

# Documentation for Green Grid Advisor App

---

## Table of Contents

---

- [Documentation for Green Grid Advisor App](#)
- [Table of Contents](#)
- [Introduction](#)
- [Hardware and data collection](#)
  - [The ESP32](#)
  - [The ADS1115](#)
  - [The hull effect sensor & TRRS jack](#)
  - [Extension cord splicing](#)
  - [SCT013 + Extension cord splice](#)
  - [Programming ESP32](#)
  - [Final circuit set up](#)
  - [Procedure](#)
  - [Results](#)
- [Rest API](#)
  - [Flask introduction](#)
  - [HTTP Arguments](#)
    - [Types of HTTP requests](#)
  - [Python functions](#)
    - [Registration](#)
  - [Login](#)
  - [Change user info](#)
  - [Get info](#)
  - [Displaying TOU rates](#)
  - [Savings calculations](#)
  - [Data collection](#)
- [Kotlin](#)
  - [Login](#)
  - [Register](#)
  - [Main Menu](#)
  - [Profile](#)
  - [Scheduler](#)
  - [Analytics](#)

## Introduction

---

The intended purpose behind this app is to develop a tool to schedule and remind users to participate in electrical usage at optimal times to reduce electrical fees. This tool is designed with residential home owners in mind as users have more immediate control over electrical usage than commercial cases.

# Hardware and data collection

---

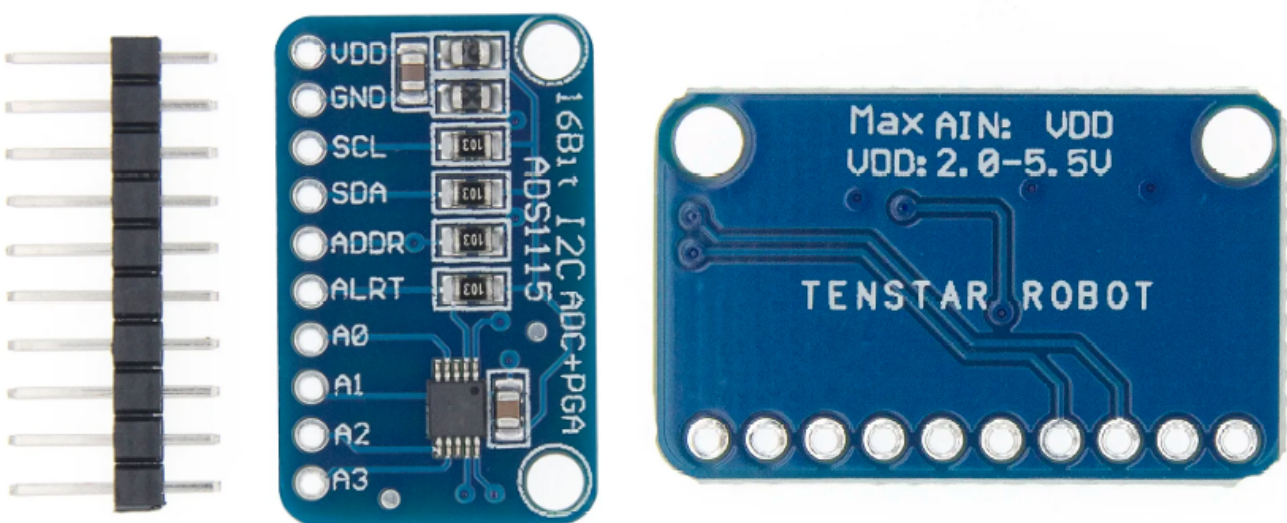
This section is dedicated to speaking about the hardware implementation for data collection on a residential household. With data collected on a residential household, our group is able to calculate and compare cost savings between fixed and time of use rates. (TOU) This hardware is unintended for app usage and is not part of the immediate design of the app.

## The ESP32



The ESP32 is employed to leverage its I2C ports for extracting data the ADS1115 and WiFi capabilities to routing the data back to our [REST API](#) server to be recorded.

## The ADS1115



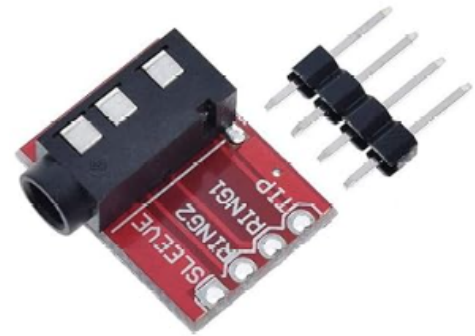
The ADS1115 an analogue to digital converter (ADC) is used to run a differential on the signal produced by our [hull effect](#) sensor. It's known that the ESP32 has onboard ADCs, but it does not have a programmable gain, unlike the ADS1115. The absence of a programmable gain resulted in immediate shifts in signals to either maximum or zero, with no legible information about the current. Switching to the ADS1115 allowed us to have a +/-5mV gain, enabling a scaled-down signal that the ADS1115 could then differentiate for us. An

I2C connection between the ADS1115 and ESP32 was made to transfer data to the ESP32 to then send HTTP requests to the REST server.

## The hull effect sensor & TRRS jack



Hull Effect  
sensor



TRRS jack

Shown above is a SCT013-000V 100A 1V Non-invasive [Split-Core Current Transformer](#) sensor. This specific model has a 100 AMP max input with a 1 volt max output (AC). The trick is to read data from the sleeve and the tip, taking the differential will give the user the ~1 volt associative amperage ratio.

Credit to [misperry](#) for having such a good video. This design was adapted from his arduino uno project for my ESP32 with slight coding modifications. The key to this project was the programmable gain pointed out.

A black power cord with a three-pronged gold plug. The cord is shown with its outer jacket stripped back, revealing three internal wires: a black wire, a green wire, and a white wire. The plug is a standard NEMA 1-15P type.

This is a 15 amp extension cord, contained within the black insulation are three other insulated wires.

**Green wire:** This is often the ground wire. It provides a path for electrical current to safely flow into the ground in the event of a fault or short circuit. It's an important safety feature in electrical systems.

**Black wire:** This is typically the hot wire, meaning it carries the electrical current from the source (e.g., an outlet) to the device (e.g., a lamp or appliance). In a standard AC electrical system, black wires are generally live or hot wires.

**White wire:** This is usually the neutral wire. It completes the electrical circuit and provides a return path for the electrical current. White wires are typically considered the grounded conductor in a system.

## SCT013 + Extension cord splice

Since black is the *hot wire*, you will take your split-core current transformer and latch to the black wire as so:



**WARNINGS:** When opening the wires, wear gloves and do not cut too deep as to damage the insulation around the three inner wires. When plugging in, please wear gloves and start with very low amperage. Say a phone charger for example.

## Programming ESP32

The following section is dedicated to helping users to program their ESP32 with template code and proper library set up.

1. Import libraries `Wire.h`, `Adafruit_ADS1X15.h`, `WiFi.h`, `HTTPClient.h` into your arduino library
2. The following is a simple set up for accessing wifi and initializing gain for the ADS1115 ADC. Adjust factor depending on what A/V ratio your current sensor produces.

```
#include <Wire.h>
#include <Adafruit_ADS1X15.h>
#include <WiFi.h>
#include <HTTPClient.h>

Adafruit_ADS1115 ads;

const char* ssid = "Router_name"; //name of router
const char* password = "abcde12345"; //password too router
const char* server = "mrzn69.pythonanywhere.com"; //REST server name
const int port = 5000; // HTTPS port

const float FACTOR = 100; // 100A/1V from the CT
const float multiplier = 0.00005;

void setup() {
  Serial.begin(9600);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  ads.setGain(GAIN_FOUR); // +/- 1.024V 1bit = 0.5mV
  ads.begin();
}
```

3. This chunk of code is responsible for printing, looping function calls and calculating current in IRMS.

```
void printMeasure(String prefix, float value, String postfix) {
  Serial.print(prefix);
  Serial.print(value, 3);
  Serial.println(postfix);
}

void loop() {
  float currentRMS = getcurrent();
  printMeasure("Irms: ", currentRMS, "A");

  // Send the value to the server
  sendToServer(currentRMS);
}
```



```

delay(10000); // Adjust this delay based on your needs
}

float getcurrent() {
float voltage;
float current;
float sum = 0;
long time_check = millis();
int counter = 0;

while (millis() - time_check < 1000) {
    voltage = ads.readADC_Differential_0_1() * multiplier;
    current = voltage * FACTOR;

    sum += sq(current);
    counter = counter + 1;
}

current = sqrt(sum / counter);
return current;
}

```

4. This remaining portion of the arduino code is to advise user on what parameters to change if a new CSV needs to be appended too. Notice `csv_name = calvin_test` this parameter is in charge of what csv a particular ESP32 is appending too. Change parameter as the REST server done with flask will take that name and search for a csv of that name or create a new csv to start appending too. The neat part about the ESP32, once you encode it, you can unplug and begin usage without having to re-program (UNLESS YOU WANT TO CHANGE CSV NAME YOU'RE APPENDING TOO).

```

void sendToServer(float value) {
HTTPClient http;

// Specify the target server and resource
String url = "https://mrazn69.pythonanywhere.com/data_collection?
csv_name=calvin_test&value_to_append=" + String(value);

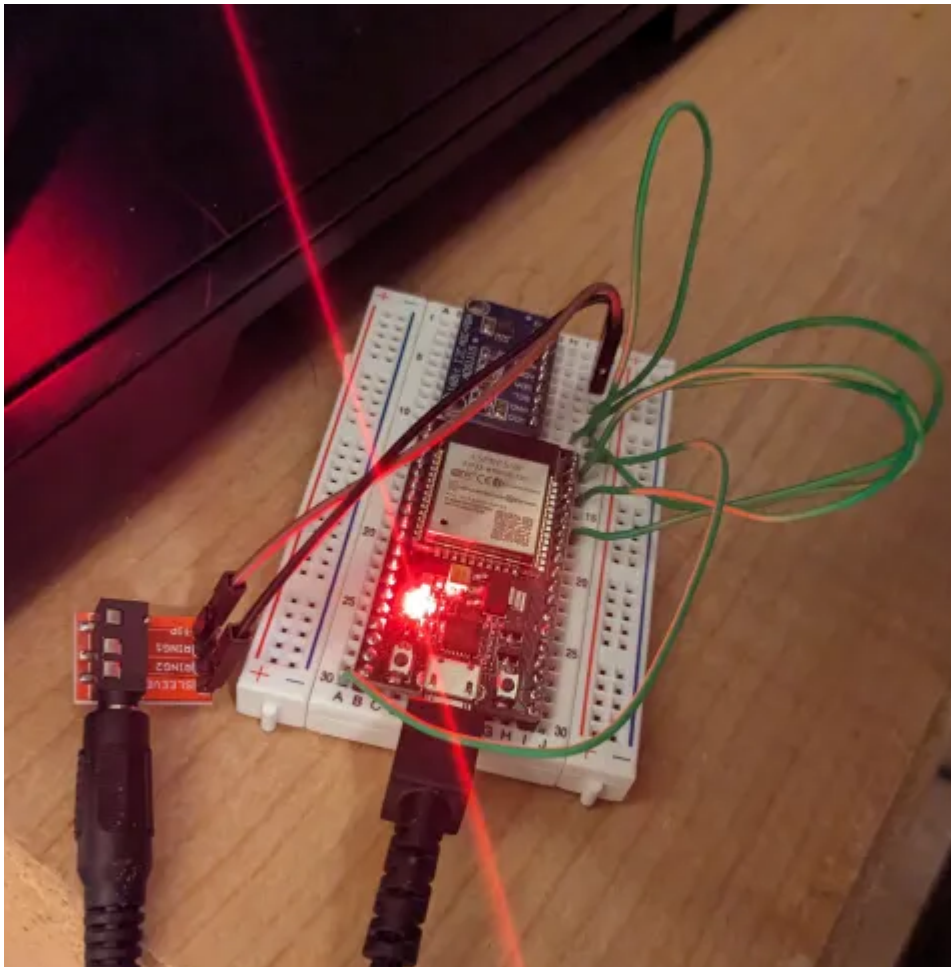
// Make the HTTP GET request
http.begin(url);
int httpCode = http.GET();

// Check for a successful response
if (httpCode > 0) {
    if (httpCode == HTTP_CODE_OK) {
        String payload = http.getString();
        Serial.println("HTTP Request successful. Response:");
        Serial.println(payload);
    } else {
        Serial.println("HTTP Request failed with error code: " + String(httpCode));
    }
}
}

```

```
} else {  
    Serial.println("Unable to connect to the server");  
}  
  
// Close the connection  
http.end();  
}
```

## Final circuit set up



This ESP32 + ADS1115 +

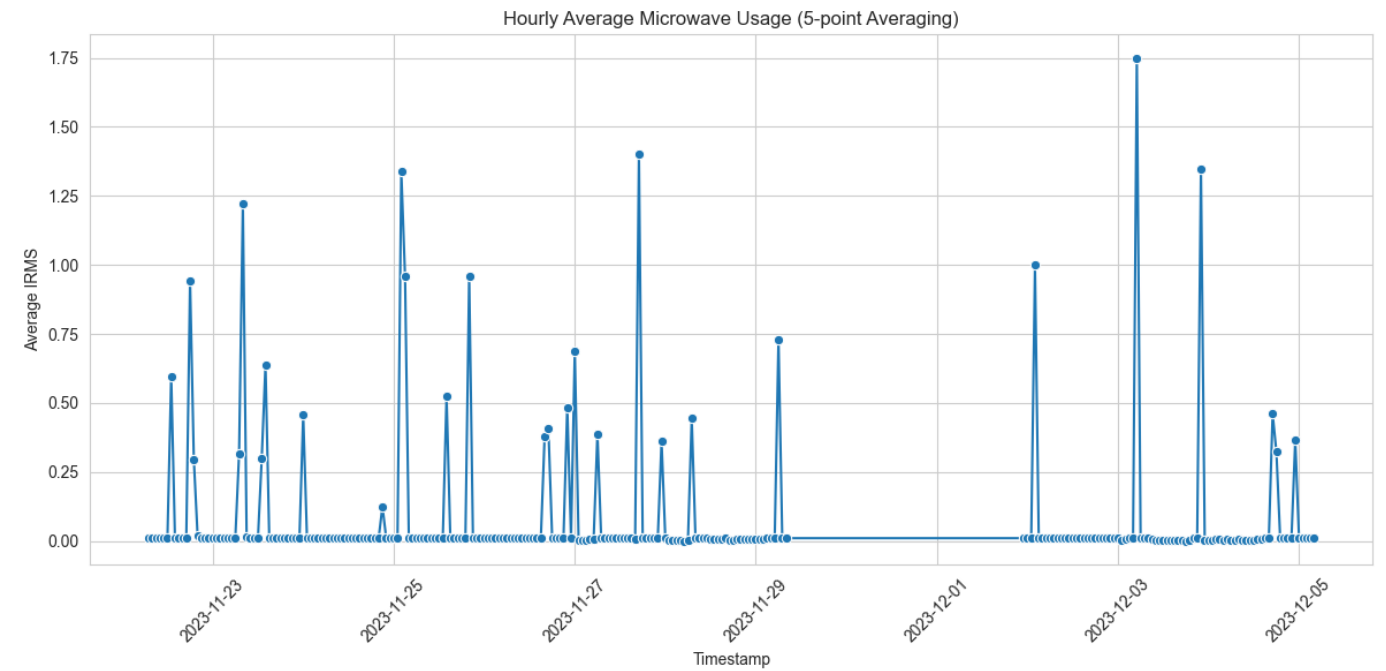
SCT013 100 A: 1V split core current transformer is capable of handling 100A measurements. Though the average wall outlet in a house is 15A, the only circuit in a house that would have 15+ A is the dryer port. You'll need a specialized/split extension cable to measure off of.

## Procedure

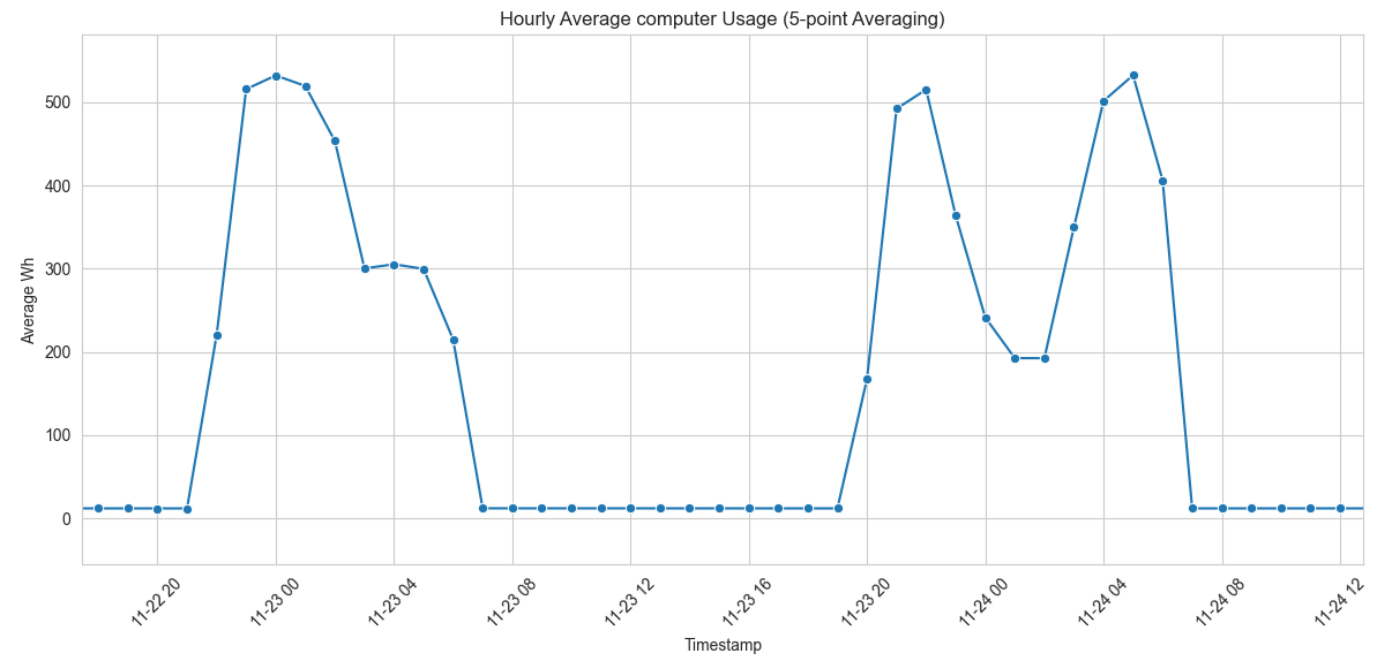
The CSV data was collected over the course of 25+ days for four power intensive appliances found in a typical residential home. The energy intensive appliances in question are a [desktop](#), [washing machine](#), [dryer](#) and [microwave](#). Using the non-invasive nature of the IRMS data collection device created by Green Grid advisor, simply using the male to outlet and using the female to appliances, a readable point is now established. All that is needed now (assuming that the hardware device is already assembled properly) is to assign a CSV name to the ESP32 and begin collecting data. System will return IRMS value every 10 seconds, system will run as long as desired.

## Results





The results above depict the IRMS of a microwave inside Calvin's house. The points are averaged to depict hourly usage of current per day.



These results are a prototype graphic to see wattage usage based on a 120V 15 A outlet. It's clear when computer is under load based on inflection points.

# Rest API

This section is for explaining the REST API concepts used with **Python Anywhere** and **flask**.

**Simplicity and Ease of Use:** REST APIs are designed to be simple and easy to use. They typically use standard HTTP methods (GET, POST, PUT, DELETE) and are stateless, which means each request from a client contains all the information needed to understand and process the request.

**Scalability:** RESTful architectures are scalable, making them suitable for a wide range of applications, from small-scale to large-scale systems. The statelessness of REST allows for easy distribution and load balancing of

services.

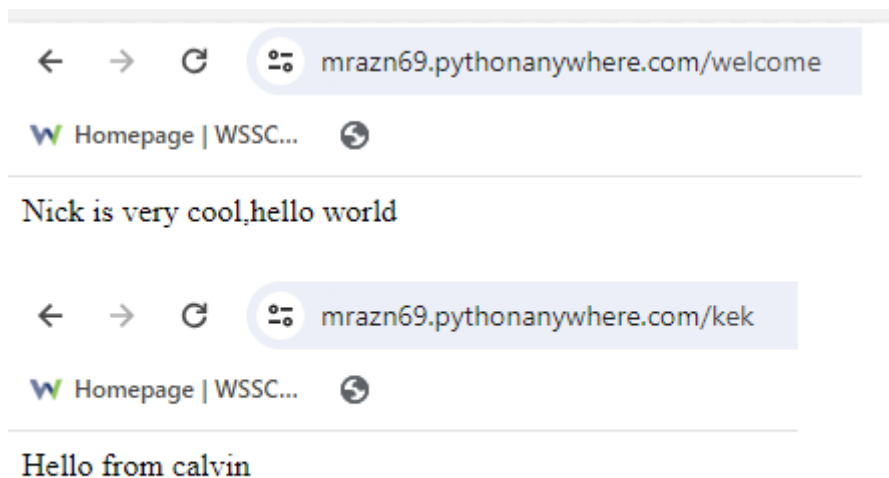
**Uniform Interface:** REST APIs have a uniform and standardized interface. This uniformity simplifies interactions between clients and servers, making it easier for developers to understand and work with different APIs.

## Flask introduction

The following code is simple snippet, intended use is to give users the idea of how to call certain functions in flask via HTTP request parameters.

```
from flask import Flask, request, jsonify
reply = []
app = Flask(__name__)

@app.route('/welcome')
def hello_world():
    return 'Nick is very cool,hello world'
@app.route('/kek')
def Calvin():
    return 'Hello from calvin'
```



Correspondant output:

Each HTTP request uses <https://mrzn69.pythonanywhere.com/> to access a specific server. Followed by [/function](#). This allows for specific functions to be easily called via HTTP request.

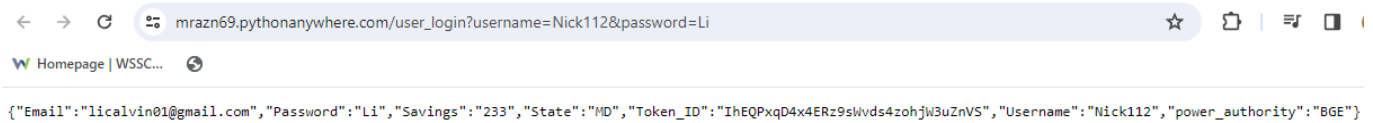
## HTTP Arguments

Parameters/arguments can also be passed via HTTP link. This section will show an example.

```
@app.route('/user_login', methods=['GET'])
def user_login():
    username = request.args.get('username', 'World')
    password = request.args.get('password', 'World')
```

```
reply = simple_login.login(username,password)
return reply
```

Correspondant output:



```
{ "Email": "licalvin01@gmail.com", "Password": "Li", "Savings": "233", "State": "MD", "Token_ID": "IhEQPxqD4x4ERz9slwds4zohjW3uZnVS", "Username": "Nick112", "power_authority": "BGE" }
```

The function `user_login` was selected, then a `?` is applied to begin passing parameters. Between each variable after assignment requires a `&`. Notice how username was passed as `username=Nick112&password=Li`. It runs the function `simple_login.login(username,password)` with the parameters past in the HTTP request. The reply is the response from python anywhere after running the function.

## Types of HTTP requests

**GET:** Used to request data from a specified resource. It is a safe and idempotent operation, meaning it should not have side effects on the server, and multiple identical requests should produce the same result as a single request.

**POST:** Used to submit data to be processed to a specified resource. It is not idempotent, meaning multiple identical requests may have different effects. It is commonly used for creating new resources on the server or submitting form data.

**Debugging tips:** The variable names you are passing are very specific. For example `Username` and `username` are not the same. Getting a `internal server error` means the function was called but there was some error in your function. It is not the same as calling a function that doesn't exist. It will simply say "Not found".

## Python functions

This section will explain the custom Python functions tailored to this capstone. Including login, database manipulation, ESP32 data collection and savings calculations.

### Registration

Like many apps, basic information is required to uniquely identify users and to contact them. The following code is the route request for registering users.

```
@app.route('/register', methods=['GET']) #simple request,
https://mrzn69.pythonanywhere.com/hello?name=Tiglao
def register():
    # username, password, savings, power_authority, State, Email
    username = request.args.get('username', 'World')
    password = request.args.get('password', 'World')
    savings = request.args.get('savings', 'World')
    power_authority = request.args.get('PA', 'World')
    State = request.args.get('State', 'World')
    Email = request.args.get('Email', 'World')
    reply =
```

```
simple_login.register(username,password,savings,power_authority,State,Email)
    return reply
```

This function requires six unique parameters to operate correctly. Upon successful registration, a JSON payload containing the user's unique 32-character alphanumeric ID is returned, along with other entered details, confirming the registration. Otherwise, an internal server error will occur.

The function being ran in the routed registration function is `simple_login.register()`. The following is the code for the function:

```
def read_user_data():
    try:
        with open('mysite/U_P_settings.csv', 'r') as file:
            reader = csv.DictReader(file)
            user_data = [row for row in reader]
    except FileNotFoundError:
        user_data = []

    return user_data

def ID_check(ID):
    user_data = read_user_data()

    for user in user_data:
        if user['Token_ID'] == ID:
            return user

    return "No ID"

def register(username, password, savings, power_authority, State, Email):
    try:
        user_data = read_user_data()

        # Check if the username already exists
        if any(user['Username'] == username for user in user_data):
            return "Username already exists"

        unique_token = generate_unique_token()

        # Check if the generated Token_ID is unique
        while any(user['Token_ID'] == unique_token for user in user_data):
            unique_token = generate_unique_token()

        user_data.append({'Username': username, 'Password': password, 'Savings':
savings, 'Token_ID': unique_token, 'power_authority': power_authority, 'State':
State, 'Email': Email})
        write_result = write_user_data(user_data)

        if write_result.startswith("Error"):
            return write_result # Return the write error message
        else:
```

```

        return f"Registration successful for {username} with Token_ID:
{unique_token}"
    except FileNotFoundError:
        return "Error: File not found"
    except json.JSONDecodeError:
        return "Error: Failed to decode JSON data"
    except KeyError as e:
        return f"Error: Key not found in JSON data - {e}"
    except Exception as e:
        return f"Internal server error during registration: {e}"

def generate_unique_token():
    characters = string.ascii_letters + string.digits
    unique_token = ''.join(secrets.choice(characters) for _ in range(32))
    return unique_token

```

The register function takes in 6 arguments and appends those details into a CSV that acts as a database. If appending and unique token generation is performed then a string in a JSON format is returned to the HTTP page of the sender. The registration function requires `generate_unique_token()` to generate a 32 character alpha numeric. This is for basic security and a method to avoid constantly passing username and password later on. This function also protects against same token generation and username, but does not protect against same email redundancies. This will be added later. `ID_Check()` was created to be able to seek an ID for internal server usage instead of using user login information.

The read user data is to get the details from the CSV file `mysite/U_P_settings.csv`. Where all user data is stored and parsed in the event something needs to be changed or read.

## Login

This section is dedicated to explaining the login implementation function.

```

def login(username, password):
    user_data = read_user_data()

    for user in user_data:
        if user['Username'] == username and user['Password'] == password:
            return user

    return "Login failed. Please check your username and password."

```

This code checks username and password inputted against the CSV database for existing users. Should match occur return the 32 character alpha numeric to user. Now the user has base functionality of the app under their own individual profiles to track their TOU and savings.

Below is the flask call for the login page with necessary parameters.

```
@app.route('/user_login', methods=['GET'])
def user_login():
    username = request.args.get('username', 'World')
    password = request.args.get('password', 'World')
    reply = simple_login.login(username,password)
    return reply
```

## Change user info

```
def change_user_stuffs(ID, category, value):
    user_data = read_user_data()

    # Check if the specified ID exists in the CSV file
    if not any(user['Token_ID'] == ID for user in user_data):
        return f"Error: User with Token_ID '{ID}' not found in CSV file."

    # Check if the category exists in the CSV file
    if not all(category in user for user in user_data):
        return f"Error: Category '{category}' not found in CSV file."

    for user in user_data:
        if user['Token_ID'] == ID:
            # Check if the category exists for the specific user
            if category in user:
                user[category] = value
                break
            else:
                return f"Error: Category '{category}' not found for the user with
Token_ID '{ID}'."

    write_result = write_user_data(user_data)
    return write_result # Return the write result
```

This code allows for users to change certain parameters. For example username, password, state and power authority so long as the user passes a unique ID that matches in the CSV database to a profile. This was mainly implemented to let users change their state and power authority to allow for app use anywhere.

Below is the flask call associated with the change user function when HTTP request is routed.

```
@app.route('/change_stuff', methods=['GET'])
def change_stuff():
    ID = request.args.get('ID', 'World')
    category = request.args.get('category', 'World')
    value = request.args.get('value', 'World')
    reply = simple_login.change_user_stuffs(ID,category,value)
    return reply
```



## Get info

This app's main feature is its capability to retrieve state and power authority information, which is then employed to guide users on optimizing their electricity usage. This effectively eliminates the burden of remembering to actively engage in energy management.

There is a JSON file that looks as so:

-put json file here

There is a function called `get_info` within `simple_login.py` that reads a hand written JSON file organized by state and power authority keys. This is important to pass to the users as these rates will ensure usability of the app dependent upon user location.

```
from datetime import datetime

def get_info(PA, State):
    file_name = "mysite/json_electric.txt"

    try:
        with open(file_name, "r") as file:
            electricity_rates = json.load(file)
            return_data = electricity_rates[f"{PA}", {State}]

            winter_interval = return_data['Winter']['Interval']
            winter_start_month, winter_end_month = winter_interval.split(" to ")

            summer_interval = return_data['Summer']['Interval']
            summer_start_month, summer_end_month = summer_interval.split(" to ")

            winter_start_num = m2n.get(winter_start_month)
            winter_end_num = m2n.get(winter_end_month)

            current_datetime = datetime.now()
            current_month = current_datetime.month

            if winter_start_num <= current_month <= 12 or 1 <= current_month <=
winter_end_num:
                return "Season is winter!\n{}\n{}".format(return_data['Winter']['Peak
Hours (Morning)'], return_data['Winter']['Peak Hours (Afternoon)'])
            else:
                return "Season is not winter!\n{}\n{}".format(return_data['Summer']
['Peak Hours (Morning)'], return_data['Summer']['Peak Hours (Afternoon)'])

    except FileNotFoundError:
        return f"Error: File {file_name} not found."
    except json.JSONDecodeError:
        return f"Error: Failed to decode JSON in {file_name}."
    except KeyError as e:
        return f"Error: Key not found in JSON data - {e}"
    except Exception as e:
        return f"An unexpected error occurred: {e}"
```

Within this function, it takes two parameters, state and power authority. It utilizes key phrases in a Python [dictionary](#) format to access the values within the JSON file. 'Summer' and 'Winter' are key phrases paired with 'Interval'. The libraries [datetime](#) comes into play assisting me in deciding to choose either winter or summer rates.

Below is the flask route call to have HTTP requests get information:

```
@app.route('/TOU_getinfo', methods=['GET'])
def TOU_getinfo():
    power_authority = request.args.get('power_authority', 'World')
    State = request.args.get('State', 'World')
    reply = simple_login.get_info(power_authority, State)
    return reply
```

## Displaying TOU rates

```
@app.route('/img')
def img():
    peak_hours = ['6:00 AM - 9:00 AM', '5:00 PM - 9:00 PM'] # need to extract
individual numbers
    peak_rate = 0.3
    non_peak_rate = 0.1
    flat_rate = 0.14 # Fixed rate
    # img =
simple_login.generate_rate_graph(peak_hours, peak_rate, non_peak_rate, flat_rate)
    # return img
    # fig, ax = plt.subplots()
    # ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
    time = np.linspace(0, 24, 1000)

    # Create arrays to hold the rate values for each time point
    rate_values_tou = np.zeros_like(time)
    rate_values_flat = np.zeros_like(time)

    # Assign the TOU rates based on peak and non-peak hours
    for i, hour in enumerate(time):
        if (6 <= hour < 9) or (17 <= hour < 21):
            rate_values_tou[i] = peak_rate
        else:
            rate_values_tou[i] = non_peak_rate

    # Assign the flat rate for the entire 24-hour period
    rate_values_flat.fill(flat_rate)

    # Create the plot
    plt.figure(figsize=(12, 6))

    # Plot TOU rates with blue color
```

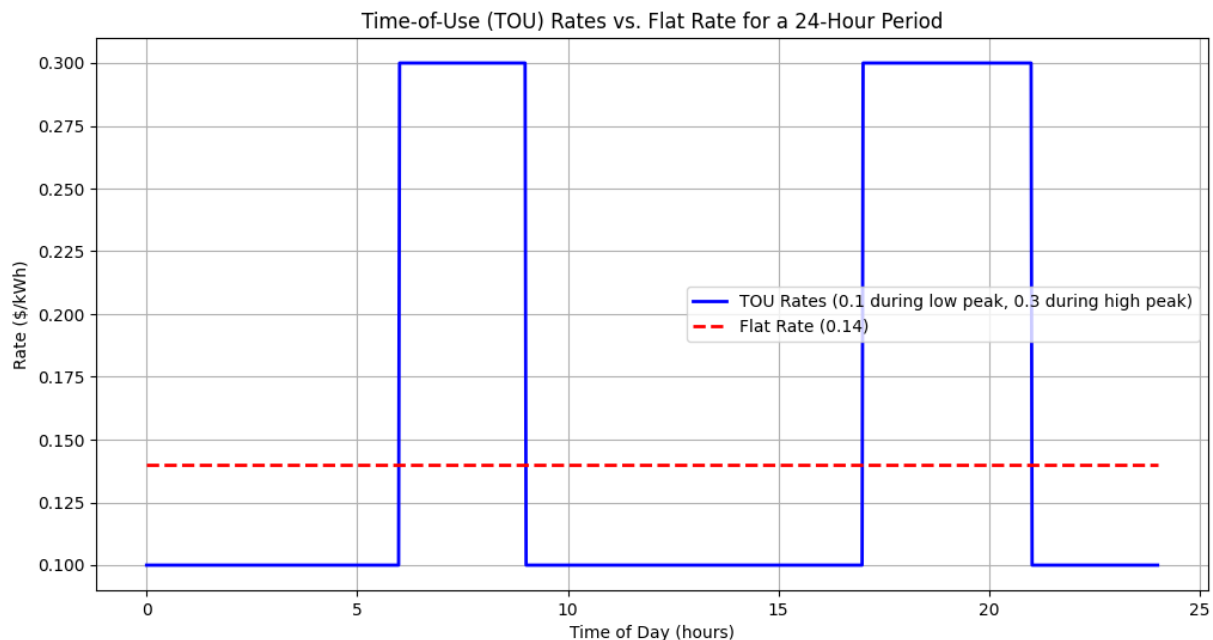
```
plt.plot(time, rate_values_tou, color='blue', linewidth=2, label=f"TOU Rates
({non_peak_rate} during low peak, {peak_rate} during high peak)")

# Plot the flat rate with red color
plt.plot(time, rate_values_flat, color='red', linestyle='--', linewidth=2,
label=f"Flat Rate ({flat_rate})")

plt.title('Time-of-Use (TOU) Rates vs. Flat Rate for a 24-Hour Period')
plt.xlabel('Time of Day (hours)')
plt.ylabel('Rate ($/kWh)')
plt.grid(True)
plt.legend()
img_buffer = BytesIO()
plt.savefig(img_buffer, format='png')
img_buffer.seek(0)

# Return the image as a response
return send_file(img_buffer, mimetype='image/png'), "hello"
```

This is just a prototype/proof of concept that an image of time-of use rates can be sent via HTTP request. Due to time constraints, it is unlikely I will be able to advance this flask function further than a proof of concept. This was done in light of Dr.Tiglao's feedback that if Kotlin app cannot display data, then why not just have it display an image instead. Here is the reply to his suggestion. This function was tested by directly doing the algorithms in the state.



## Savings calculations

```
def calculate_savings(ID, duration, task):
    user = ID_check(ID)
    try:
        str_rates = get_info(user['power_authority'], user['State'])
        data_list = str_rates.split(',')
    except:
```

```

rates = data_list[-3:]

task_wattage = 0.0

if task == 'Dryer':
    task_wattage = 1.600
elif task == 'Computer':
    task_wattage = 0.24535
elif task == 'Microwave':
    task_wattage = 0.6000
elif task == 'Washing':
    task_wattage = 0.14
else:
    return "Error: Invalid task type"

Low = float(rates[0]) * duration
fixed = float(rates[2]) * duration
# return "hello"
savings = (fixed - Low)*task_wattage # Should be a positive number
current_savings = float(user['Savings'])
new_savings = round(current_savings + savings, 2)
change_user_stuffs(ID, "Savings", new_savings)
user = ID_check(ID)
return ','.join([user['Username'], user['Password'], str(user['Savings']),
user['Token_ID'], user['power_authority'], user['State'], user['Email']])

except FileNotFoundError:
    return "Error: File not found"

```

When writing this function, there was a realization that the savings is also heavily dependent on wattage usage. Utilizing the hardware for data collection could get better accuracy than averaging power usage per appliance. This will be an added feature in the future. The `task_wattage` values based on `task` was hand calculated based on data collected from IRMS sensors.

## Data collection

Mentioned earlier in the hardware portion of this documentation, IRMS data was needed in order to prove potential savings with TOU rates over fixed rates.

```

import csv
import os
from datetime import datetime

def append_value_to_csv(base_filename, value_to_append, timestamp=None):
    try:
        # Create the 'csvs' folder if it doesn't exist
        csvs_folder = 'csvs'
        os.makedirs(csvs_folder, exist_ok=True)

        # Add '.csv' extension to the base filename
        csv_filename = f"{base_filename}.csv"

```

```

# Create the full path for the CSV file
csv_path = os.path.join(csvs_folder, csv_filename)

# Check if the CSV file exists
if not os.path.isfile(csv_path):
    # If the file doesn't exist, create it with a header
    with open(csv_path, 'w', newline='') as csvfile:
        csv_writer = csv.writer(csvfile)
        csv_writer.writerow(['IRMS', 'Timestamp']) # Header

# Open the CSV file in append mode
with open(csv_path, 'a', newline='') as csvfile:
    # Create a CSV writer
    csv_writer = csv.writer(csvfile)

    # Use current timestamp if none provided
    timestamp = timestamp or datetime.now().strftime('%Y-%m-%d %H:%M:%S')

    # Append the value and timestamp to each row in the CSV
    csv_writer.writerow([value_to_append, timestamp])

    return f"Value '{value_to_append}' with timestamp '{timestamp}' appended to '{csv_path}'"
except Exception as e:
    return f"Error: {e}"

```

This function takes two inputs, CSV name and IRMS value. The python function being called has redundancy checks that make this function versatile. Should a CSV not exist in a folder then it will generate a first instance and append the IRMS value to that CSV. Else if the CSV exists then it keeps appending.

There is data flow every 10 seconds from the ESP32s measuring IRMS. Every point was tracked except for when the ESP32 timed themselves out for long periods of usage.

Below is the routing function for flask integration for data collection.

```

@app.route('/data_collection', methods=['GET'])
def data_collection():
    # Assuming data is sent in the request body as form data
    csv_name = request.args.get('csv_name', 'World')
    value_to_append = request.args.get('value_to_append', 'World')
    reply = data_collect.append_value_to_csv(csv_name, value_to_append)
    return reply

if __name__ == '__main__':
    app.run(debug=True, port=5000)

```

It is crucial to set port=5000 for the http server and ESP32; this will have the server and device communicating on the same port number. I struggled with this early on, don't make the mistake.

# Kotlin

---

This section is dedicated to the documentation of the Android app developed in the Kotlin programming language. The app has multiple layers that include a login page, register page, analytics page, profile page, and scheduling page. It is important to note that programming Kotlin requires you to design/code a user interface through eXtensible Markup Language (XML) which is separate from the activity main code that is responsible for the actual logic that goes on. There exists XML code for each page designed, but only the output of that code will be shown as images alongside code for the actual logic to save space and time.

An additional note to be made is that the app utilizes HTTP GET and POST requests to send and receive data. This was done utilizing the [OkHttp](#) API built for handling HTTP requests in Kotlin.

## Login Page

Upon opening the app, the main screen the user encounters is the login page which prompts for username and password should they have an account already. If an existing account does not exist, they can register for one. The image below is an example of the main login page.



In order for the login page to work as every other login page does, the user must insert a username and password into the EditText fields provided and press the "login" button. Once that was pressed, an HTTP GET



request will be sent to the server that stores all user information. Should that username and password combination exist, the screen will proceed to the main menu page. The code for the logic is shown below.

```
package com.example.prototype

import android.content.Intent
import android.os.AsyncTask
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity
import okhttp3.OkHttpClient
import okhttp3.Request

class MainActivity : AppCompatActivity() {

    private lateinit var userNameEditText: EditText
    private lateinit var passwordEditText: EditText
    private lateinit var loginButton: Button
    private lateinit var registerTextView: TextView

    lateinit var invisEmail:TextView
    lateinit var invisPassword:TextView
    lateinit var invisSavings:TextView
    lateinit var invisState:TextView
    lateinit var invisToken:TextView
    lateinit var invisUsername:TextView
    lateinit var invisPA:TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        userNameEditText = findViewById(R.id.emailEditText)
        passwordEditText = findViewById(R.id.passwordEditText)
        loginButton = findViewById(R.id.loginButton)
        registerTextView = findViewById(R.id.registerTextView)

        invisEmail = findViewById(R.id.invisEmail)
        invisPassword = findViewById(R.id.invisPassword)
        invisSavings = findViewById(R.id.invisSavings)
        invisState = findViewById(R.id.invisState)
        invisToken = findViewById(R.id.invisToken)
        invisUsername = findViewById(R.id.invisUsername)
        invisPA = findViewById(R.id.invisPA)

        loginButton.setOnClickListener {
            val usernameTemp = userNameEditText.text.toString()
            val passwordTemp = passwordEditText.text.toString()
            val url = "https://mrzn69.pythonanywhere.com/user_login?
username=$usernameTemp&password=$passwordTemp"
```

```

        MyAsyncTask().execute(url)
    }

    registerTextView.setOnClickListener {
        var intent = Intent(this@MainActivity, RegisterPage::class.java)
        startActivity(intent)
    }
}

private inner class MyAsyncTask : AsyncTask<String, Void, String>() {

    override fun doInBackground(vararg params: String): String {
        try {
            val url = params[0]
            val client = OkHttpClient()

            // Create a request
            val request = Request.Builder()
                .url(url)
                .build()

            // Execute the request and get the response
            val response = client.newCall(request).execute()

            if (!response.isSuccessful) {
                // Handle non-successful response (e.g., log or return an
error message)
                return "Error: ${response.code}"
            }

            return response.body?.string() ?: ""
        } catch (e: Exception) {
            // Handle any exceptions that might occur during the network
operation
            e.printStackTrace()
            return "Error: ${e.message}"
        }
    }

    override fun onPostExecute(result: String) {
        super.onPostExecute(result)

        // Parse the response string
        val parsedStrings = parseResponse(result)

        // Now you can use parsedStrings for further processing
        val username = parsedStrings[0]
        val password = parsedStrings[1]
        val savings = parsedStrings[2]
        val token = parsedStrings[3]
        val powerAuthority = parsedStrings[4]
        val state = parsedStrings[5]
    }
}

```

```
        val email = parsedStrings[6]

        // Do something with the parsed strings (e.g., update UI elements)
        invisEmail.text = email
        invisPassword.text = password
        invisSavings.text = savings
        invisState.text = state
        invisToken.text = token
        invisUsername.text = username
        invisPA.text = powerAuthority

        var tempEmail:String = invisEmail.text.toString()
        var tempPassword:String = invisPassword.text.toString()
        var tempSavings:String = invisSavings.text.toString()
        var tempState:String = invisState.text.toString()
        var tempToken:String = invisToken.text.toString()
        var tempUsername:String = invisUsername.text.toString()
        var tempPA:String = invisPA.text.toString()

        var intent = Intent(this@MainActivity, MainMenuV2::class.java)

        intent.putExtra("emailPR", tempEmail)
        intent.putExtra("passwordPR", tempPassword)
        intent.putExtra("savingsPR", tempSavings)
        intent.putExtra("statePR", tempState)
        intent.putExtra("tokenPR", tempToken)
        intent.putExtra("usernamePR", tempUsername)
        intent.putExtra("PAPR", tempPA)

        startActivity(intent)
    }

    private fun parseResponse(response: String): List<String> {
        val lines = response.split(",")

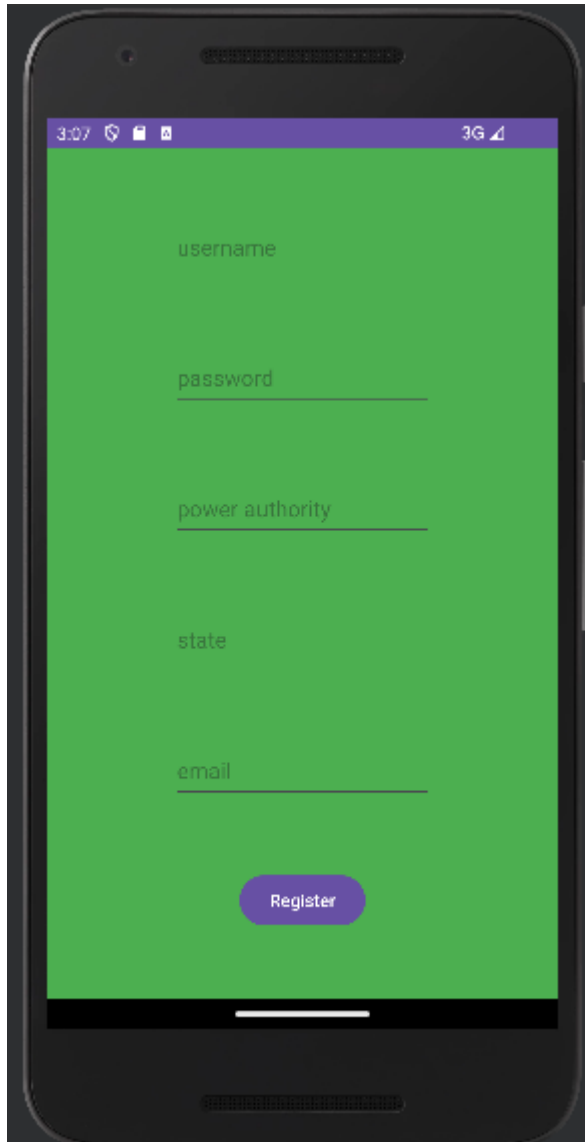
        // Assuming the response structure is consistent
        val username = lines[0]
        val password = lines[1]
        val savings = lines[2]
        val token = lines[3]
        val powerAuthority = lines[4]
        val state = lines[5]
        val email = lines[6]

        return listOf(username, password, savings, token, powerAuthority,
state, email)
    }
}
```

The main portion of the code that makes the GET request is when `MyAsyncTask().execute(url)` is called with a link as a parameter. The link made as parameter was specifically made so that when the GET request is made, it would take the user input of username and password and embed it into the link. Once the link is executed, the request is made and a response body is returned. Once that occurs, parsing happens to separate the information from the server which can then be used for the profile page.

## Register Page

Should the user not own an account already, they may press the prompt on the login page that leads them to the register page which is shown below.



Once the user enters the necessary information to make an account, they can press register and a confirmation page will appear. From there, the user can backtrack to the login page and login using the newly registered account information.

```
package com.example.prototype

import android.content.Intent
import android.net.Uri
import androidx.appcompat.app.AppCompatActivity
```

```
import android.os.Bundle
import android.widget.Button
import android.widget.EditText

class RegisterPage : AppCompatActivity() {

    lateinit var username: EditText
    lateinit var password: EditText
    lateinit var powerAuthority: EditText
    lateinit var state: EditText
    lateinit var email: EditText
    lateinit var send: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_register_page)

        username = findViewById(R.id.username)
        password = findViewById(R.id.password)
        powerAuthority = findViewById(R.id.powerAuthority)
        state = findViewById(R.id.state)
        email = findViewById(R.id.email)
        send = findViewById(R.id.button)

        send.setOnClickListener {
            // Get user-inputted data from the EditText fields
            val usernameTemp = username.text.toString()
            val passwordTemp = password.text.toString()
            val powerAuthorityTemp = powerAuthority.text.toString()
            val stateTemp = state.text.toString()
            val emailTemp = email.text.toString()

            var intent = Intent(this@RegisterPage, ConfirmationPage::class.java)

            intent.putExtra("username", usernameTemp)
            intent.putExtra("password", passwordTemp)
            intent.putExtra("PA", powerAuthorityTemp)
            intent.putExtra("state", stateTemp)
            intent.putExtra("email", emailTemp)

            startActivity(intent)
        }
    }
}
```

The code above is the logic for obtaining the user input and temporarily storing it in local variables. From there, the data is sent to the next page/activity through [Intent](#) where the actual POST request occurs. The code below is the logic necessary for the POST request to go through with the corresponding screen being a small confirmation page that shows registration as successful. The code below takes the data from the

previous page using the same codenames to access the locked data, assigns it to a temporary variable, and plugs it into a link for execution of the request.

```
package com.example.prototype

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.webkit.WebView
import android.webkit.WebViewClient

class ConfirmationPage : AppCompatActivity() {

    lateinit var webView: WebView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_confirmation_page)

        webView = findViewById(R.id.web)

        var userName: String = intent.getStringExtra("username").toString()
        var password: String = intent.getStringExtra("password").toString()
        var powerAuthority: String = intent.getStringExtra("PA").toString()
        var state: String = intent.getStringExtra("state").toString()
        var email: String = intent.getStringExtra("email").toString()

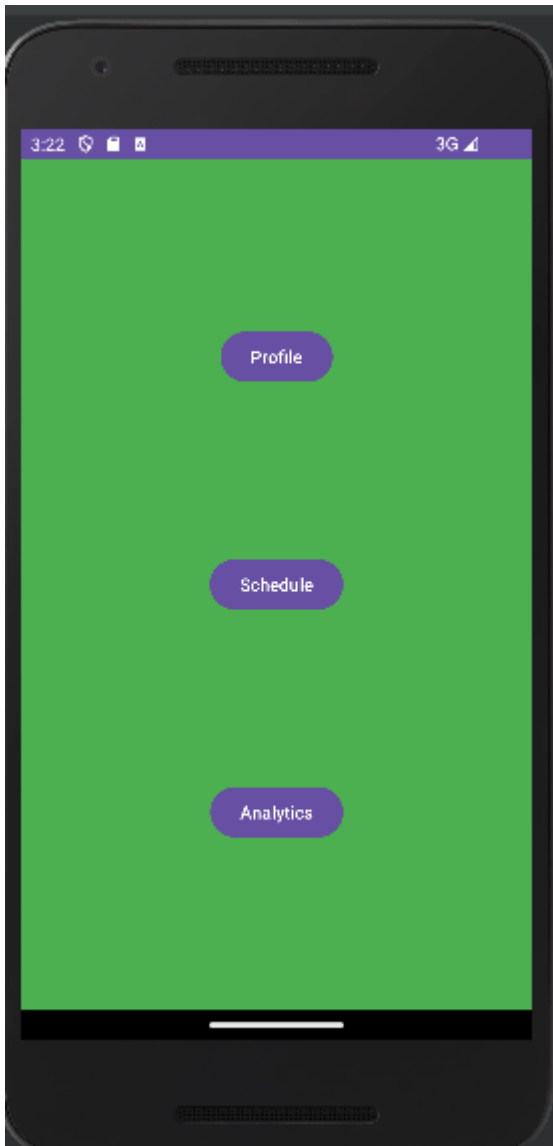
        webView.webViewClient = WebViewClient()
        webView.loadUrl("https://mrzn69.pythonanywhere.com/register?
username=$userName&password=$password&savings=233&PA=$powerAuthority&State=$state&
Email=$email")

    }
}
```

## Main Menu

The main menu consists of a generic screen with buttons that lead to three different activities. The image below is a depiction of the main menu.





The code below is the simple logic that allows for transition to the three other activities using Intent should the user click on one of the three buttons.

```
package com.example.prototype

import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button

class MainMenuV2 : AppCompatActivity() {

    lateinit var profileButton: Button
    lateinit var schedButton: Button
    lateinit var analyticsButton: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main_menu_v2)

        profileButton = findViewById(R.id.profileButton)
```

```
        schedButton = findViewById(R.id.schedButton)
        analyticsButton = findViewById(R.id.analyticsButton)

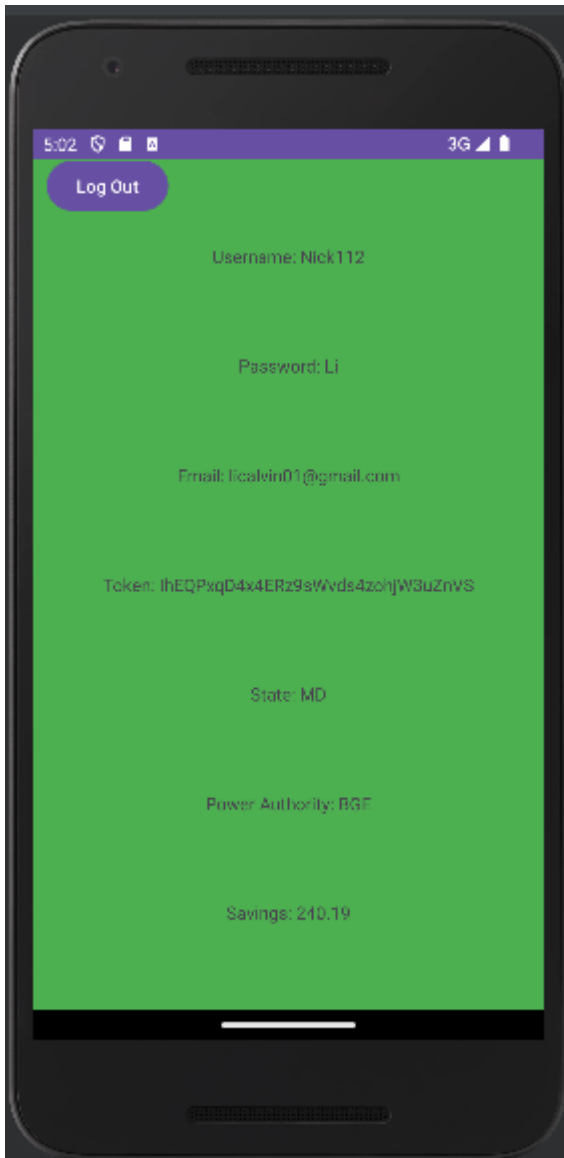
        profileButton.setOnClickListener {
            var intent = Intent(this@MainMenuV2, ProfilePage::class.java)
            startActivity(intent)
        }

        schedButton.setOnClickListener {
            var intent = Intent(this@MainMenuV2, CalendarEvent::class.java)
            startActivity(intent)
        }

        analyticsButton.setOnClickListener {
            var intent = Intent(this@MainMenuV2, AnalyticsPage::class.java)
            startActivity(intent)
        }
    }
}
```

## Profile Page

Should the user choose to navigate to the profile page, the app will send a GET request using the login information with the same logic incorporated. Once the request is completed, the information is stored temporarily and displayed on screen as shown below.



The profile page also allows the option for the user to log out which when pressed sends the user back to the login screen. The code for the profile page can be found below.

```
package com.example.prototype

import android.content.Intent
import android.os.AsyncTask
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.TextView
import okhttp3.OkHttpClient
import okhttp3.Request

class ProfilePage : AppCompatActivity() {
    lateinit var button: Button

    lateinit var emailPR: TextView
    lateinit var passwordPR: TextView
    lateinit var savingsPR: TextView
    lateinit var statePR: TextView
```

```

lateinit var tokenPR: TextView
lateinit var usernamePR: TextView
lateinit var PAPR: TextView

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_profile_page)

    button = findViewById(R.id.logout)

    emailPR = findViewById(R.id.emailProfile)
    passwordPR = findViewById(R.id.passwordProfile)
    savingsPR = findViewById(R.id.savingsProfile)
    statePR = findViewById(R.id.stateProfile)
    tokenPR = findViewById(R.id.tokenProfile)
    usernamePR = findViewById(R.id.usernameProfile)
    PAPR = findViewById(R.id.powerAuthorityProfile)

    /*
        var getEmail:String = intent.getStringExtra("emailPR").toString()
        var getPassword:String = intent.getStringExtra("passwordPR").toString()
        var getSavings:String = intent.getStringExtra("savingsPR").toString()
        var getState:String = intent.getStringExtra("statePR").toString()
        var getToken:String = intent.getStringExtra("tokenPR").toString()
        var getUsername:String = intent.getStringExtra("usernamePR").toString()
        var getPA:String = intent.getStringExtra("PAPR").toString()

        email.text = "Email: $getEmail"
        password.text = "Password: $getPassword"
        savings.text = "Savings: $getSavings"
        state.text = "State: $getState"
        token.text = "Token: $getToken"
        username.text = "Username: $getUsername"
        PA.text = "Power Authority: $getPA"

    */

    val url = "https://mrzn69.pythonanywhere.com/user_login?
username=Nick112&password=Li"

    MyAsyncTask().execute(url)

    button.setOnClickListener {
        var intent = Intent(this@ProfilePage, MainActivity::class.java)
        startActivity(intent)
    }
}

private inner class MyAsyncTask : AsyncTask<String, Void, String>() {

    override fun doInBackground(vararg params: String): String {
        try {
            val url = params[0]
            val client = OkHttpClient()

```

```

        // Create a request
        val request = Request.Builder()
            .url(url)
            .build()

        // Execute the request and get the response
        val response = client.newCall(request).execute()

        if (!response.isSuccessful) {
            // Handle non-successful response (e.g., log or return an
error message)
            return "Error: ${response.code}"
        }

        return response.body?.string() ?: ""
    } catch (e: Exception) {
        // Handle any exceptions that might occur during the network
operation
        e.printStackTrace()
        return "Error: ${e.message}"
    }
}

override fun onPostExecute(result: String) {
    super.onPostExecute(result)

    // Parse the response string
    val parsedStrings = parseResponse(result)

    // Now you can use parsedStrings for further processing
    val username = parsedStrings[0]
    val password = parsedStrings[1]
    val savings = parsedStrings[2]
    val token = parsedStrings[3]
    val powerAuthority = parsedStrings[4]
    val state = parsedStrings[5]
    val email = parsedStrings[6]

    // Do something with the parsed strings (e.g., update UI elements)
    emailPR.text = "Email: $email"
    passwordPR.text = "Password: $password"
    savingsPR.text = "Savings: $savings"
    statePR.text = "State: $state"
    tokenPR.text = "Token: $token"
    usernamePR.text = "Username: $username"
    PAPER.text = "Power Authority: $powerAuthority"
}

private fun parseResponse(response: String): List<String> {
    val lines = response.split(",")

```

```

        // Assuming the response structure is consistent
        val username = lines[0]
        val password = lines[1]
        val savings = lines[2]
        val token = lines[3]
        val powerAuthority = lines[4]
        val state = lines[5]
        val email = lines[6]

        return listOf(username, password, savings, token, powerAuthority,
state, email)
    }
}
}

```

## Scheduler

The second option the user is able to select is the scheduling activity which allows the user to schedule tasks throughout the day. Once they select scheduling, it sends them to the calendar app integrated into their Android device and creates the event. Due to time constraints, some features of the calendar were not able to be implemented on time including a pop up for the peak hour of the day. The logic for the calendar can be found below where data is essentially sent with Intent to create the event for the calendar.

```

package com.example.calendar

import android.annotation.SuppressLint
import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import java.text.SimpleDateFormat
import java.util.*

class MainActivity : AppCompatActivity() {
    @SuppressLint("SimpleDateFormat")
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Declaring and initializing
        // the button from the layout file
        val mButton = findViewById<Button>(R.id.button_1)

        // Event start and end time with date
        val startTime = "2022-02-1T09:00:00"
        val endTime = "2022-02-1T12:00:00"

        // Parsing the date and time
        val mSimpleDateFormat = SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss")
        val mStartTime = mSimpleDateFormat.parse(startTime)
    }
}

```



```

        val mEndTime = mSimpleDateFormat.parse(endTime)

        // When Button is clicked, Intent started
        // to create an event with given time
        mButton.setOnClickListener {
            val mIntent = Intent(Intent.ACTION_EDIT)
            mIntent.type = "vnd.android.cursor.item/event"
            mIntent.putExtra("beginTime", mStartTime.time)
            mIntent.putExtra("time", true)
            mIntent.putExtra("rule", "FREQ=YEARLY")
            mIntent.putExtra("endTime", mEndTime.time)
            mIntent.putExtra("title", "Event")
            startActivity(mIntent)
        }
    }
}

```

## Analytics

The last activity the user can navigate to is the analytics page which displays a graph of the comparison of the user's savings compared to a user who had a flat rate. An example of the analytics page can be found below.





To obtain this image, a GET request was made to the server that contained the image of the graph of the user's analytics at that time and once the request was completed, it was then locally saved and pasted to fit in the middle of the Android device. The code for the analytics can be found below.

```
package com.example.prototype

import android.graphics.Bitmap
import android.graphics.BitmapFactory
import android.os.AsyncTask
import android.os.Bundle
import android.widget.ImageView
import androidx.appcompat.app.AppCompatActivity
import okhttp3.OkHttpClient
import okhttp3.Request
import java.io.IOException
import java.io.InputStream
import java.net.URL

class AnalyticsPage : AppCompatActivity() {

    private lateinit var imageView: ImageView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_analytics_page)
    }
}
```

```
        imageView = findViewById(R.id.graph)

        // Replace 'YOUR_IMAGE_URL' with the actual URL of the image you want to
display
        val imageUrl = "https://mrzn69.pythonanywhere.com/img"

        // Execute AsyncTask to download and display the image
        ImageDownloader().execute(imageUrl)
    }

    private inner class ImageDownloader : AsyncTask<String, Void, Bitmap?>() {

        override fun doInBackground(vararg params: String): Bitmap? {
            val imageUrl = params[0]
            return try {
                // Use OkHttp client to send a GET request
                val client = OkHttpClient()
                val request: Request = Request.Builder().url(imageUrl).build()
                val response = client.newCall(request).execute()

                // Check if the request was successful (status code 200)
                if (response.isSuccessful) {
                    // Get the input stream from the response body
                    val inputStream: InputStream = response.body?.byteStream() ?:
return null

                    // Decode the input stream into a Bitmap
                    BitmapFactory.decodeStream(inputStream)
                } else {
                    null
                }
            } catch (e: IOException) {
                e.printStackTrace()
                null
            }
        }

        override fun onPostExecute(result: Bitmap?) {
            // Display the downloaded image in the ImageView
            result?.let {
                imageView.setImageBitmap(result)
            }
        }
    }
}
```