop charging.