Massachusetts Institute of Technology

6.S08 Final Project: MusicBuddy

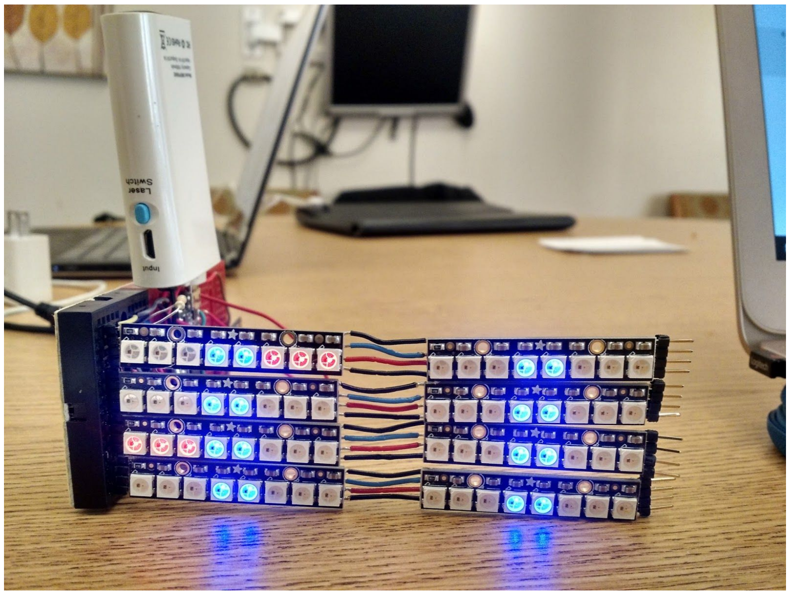
Sarah Aladetan & Matias Hanco

Date: May 12, 2016

Documentation

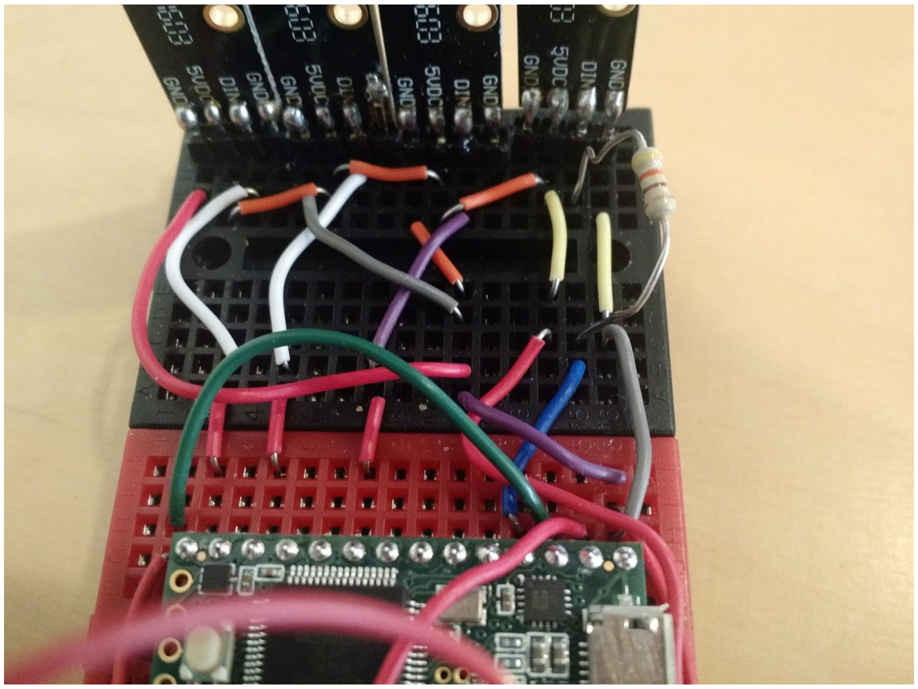
MusicBuddy consists of two separate devices - the device with LEDs to show chords and a remote to select the chords to show.

LED Hardware



The main component of this hardware is the set of NeoPixel LED strips we have. [Insert image of strips] Since we wanted these strips to represent four frets of each string on the ukulele, we decided to have four sets (one for each string) of two strips (two strips could show ~4 frets on a single string) of NeoPixel LEDs that were wired together so that they would be easier to control.

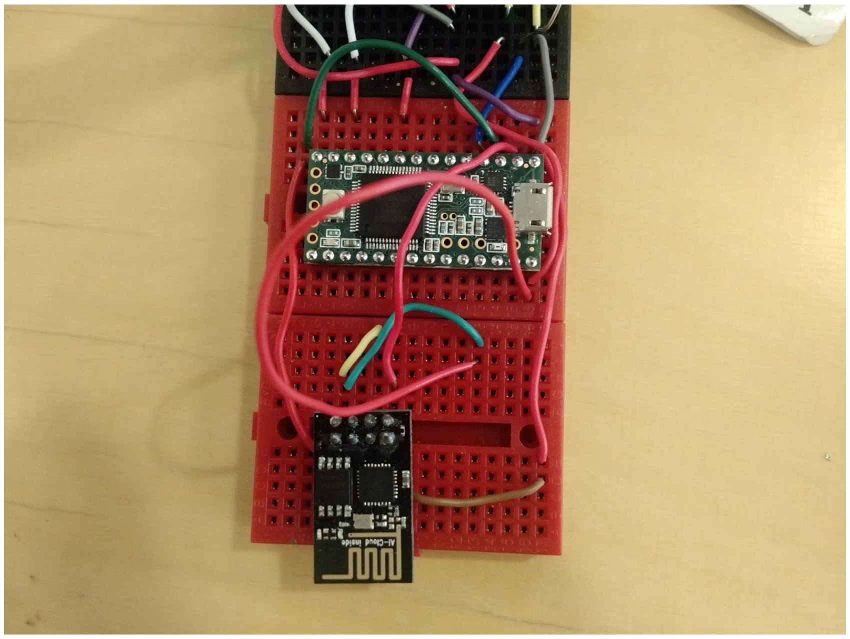
Each strip has four pads - Ground, DIN, 5VCC, and another Ground. Since we want two strips to be connected, you need to solder on small wires on one side of a strip so that the DOUT pad matches with the DIN pad of another strip. This will allow us to communicate with both strips and tell them which of the LEDs to turn on. On the opposite sides of the two strips, we soldered on pins so that we could stick them into the breadboard (you could stick them either way - it's up to preference). Finally, stick in the black pin protectors.



Now that we have the NeoPixel LEDs assembled, we have to connect them to the Teensy. As you can see in the image above, we have the rightmost strip connected to ground, while the rest of the other strips connect ground to a central hub, which from there has a wire connecting to ground. Of course, the layout for the wiring does not have to be exactly the same, it just needs to make sense (and if you can - clean).

The power pins of each strip, however, are all connected via the orange wires (and one yellow wire). Again, we just decided to do this because it seemed more efficient. They all lead to this one red wire that snakes around the Teensy to the Vin pin, which is important. **The NeoPixels require you to use the Vin pin instead of 3.3V since they could require a lot of power at higher brightness.**

The DIN pads had a different wiring. Since each DIN was supposed to be independent from the others, we connected them to different Teensy digital pins, as shown by the small red bridge wires crossing the breadboards. The digital pins can be of your choosing - just remember which digital pin controls which strip, and you're good to go. As for the resistor at the first digital pin, we put it in the circuit because the NeoPixel tutorials told us it would ensure that the Teensy would not break the NeoPixel strip if the strip asked for too much power.

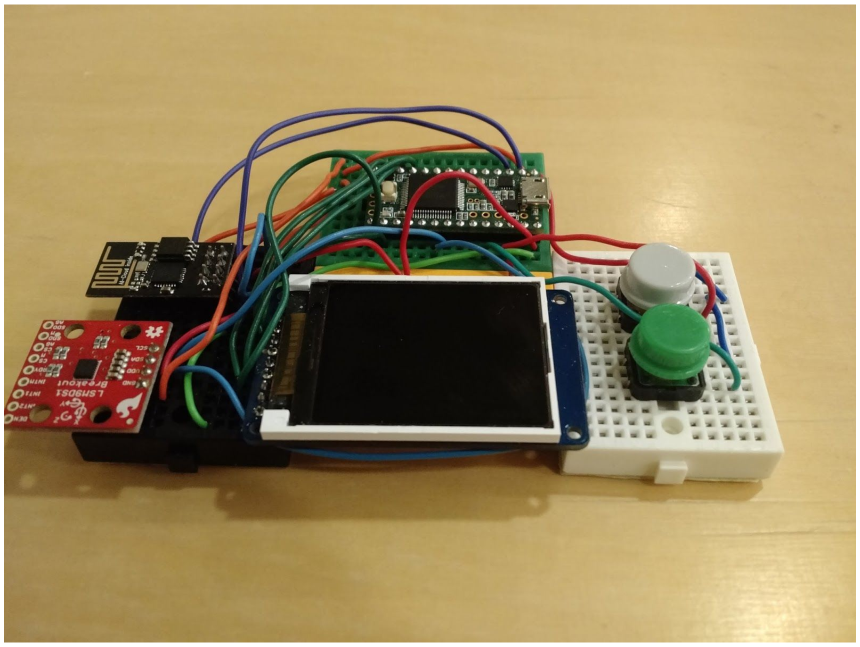


On the other side of the Teensy, we have another breadboard to house the Wifi chip. The setup is much like the tutorial given in one of the 6.S08 lessons, where the Teensy's Ground connects to Ground, the RX of the Teensy connects to the TX of the WiFi, the TX of the Teensy connects to the RX of the WiFi, and the CH-PD also connects to VCC on the chip (as shown by the yellow wire).

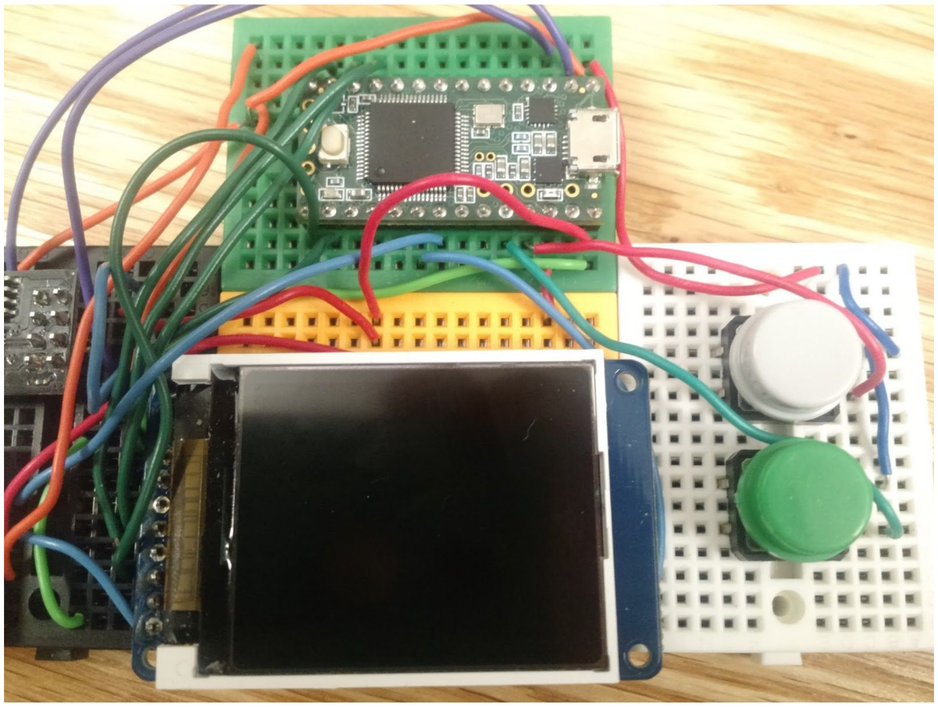
Finally, the WiFi's VCC connects to the Vin of the Teensy, a decision we made so that more power is given to send requests and display them on the NeoPixel strips in case there was a pattern to be displayed.

Once all this is set up, the LED hardware should be ready to be programmed.

Remote Hardware



The remote allows to you get the songs you selected to be added to your own table from the Web UI; from there you can select which song you want to learn and cycle through its chords. The remote includes an LCD Display that we had to buy separately, and it requires more specific wiring. We will touch on how to setup it up next.



Display

The picture above seems a bit obscure, but the green wires are the main ones that the display works with. The pin on the top right of the display is for Ground, so we used orange wires to connect it to a hub of other ground wires, with one last orange wire leading to the Teeny's Ground pin. The pin second to the top right of the display is for power, so we connect it to the Vin pin of the Teensy using the light green wire.

The pin below that is the Reset pin, which we hooked up to pin 14 on our Teensy, but you can actually move it to any digital pin you want.

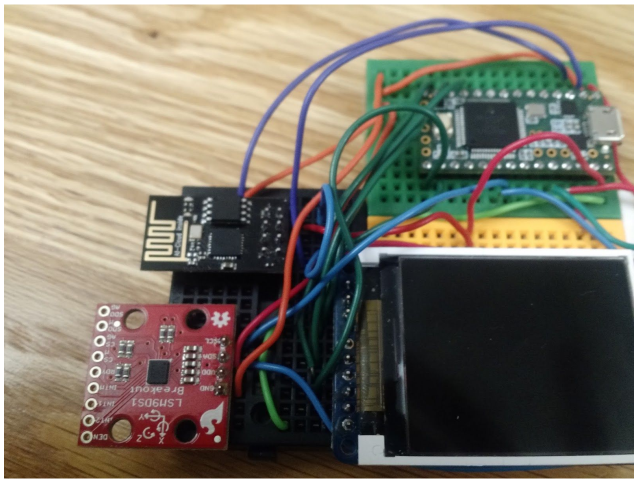
The pin below Reset is the D/C pin, which we set to RX2/ digital pin 9 on the Teensy. We were told by a guide to connect D/C to an RX pin, so that was why we decided to connect it to this specific pin.

Skip the next pin (CARD-CS) since we don't use the card slot. The next pin is called TFT-CS, which we connected to TX2/ digital pin 10, again following the guide that said to connect to a TX pin.

The next pin is MOSI, which we connected to DOUT pin 11. The pin below MOSI is the SCK which we connected to digital pin 13, not because it controls the Teensy's LED, but because it also doubles as the SCK pin

The last pin on the bottom right of the picture is the LITE pin which supports the display's backlight. We have a light green wire connecting it to the VCC pin which will allow it to power up.

We are now done with wiring the LCD display!

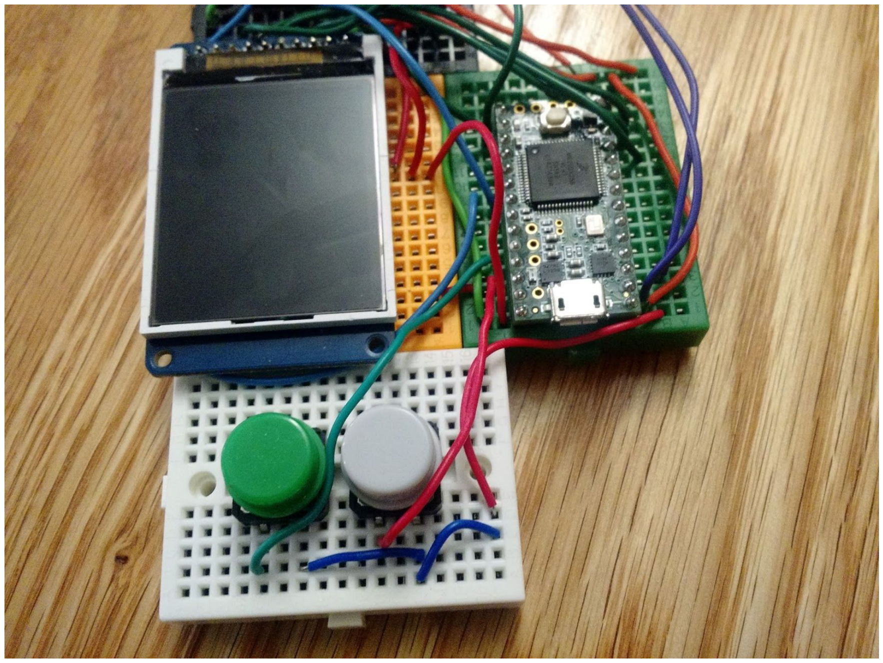


WiFi chip and IMU

The wiring for both components should be straightforward since they are exact same as the 6.S08 labs. There is only one difference - we hooked up both to Vin as shown by the red wires leading out of them to provide more power, especially since the WiFi was having issues loading up at 3.3V.

Just like in the LED hardware, the WiFi's RX and TX pins are wired to the TX and RX pins on the Teensy respectively and by the purple wires. The CH_PD pin is connected via the blue wire to VCC of the chip, and the Ground chip of the WiFi is connected via orange wire to the hub of other ground wires that all lead to the Teensy's ground pin.

The ground pin for the IMU also has an orange wire that goes to that hub, and the SCL and SDA pins of the IMU are connected via blue wires to the SCL0 and SDA0 pins of the Teensy respectively. Finally, the VCC pin has a red wire connecting to Vin of the Teensy. After finishing that, the IMU and WiFi chip will be set up.



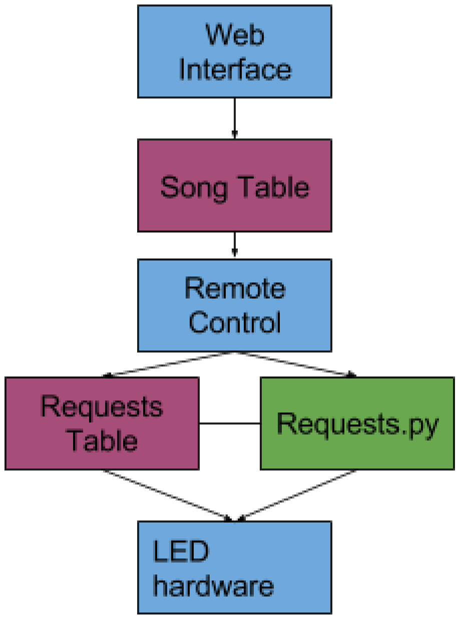
Buttons

The final part of the remote consists of two buttons, a green one for selecting items on the display, and a gray one to move back from a selection. The wiring is pretty standard - simply have one side of the buttons wired to a digital pin that you specify (remember which one to put into the code), and have the other side of each connected to ground, which we did using those blue and red wires that you see in the picture above.

The remote is set! Now all you have to do is code the devices, and they will work like the prototype we demonstrated on Thursday.

Block Diagram

The following block diagram describes the Internet of Things loop our project follows. The loop starts with a **web interface** that allows the user to add/remove songs from their personal **song database**. From there, the **remote control** can make a request to **requests.py** for the songs from song table. When a user selects a song from the remote, a request in the form of a 4 digit pattern is sent to the requests table, using requests.py. From there, the **LED hardware** is able to continuously check for a update to the **requests database table**. If there is a recent update, it will be pushed to the LED device.



Design Challenges

The biggest design challenge we had to overcome was how the LEDs would be displayed and used. Originally, our idea was that the lights would be attached to the ukulele and shine on the individual frets, but we realized that wouldn't work due to the way the strips were connected to each other and the precision of the LEDs. In the end, we decided that the LEDs would instead be a visualization device that the user would look at and then match their fingers on their ukulele to the corresponding lights on the device.

Extra Parts

Description	Item No.	Vendor	Price
LCD Display	**358**	**Adafruit**	**$19.95**

LED Strip	**306**	**Adafruit**	**$27.95**

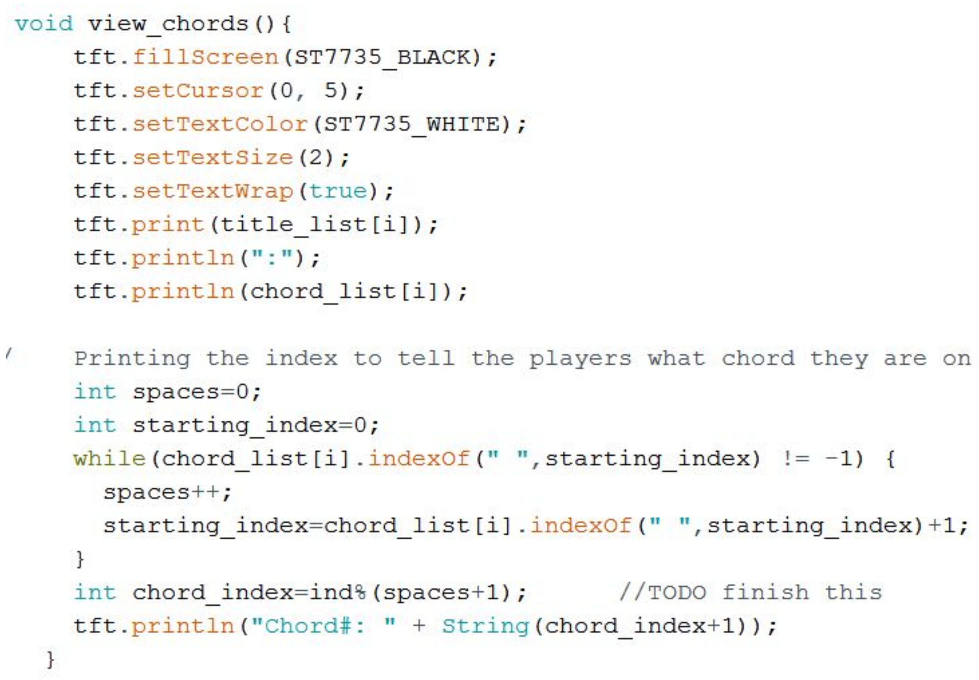
Code Walkthrough

song-tutor.io

The purpose of song-tutor.io is to be the operating system and interface for the remote control hardware. It connects to wifi, receives the user's songs and chords, allows the user to select a song, and sends requests to the server every time a user selects a chord in their chosen song. Also has power management features.

[code sets up declarations and includes libraries]

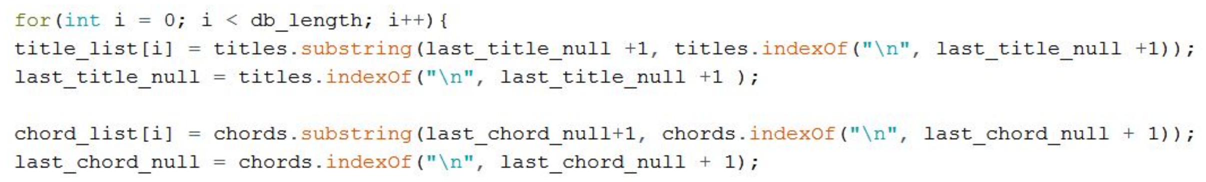
We start out our code file with three different classes: PowerMonitor, Selector, Angle. PowerMonitor and Angle are the same classes drawn from previous labs. Selector is a modification of the code created in the Wikipedia lab. Selector displays the list of all the titles, and allows the user to tilt their device using pitch (because the imu is aligned opposite our display). The user is then able to select a song using buttons on the device.



```
 void view_chords(){
     tft.fillScreen(ST7735_BLACK);
     tft.setCursor(0, 5);
     tft.setTextColor(ST7735_WHITE);
     tft.setTextSize(2);
     tft.setTextWrap(true);
     tft.print(title_list[i]);
     tft.println(":");
     tft.println(chord_list[i]);

'    Printing the index to tell the players what chord they are on
     int spaces=0;
     int starting_index=0;
     while(chord_list[i].indexOf(" ",starting_index) != -1) {
       spaces++;
       starting_index=chord_list[i].indexOf(" ",starting_index)+1;
     }
     int chord_index=ind%(spaces+1);      //TODO finish this
     tft.println("Chord#: " + String(chord_index+1));
   }
```

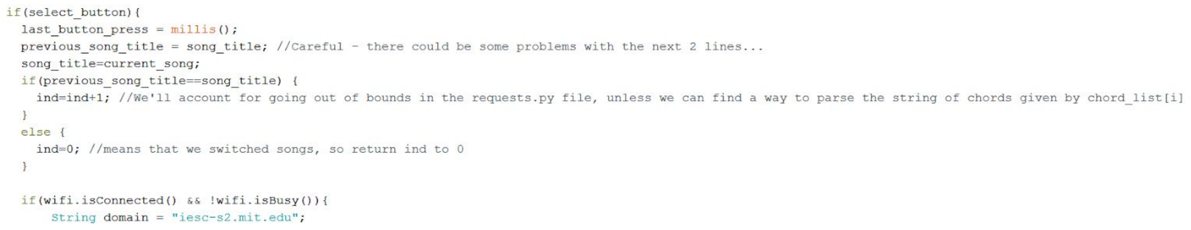
In the Selector class, we have also included a view_chords function that allows the prints the selected song and chords from the update function of the Selector class, and notifies the user which chord they are currently on.

In our setup loop, we change the typical IOT wifi setup a bit by only running the get request for the songs and chords once. After the system is setup, it uses the cached songs. We then parse through the html response to the get the titles and chords, and parse through our strings of chords and song titles to create two lists of each.



```
for(int i = 0; i < db_length; i++){
title_list[i] = titles.substring(last_title_null +1, titles.indexOf("\n", last_title_null +1));
last_title_null = titles.indexOf("\n", last_title_null +1 );

chord_list[i] = chords.substring(last_chord_null+1, chords.indexOf("\n", last_chord_null + 1));
last_chord_null = chords.indexOf("\n", last_chord_null + 1);
```

In our loop, we initialize the buttons and use them to change states. We also run if statements to turn on/off power saving functions based on the set interval. The loop updates the angle of the IMU, and if a movement boolean state is true, it updates the movement of the song selector. When the select button is pushed, it grabs the current song title, as well as the index of current chord selected. It then posts that information to our requests.py file and sets our movement boolean back to false.



```
if(select_button){
  last_button_press = millis();
  previous_song_title = song_title; //Careful - there could be some problems with the next 2 lines...
  song_title=current_song;
  if(previous_song_title==song_title) {
    ind=ind+1; //We'll account for going out of bounds in the requests.py file, unless we can find a way to parse the string of chords given by chord_list[i]
  }
  else {
    ind=0; //means that we switched songs, so return ind to 0
  }

  if(wifi.isConnected() && !wifi.isBusy()){
      String domain = "iesc-s2.mit.edu";
```

When the back button is selected, the screen is cleared, and movement is set back to true to allow for song selection to occur.

led.io

The purpose of led.ino is to act as a simple IoT loop that continuously checks for a new pattern request and maps that request to predefined “frets” on the strings.

[code sets up declarations and includes libraries]

In the constant declarations, 4 pins are assigned to be the four data pins on the NeoPixel LED strip, one strip for each string. Another important thing for code efficiency that occurs is that a list of all the string initializations is created to allow for easier looping through the strings later.

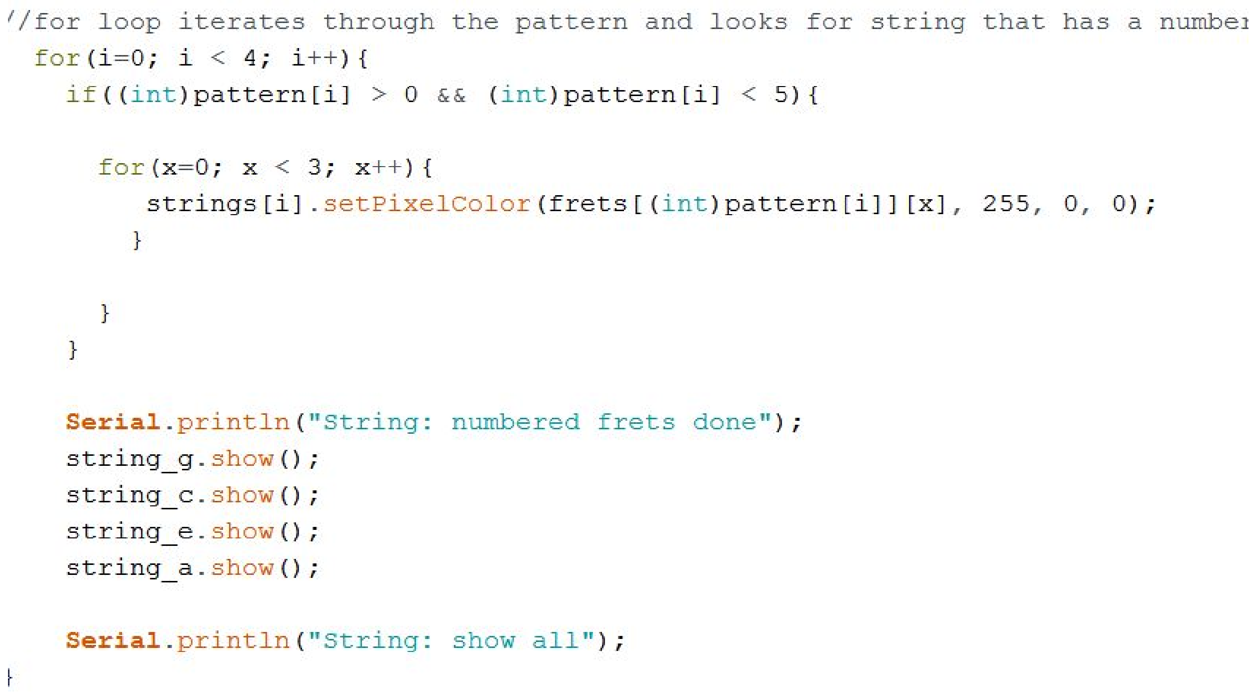
```
Adafruit_NeoPixel string_g = Adafruit_NeoPixel(16, PIN_G, NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel string_c = Adafruit_NeoPixel(16, PIN_C, NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel string_e = Adafruit_NeoPixel(16, PIN_E, NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel string_a = Adafruit_NeoPixel(16, PIN_A, NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel strings[4] = {string_g,string_c ,string_e,string_a};
```

In the loop function, the wifi IoT magic occurs. The loop continuously makes a GET request to the server, where it receives either a 4-string chord pattern or stop code (‘5555’) . After the loop receives the pattern, the code parses the string of fret numbers and converts it into integers to feed to our chord function.

In our chord function, we make the LEDs shine (literally). The chord function takes in four parameters, which are integers correlating to the fret that needs to be lit for that string. We also have a frets matrix that defines the LEDs (in sets of three) that need to be lit up to show that fret.

```
void chord(int g, int c, int e, int a) {


int pattern[4] = {g,c,e,a};

int frets[5][3] = {{100,100,100},{0,1,2},{5,6,7},{8,9,10},{13,14,15}};
```

After that, we have for statements that clear out the LEDs, set their brightness, and then light up defined “fret dividers” for easier visualization of the chords. Following that, we have one last for loop that iterates through the pattern fed to the chord function and using our frets matrix, lights up the specified frets.

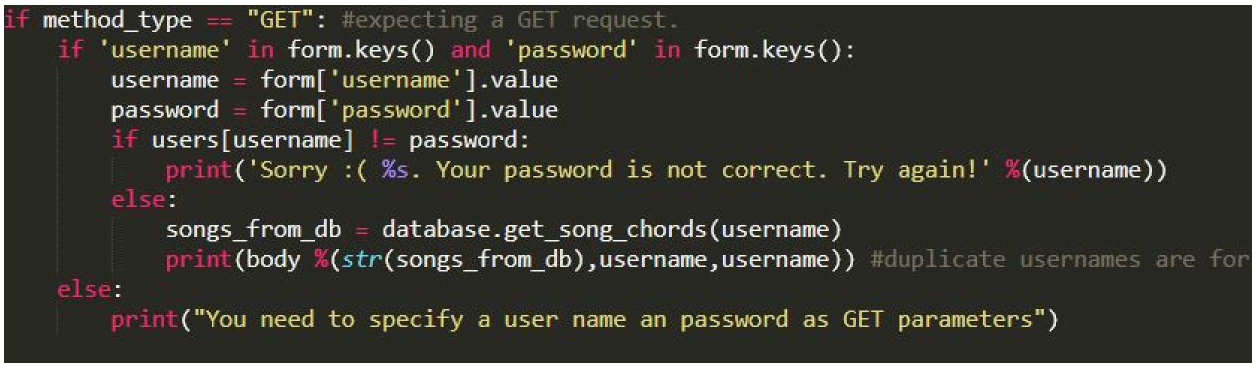


```
//for loop iterates through the pattern and looks for string that has a numbe
  for(i=0; i < 4; i++){
    if((int)pattern[i] > 0 && (int)pattern[i] < 5){

      for(x=0; x < 3; x++){
         strings[i].setPixelColor(frets[(int)pattern[i]][x], 255, 0, 0);
        }

      }
    }

    Serial.println("String: numbered frets done");
    string_g.show();
    string_c.show();
    string_e.show();
    string_a.show();

    Serial.println("String: show all");
}
```

web.py

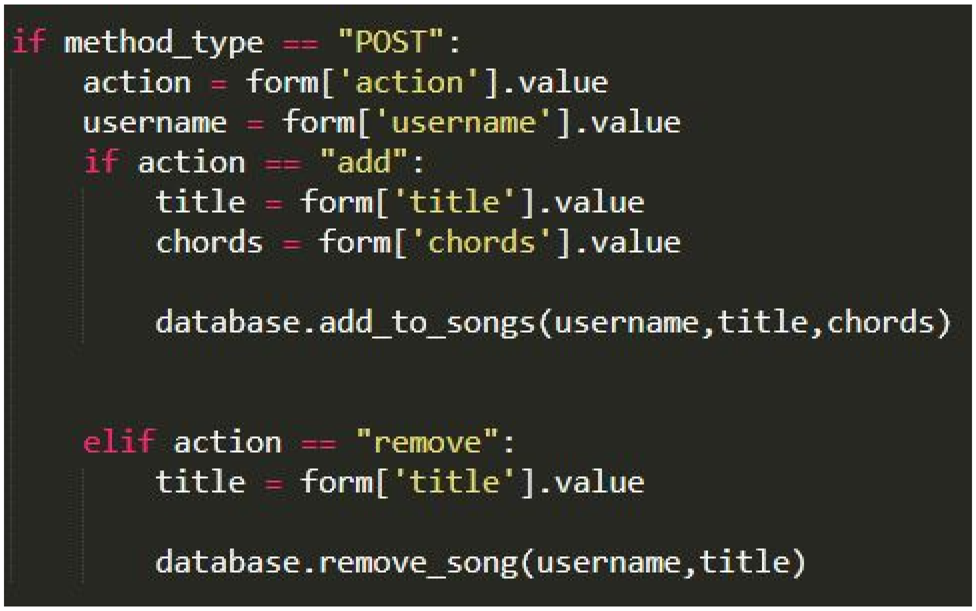
The purpose of web.py is to act as an all-in-one web client. It is the front-facing web interface for users to view their songs in their personal database, as well as take in the post requests needed to add/remove songs from the database.

The code is basically split up into two sections, the GET and the POST request. For the GET request, the code acts similarly to the messaging client lab in that it expects a valid username/password from the users dictionary. If it receives a valid combo, it will print the body variable (which holds all the html/css/javascript) for the page, feeding in songs from a request to the database class.



```
if method_type == "GET": #expecting a GET request.
    if 'username' in form.keys() and 'password' in form.keys():
        username = form['username'].value
        password = form['password'].value
        if users[username] != password:
            print('Sorry :( %s. Your password is not correct. Try again!' %(username))
        else:
            songs_from_db = database.get_song_chords(username)
            print(body %(str(songs_from_db),username,username)) #duplicate usernames are for
    else:
        print("You need to specify a user name an password as GET parameters")
```

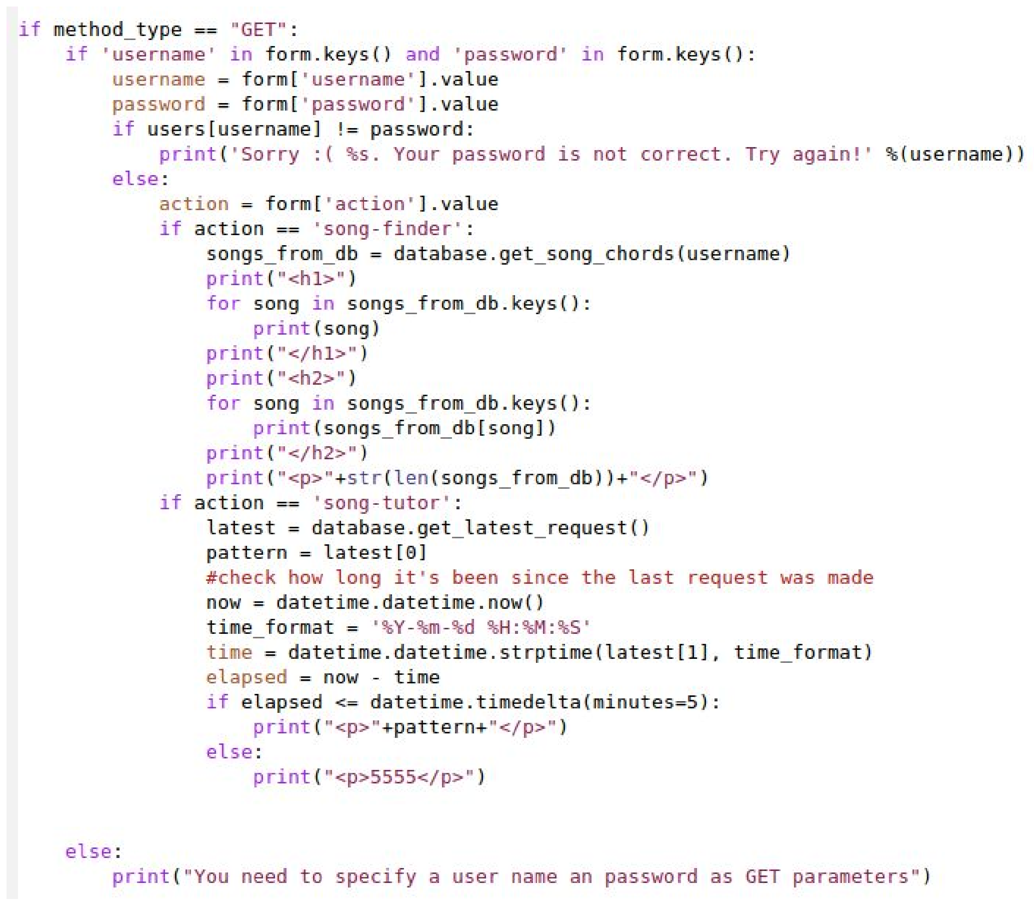
For the POST request, it branches down two paths based on what action parameter is received.If the action is ‘add’, the post request adds the specified title and chords to the database. If it’s ‘remove’, it removes that specific title from the user’s database.



```
if method_type == "POST":
    action = form['action'].value
    username = form['username'].value
    if action == "add":
        title = form['title'].value
        chords = form['chords'].value

        database.add_to_songs(username,title,chords)


    elif action == "remove":
        title = form['title'].value

        database.remove_song(username,title)
```

requests.py

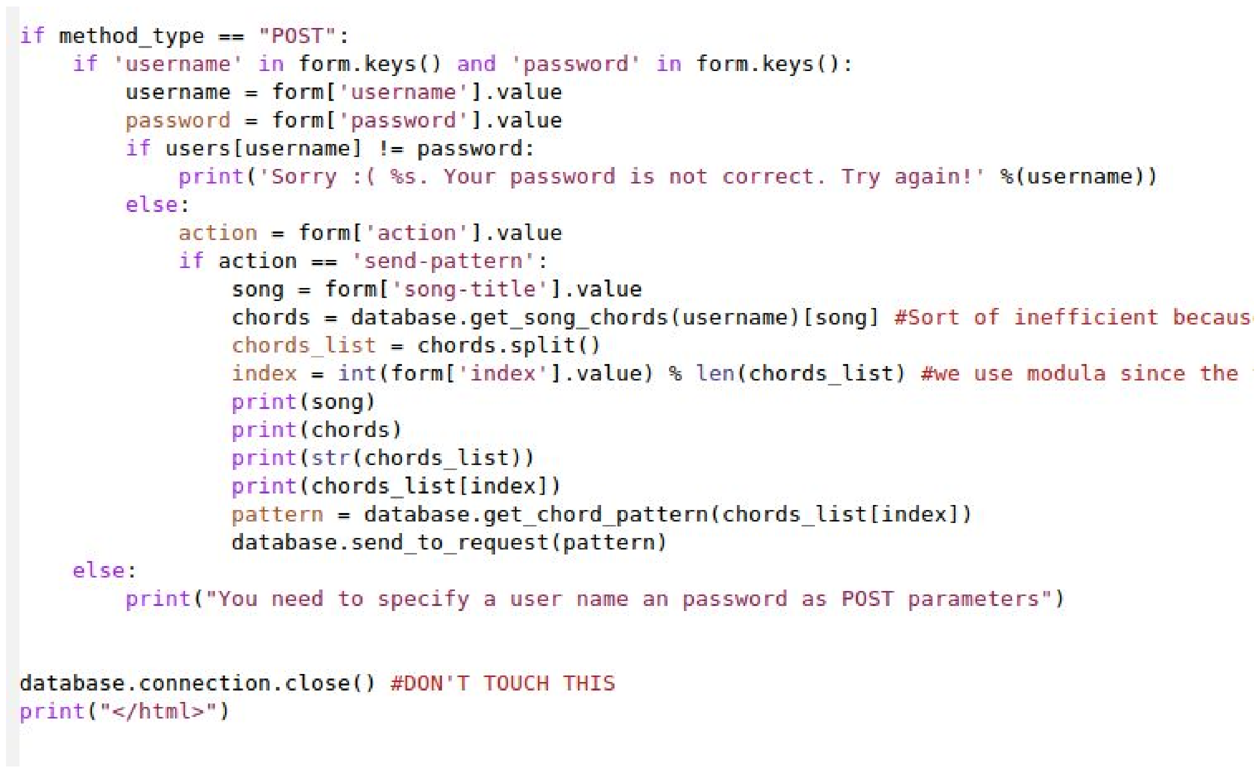
Request.py is a script that interacts with both the remote and LED hardware to get and post songs and chord patterns.



```
if method_type == "GET":
    if 'username' in form.keys() and 'password' in form.keys():
        username = form['username'].value
        password = form['password'].value
        if users[username] != password:
            print('Sorry :( %s. Your password is not correct. Try again!' %(username))
        else:
            action = form['action'].value
            if action == 'song-finder':
                songs_from_db = database.get_song_chords(username)
                print("<h1>")
                for song in songs_from_db.keys():
                    print(song)
                print("</h1>")
                print("<h2>")
                for song in songs_from_db.keys():
                    print(songs_from_db[song])
                print("</h2>")
                print("<p>"+str(len(songs_from_db))+"</p>")
            if action == 'song-tutor':
                latest = database.get_latest_request()
                pattern = latest[0]
                #check how long it's been since the last request was made
                now = datetime.datetime.now()
                time_format = '%Y-%m-%d %H:%M:%S'
                time = datetime.datetime.strptime(latest[1], time_format)
                elapsed = now - time
                if elapsed <= datetime.timedelta(minutes=5):
                    print("<p>"+pattern+"</p>")
                else:
                    print("<p>5555</p>")


    else:
        print("You need to specify a user name an password as GET parameters")
```

The first event the file looks for is if there is a GET request from one of the devices. It there is, then it will check to see if the username and password match to an entry in the master list, and if so it finds which device is asking for a get request. If it is the remote, the action will be ‘song-finder’, and the file will pull up the database for the specific songs that user has added to their songs table. The result will be returned and shown on the remote’s display.

If the action is ‘song-tutor’, it means that the LED hardware is listening for a chord. If there has been a chord posted on the requests table of the database within the past 5 minutes, requests.py will pull up the most recent posted chord pattern and return ir for the NeoPixel LED strips to show. If there is no pattern, it return the pattern that turns off the lights.

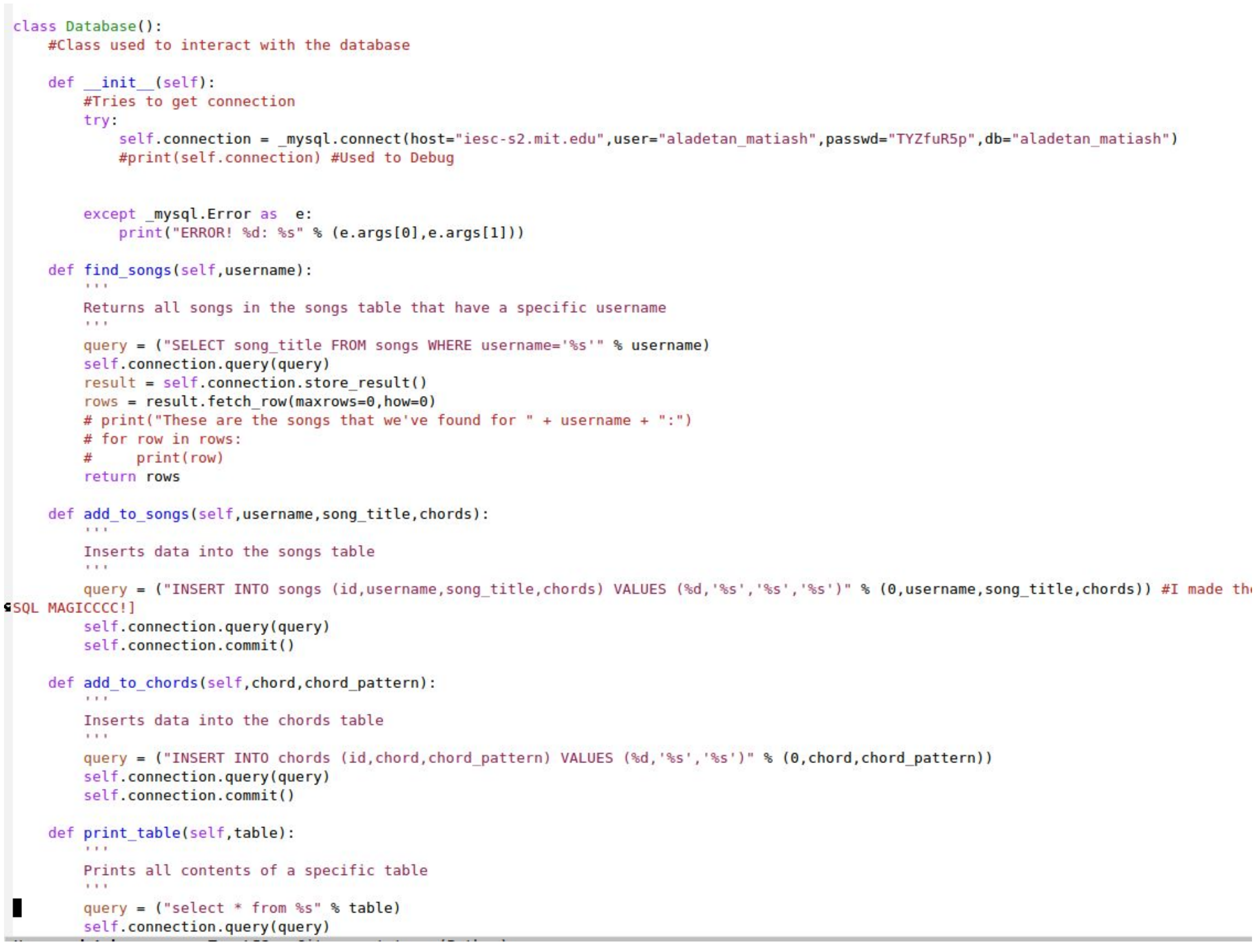


```
if method_type == "POST":
    if 'username' in form.keys() and 'password' in form.keys():
        username = form['username'].value
        password = form['password'].value
        if users[username] != password:
            print('Sorry :( %s. Your password is not correct. Try again!' %(username))
        else:
            action = form['action'].value
            if action == 'send-pattern':
                song = form['song-title'].value
                chords = database.get_song_chords(username)[song] #Sort of inefficient becaus
                chords_list = chords.split()
                index = int(form['index'].value) % len(chords_list) #we use modula since the
                print(song)
                print(chords)
                print(str(chords_list))
                print(chords_list[index])
                pattern = database.get_chord_pattern(chords_list[index])
                database.send_to_request(pattern)
    else:
        print("You need to specify a user name an password as POST parameters")


database.connection.close() #DON'T TOUCH THIS
print("</html>")
```

If the remote is doing a POST request, the file will first check to see if the username and passwords match, and if so it will get the current song selected and the index from the POST request. This is sort of inefficient since the file looks for the same song even though it could have been selected before the buttons is pressed, but it seemed to work well enough for us to keep. After looking through tables, it will retrieve the pattern to post to the requests table of the database.

database.py

This file has all to do with the tables of our database and the database itself. It has functions that add entries to the tables and functions that retrieve data from the tables. We used the _mysql library and the connection methods (especially connection's query method) to make change to the db file that stored all of our tables.



```
class Database():
    #Class used to interact with the database

    def __init__(self):
        #Tries to get connection
        try:
            self.connection = _mysql.connect(host="iesc-s2.mit.edu",user="aladetan_matiash",passwd="TYZfuR5p",db="aladetan_matiash")
            #print(self.connection) #Used to Debug


        except _mysql.Error as  e:
            print("ERROR! %d: %s" % (e.args[0],e.args[1]))

    def find_songs(self,username):
        '''
        Returns all songs in the songs table that have a specific username
        '''
        query = ("SELECT song_title FROM songs WHERE username='%s'" % username)
        self.connection.query(query)
        result = self.connection.store_result()
        rows = result.fetch_row(maxrows=0,how=0)
        # print("These are the songs that we've found for " + username + ":")
        # for row in rows:
        #     print(row)
        return rows

    def add_to_songs(self,username,song_title,chords):
        '''
        Inserts data into the songs table
        '''
        query = ("INSERT INTO songs (id,username,song_title,chords) VALUES (%d,'%s','%s','%s')" % (0,username,song_title,chords)) #I made th
SQL MAGICCCC!]
        self.connection.query(query)
        self.connection.commit()

    def add_to_chords(self,chord,chord_pattern):
        '''
        Inserts data into the chords table
        '''
        query = ("INSERT INTO chords (id,chord,chord_pattern) VALUES (%d,'%s','%s')" % (0,chord,chord_pattern))
        self.connection.query(query)
        self.connection.commit()

    def print_table(self,table):
        '''
        Prints all contents of a specific table
        '''
        query = ("select * from %s" % table)
        self.connection.query(query)
```

There are many methods in the Database class we created, and many of them are self explanatory. For example, find_songs returns all of the songs that a username has on their table. add_to_songs() adds an entry to to the songs table. add_to_chords() and print_table() are also self explanatory, so we'll move through to the next section.

```
    def reset_table(self,table):
        '''
        Deletes everything from a specific table
        '''
        query = ("DELETE from %s" % table)
        self.connection.query(query)
        self.connection.commit()

    def get_song_chords(self, username):
        '''
        Returns a dictionary of the user's songs mapped to their respective chords
        '''
        #Dictionary that hold all of the song name and their respective chords
        self.songs_to_chords = {}
        query = ("SELECT song_title, chords FROM songs WHERE username = '%s'" % username)
        self.connection.query(query)
        result = self.connection.store_result()
        rows = result.fetch_row(maxrows=0,how=0)
        for row in rows:
            if row[0] not in self.songs_to_chords:
                self.songs_to_chords[row[0].decode("utf-8")]=row[1].decode("utf-8")
        return self.songs_to_chords

    def get_chord_pattern(self,chord):
        query = ("SELECT chord_pattern FROM chords WHERE chord = '%s'" % chord)
        self.connection.query(query)
        result = self.connection.store_result()
        rows = result.fetch_row(maxrows=0,how=0)
        return rows[0][0].decode("utf-8")


    def send_to_request(self,pattern):
        query = ("INSERT INTO requests (id,code) VALUES (%d,'%s')" % (0,pattern))
        self.connection.query(query)

    def get_latest_request(self):
        query = ("SELECT * FROM requests ORDER BY id DESC LIMIT 1;")
        self.connection.query(query)
        result = self.connection.store_result()
        row = result.fetch_row(maxrows=0,how=0)
        return [row[0][1].decode("utf-8"),row[0][2]]

    def remove_song(self,username, song_title):
        query = ("DELETE FROM songs WHERE username='%s' AND song_title='%s'" % (username,song_title))
        self.connection.query(query)
```

The second section of database.py holds more important functions.

- get_song_chords() is documented above, return a dictionary of a user’s songs and each of the songs’ chords
- get_chord_pattern() looks through the chords table to extract the correct pattern
- send_to_request() posts a chord pattern to the requests table
- get_latest_request() is what we used to find the last pattern that we should show to the LED hardware
- remove_song() removes a song from the user’s song table if he or she does not want it anymore, which can be done through the web UI.

Power Management

For power management capabilities in our IoT project, we decided to use the PowerMonitor class from the power labs. From there we decided to set an interval for which the device would have to sit still with no button presses. When that interval has been surpassed, we shut down the **teensy & wifi chip**, and then clear the screen with a black rectangle. When a button is pushed, the system wakes up, and the teensy and wifi chip are turned back on, and the song menu is displayed on the screen.

For the led hardware, the LEDs turn off after five minutes if there has been no new request sent to the requests database.

Final Thoughts

We would like to thank all of the staff and lecturers for the suggestions and help they gave us during the project - this would not have been done without your support!