BINUS UNIVERSITY
BINUS INTERNATIONAL

Assignment Cover Letter

(Individual Work)

Student Information:		**Surname**	**Given Names**	**Student ID Number**
	1.	**Lumban Tobing**	**Zefanya**	**2201796970**

Course Code	**: COMP6056**	**Course Name**	**: Program Design Methods**
Class	**: L1AC**	**Name of Lecturer(s)**	: 1. Minaldi Loeis
Major	**: CS**		
Title of Assignment (if any)	: Voice-controlled Actions		
Type of Assignment	**: Final Project**		
Submission Pattern			
Due Date	**: 20-11-2018**	**Submission Date**	**: 20-11-2018**

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: (Name of Student)

1. Zefanya Gedalya B.L.T

“Voice-controlled Actions”

Zefanya Gedalya Banikristo Lumban Tobing
2201796970

1. Introduction

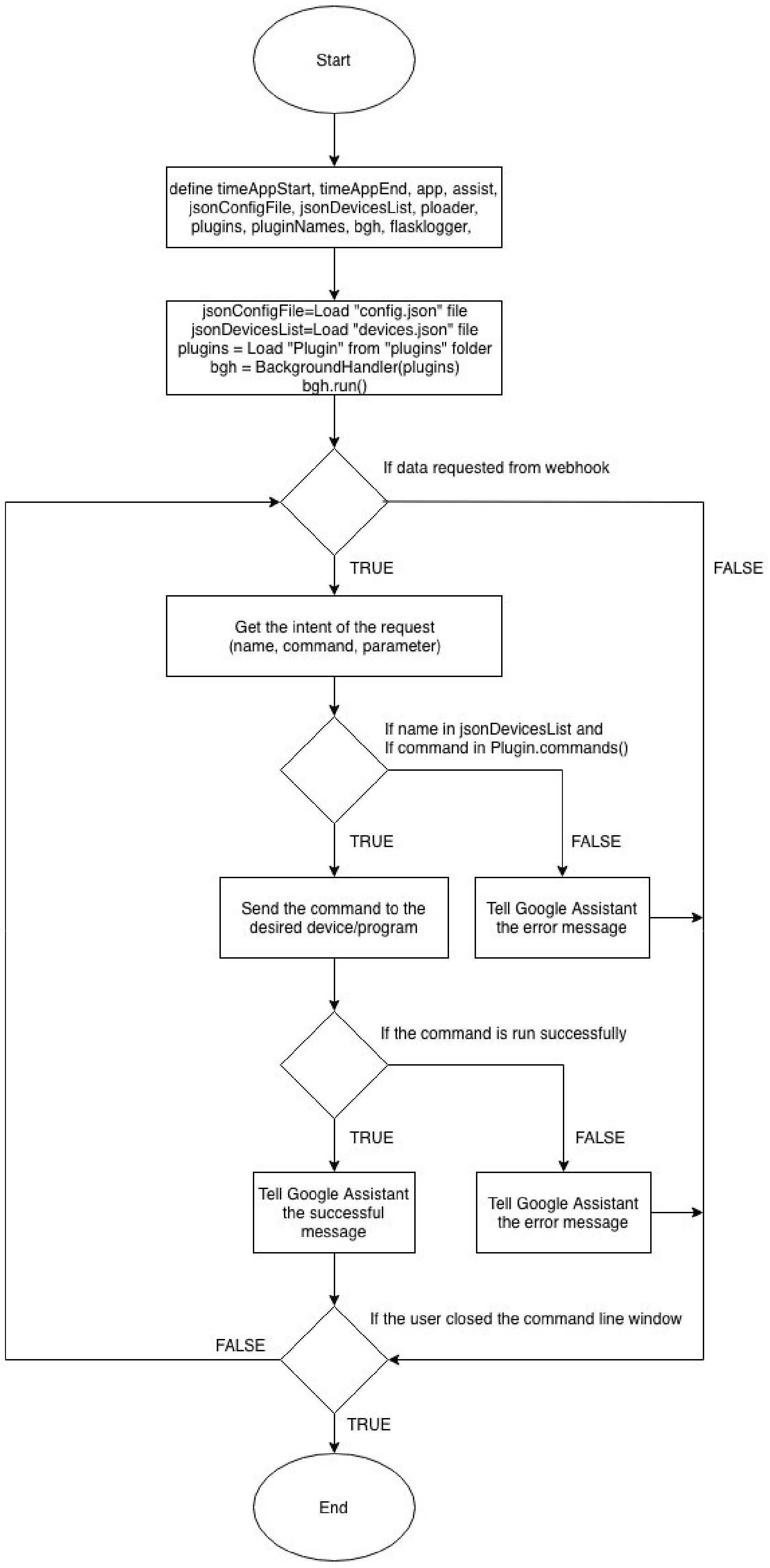
Concept

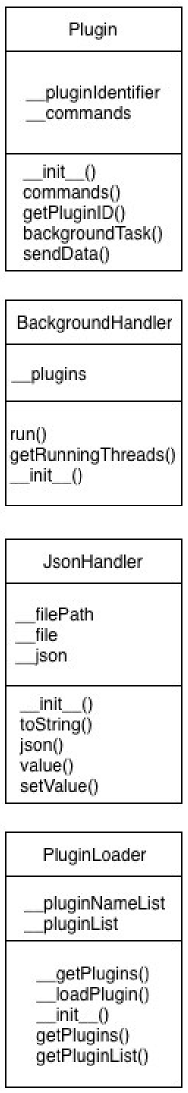
This program is designed to handle voice input from Google Assistant, and process it accordingly. It acts as a bridge for other programs, or even a microcontroller such as Arduino. It works by processing the user's input by voice. The program then processes what the user wanted to do, check if the action is set-up by the user and then sends the command to the program or device that the user intended to do.

Problem

Voice recognition and voice assistant technology are available for the past few years. But the technology is very underutilized. It had useful features already such as reminders, calendar management and many more. But there is no way to customize it fully by default. With this program, custom voice commands and actions are possible.

2. Design


Start
define timeAppStart, timeAppEnd, app, assist, jsonConfigFile, jsonDevicesList, ploader, plugins, pluginNames, bgh, flasklogger,
jsonConfigFile=Load "config.json" file
jsonDevicesList=Load "devices.json" file
plugins = Load "Plugin" from "plugins" folder
bgh = BackgroundHandler(plugins)
bgh.run()
If data requested from webhook
TRUE
FALSE
Get the intent of the request (name, command, parameter)
If name in jsonDevicesList and
If command in Plugin.commands()
TRUE
FALSE
Send the command to the desired device/program
Tell Google Assistant the error message
If the command is run successfully
TRUE
FALSE
Tell Google Assistant the successful message
Tell Google Assistant the error message
If the user closed the command line window
FALSE
TRUE
End


Plugin
__pluginIdentifier
__commands
__init__()
commands()
getPluginID()
backgroundTask()
sendData()
BackgroundHandler
__plugins
run()
getRunningThreads()
__init__()
JsonHandler
__filePath
__file
__json
__init__()
toString()
json()
value()
setValue()
PluginLoader
__pluginNameList
__pluginList
__getPlugins()
__loadPlugin()
__init__()
getPlugins()
getPluginList()

3. Discussion

Implementation

To make this project possible, it uses API and external applications. This program depends on packages such as Flask, Flask-Assistant, paho-mqtt, and colorama. It also uses other applications nor services such as Dialogflow, Actions on Google, and ngrok.

Actions on Google is a service from Google for developers to connect Google Assistant with another service, Dialogflow. This is crucial to this project, as it uses Google's voice recognition services.

Colorama is a package that makes adding color easier on the command-line output. In this project, it is used to differentiate between different message easily. For example: info, errors, and warnings.

Dialogflow is a service that handles the input from Google Assistant and interprets the human language to something that computers can understand. This is used to determine what the user wanted to do and sends it to the program through a webhook request.

Flask is a web framework used to handle webhook requests from Dialogflow. Flask works with Flask-Assistant to parse the data from Dialogflow.

Ngrok is a program to forward local web servers to the internet. In this project, its used as an alternative for port-forwarding.

How it works

When the program starts, it will initialize the Flask framework first. The program then reads the "config.json" file which contains the configuration of the app and reads "devices.json" file which contains the things that can be controlled. When that's done, the program will load the modules stored in the "plugins" folder. Then the program will run "backgroundTask()" function from the loaded plugins. That function is running all the time, parallel with the main code. It handles actions that are needed to be done while the main code is still running. In this case, one of the plugins will check the connection to a device periodically. After all of this process, the program will wait for webhook requests from Dialogflow.

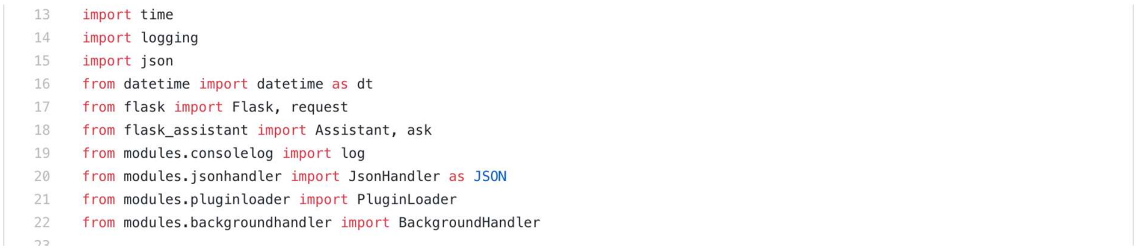
The plugin system works by having identical classes in different files within the same folder. The class has to have the same name (in this case the class name must be "Plugin"), core attributes (__pluginIdentifier, __commands), and core methods (commands, getPluginID, backgroundTask, and sendData). It is needed because the main program will call those methods. If another attributes or methods are needed, it needs to be called by one of those methods. The plugin files are loaded using a module built into Python called importlib. The function of importlib is to dynamically import modules, so it does not have to be hardcoded like "import plugin1", "import plugin2", etc.

When the program received a webhook request, it will first determine what's the intent of the request. Then it will run the processes needed for certain intent. It will check if the "name" on the webhook request is listed in jsonDeviceList. Then it will check id the plugin ID configured in the devices.json file exists. It will also check if the "command" on the webhook request is on the plugin's "command" attribute. If all of the conditions are true, the program will send the command to the desired device. If one of them is false, it will tell the user that something is wrong. After the code runs, it will check if the command is sent successfully, by waiting for a message sent back from the device. If the message is received by the program, then we can assume that the command has been received and done successfully. If there is no response, it will tell the user that something went wrong.

Ngrok is used so the local web server can be accessed from the internet. It’s used as an alternative for port forwarding which is not an option for some places like public Wi-Fi networks. This is mainly used for connecting between programs on a different network and can be used for webhook address for Dialogflow. But in this case, the program will be running on a virtual private server (with a static IP address), and the other program that’s controlled are running on a PC, thus ngrok is used on the PC (which the IP address changes dynamically).

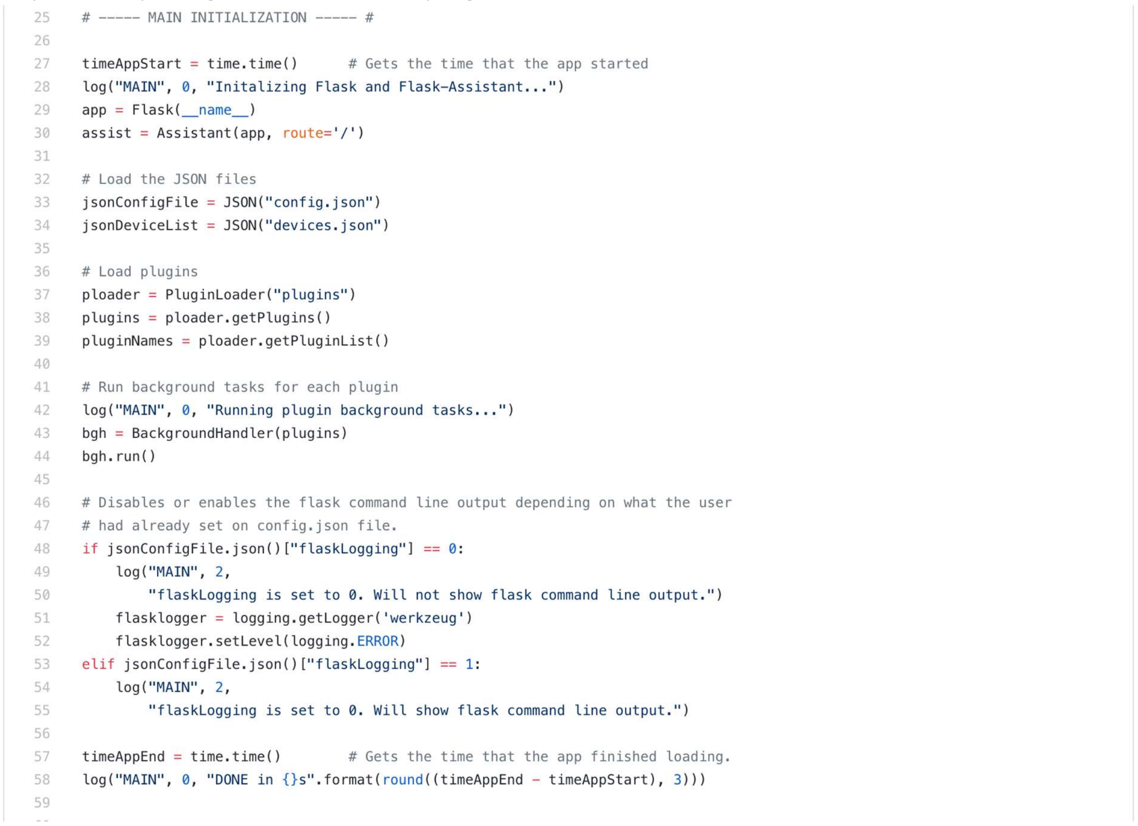
Code Explanation

main.py



```python
import time
import logging
import json
from datetime import datetime as dt
from flask import Flask, request
from flask_assistant import Assistant, ask
from modules.consolelog import log
from modules.jsonhandler import JsonHandler as JSON
from modules.pluginloader import PluginLoader
from modules.backgroundhandler import BackgroundHandler
```

This part of the code is used to import the modules from the “modules” folder, and imports the packages needed for this program.



```python
# ----- MAIN INITIALIZATION ----- #

timeAppStart = time.time()      # Gets the time that the app started
log("MAIN", 0, "Initalizing Flask and Flask-Assistant...")
app = Flask(__name__)
assist = Assistant(app, route='/')

# Load the JSON files
jsonConfigFile = JSON("config.json")
jsonDeviceList = JSON("devices.json")

# Load plugins
ploader = PluginLoader("plugins")
plugins = ploader.getPlugins()
pluginNames = ploader.getPluginList()

# Run background tasks for each plugin
log("MAIN", 0, "Running plugin background tasks...")
bgh = BackgroundHandler(plugins)
bgh.run()

# Disables or enables the flask command line output depending on what the user
# had already set on config.json file.
if jsonConfigFile.json()["flaskLogging"] == 0:
    log("MAIN", 2,
        "flaskLogging is set to 0. Will not show flask command line output.")
    flasklogger = logging.getLogger('werkzeug')
    flasklogger.setLevel(logging.ERROR)
elif jsonConfigFile.json()["flaskLogging"] == 1:
    log("MAIN", 2,
        "flaskLogging is set to 0. Will show flask command line output.")

timeAppEnd = time.time()        # Gets the time that the app finished loading.
log("MAIN", 0, "DONE in {}s".format(round((timeAppEnd - timeAppStart), 3)))
```

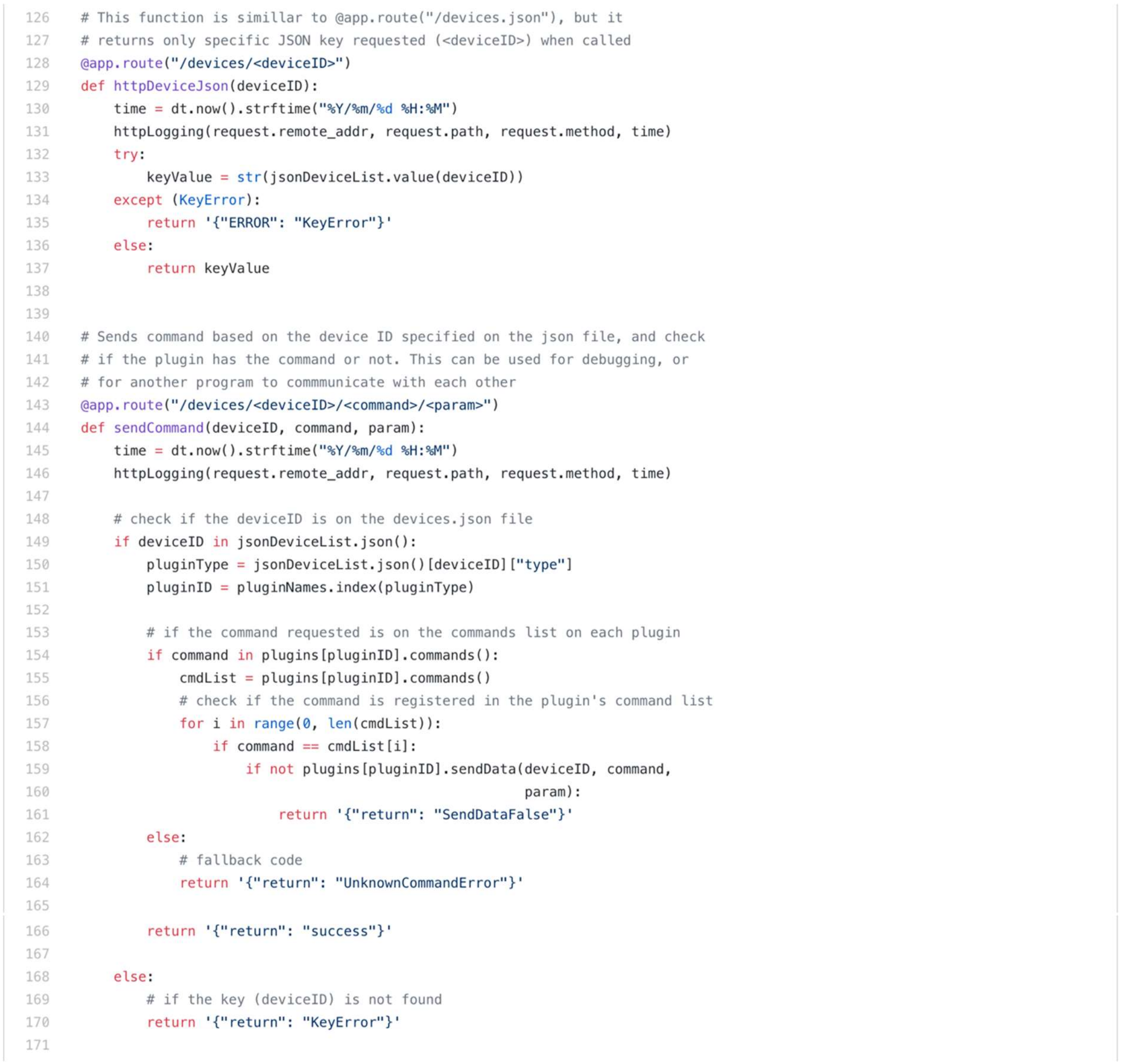
timeAppStart and timeAppEnd are used to count on how long it takes for the program to load everything. It’s used for debugging purposes as it helps what changes made the program slower. The variable app = Flask(__name__) basically initializes Flask based on the QuickStart guide. assist = Assistant(app, route=’/’) initializes Flask-Assistant also based on the

QuickStart guide. jsonConfigFile and jsonDeviceList are an object of a class “JsonHandler”. It reads the config.json file and devices.json file to be accessed later during runtime. bgh is an object of a class “BackgroundHandler”. It loads background tasks that from each plugin. The variable flasklogger is used to disable the command line output of flask, except for error messages.

```python
# A function that handles data sending to a certain plugin that returns
# a boolean value
def sendData(plugin, id, value, param):
    for i in range(len(pluginNames)):
        if pluginNames[i].lower() == plugin.lower():
            if plugins[i].sendData(id, value, param):
                return True
    return False

# A function that handles data sending to a certain plugin that returns
# a string


def sendDataStr(plugin, id, value, param):
    for i in range(len(pluginNames)):
        if pluginNames[i].lower() == plugin.lower():
            return plugins[i].sendData(id, value, param)
    return False


# A function to log the HTTP request activities
def httpLogging(ip, path, method, time):
    log("MAIN", 0, "{} {} \"{}\" at {}".format(ip, method, path, time))

```

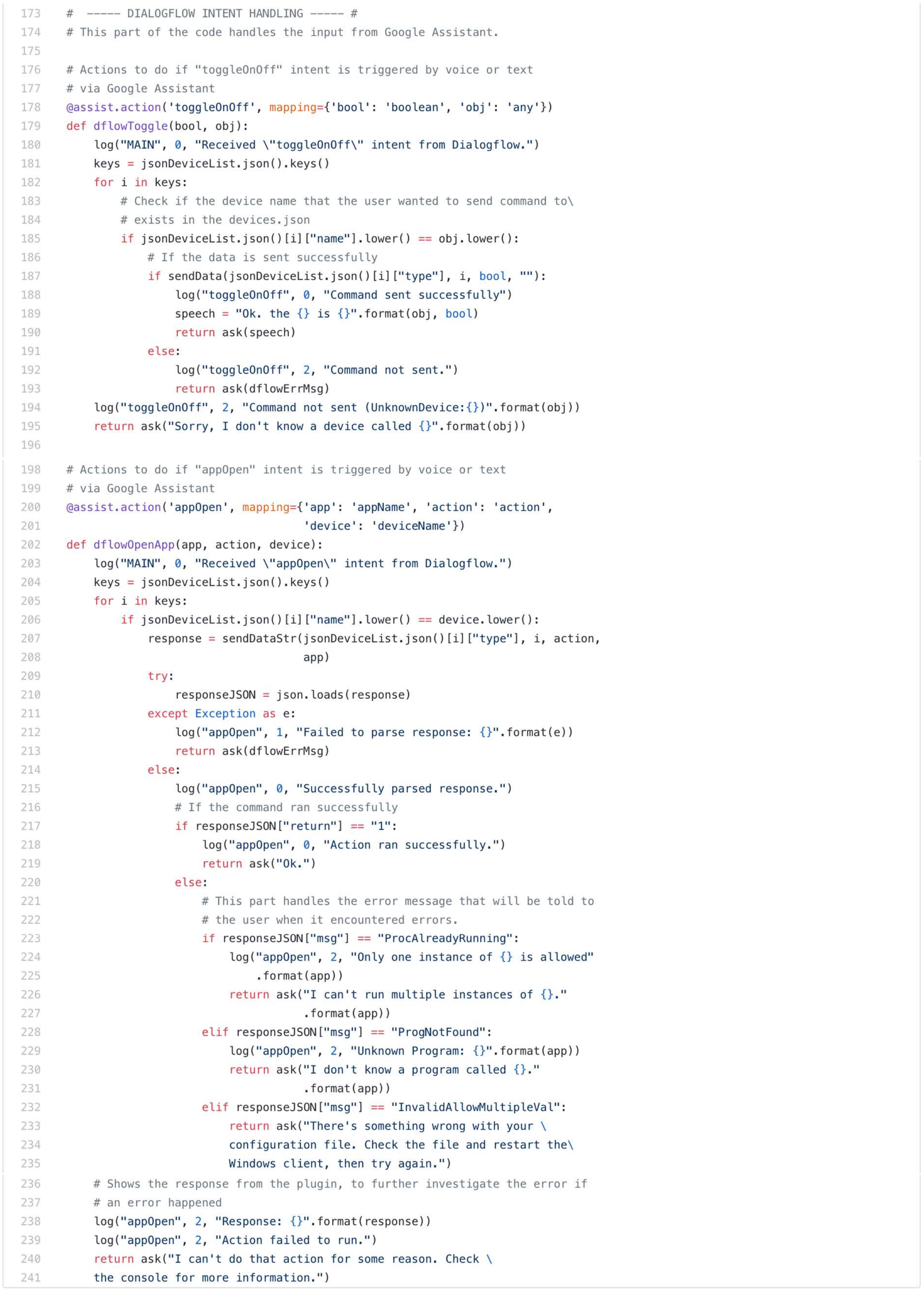
The “sendData” function is used to call the “sendData()” function of the desired plugin and passes the information to it. It will return a Boolean value. On the other hand, “sendDataStr()” has the same function, but it will return a string. This is used when a plugin outputs a string. The “httpLogging()” function is there for debugging purposes. It prints the information when a HTTP request is made.

```python
# ----- HTTP REQUEST HANDLING ----- #
# This part of the code is mainly used for debugging purposes. Some of it is
# used to communicate between programs via the internet

# Shows the message when user opened "/" in the web browser, indicating that
# the program is running successfully
@app.route("/")
def httpRoot():
    time = dt.now().strftime("%Y/%m/%d %H:%M")
    httpLogging(request.remote_addr, request.path, request.method, time)
    return "If you see this message, the program is running."


# Shows the devices.json file when user opened "/devices.json"
@app.route("/devices.json")
def httpDevicesJson():
    time = dt.now().strftime("%Y/%m/%d %H:%M")
    httpLogging(request.remote_addr, request.path, request.method, time)
    return jsonDeviceList.toString()


# Shows the config.json file when user opened "/config.json"
@app.route("/config.json")
def httpConfigJson():
    time = dt.now().strftime("%Y/%m/%d %H:%M")
    httpLogging(request.remote_addr, request.path, request.method, time)
    return jsonConfigFile.toString()


# Shows the list of running threads when the user opened "/threads"
@app.route("/threads")
def httpThreadsList():
    time = dt.now().strftime("%Y/%m/%d %H:%M")
    httpLogging(request.remote_addr, request.path, request.method, time)
    return str(bgh.getRunningThreads())


```



```python
# This function is simillar to @app.route("/devices.json"), but it
# returns only specific JSON key requested (<deviceID>) when called
@app.route("/devices/<deviceID>")
def httpDeviceJson(deviceID):
    time = dt.now().strftime("%Y/%m/%d %H:%M")
    httpLogging(request.remote_addr, request.path, request.method, time)
    try:
        keyValue = str(jsonDeviceList.value(deviceID))
    except (KeyError):
        return '{"ERROR": "KeyError"}'
    else:
        return keyValue


# Sends command based on the device ID specified on the json file, and check
# if the plugin has the command or not. This can be used for debugging, or
# for another program to commmunicate with each other
@app.route("/devices/<deviceID>/<command>/<param>")
def sendCommand(deviceID, command, param):
    time = dt.now().strftime("%Y/%m/%d %H:%M")
    httpLogging(request.remote_addr, request.path, request.method, time)

    # check if the deviceID is on the devices.json file
    if deviceID in jsonDeviceList.json():
        pluginType = jsonDeviceList.json()[deviceID]["type"]
        pluginID = pluginNames.index(pluginType)

        # if the command requested is on the commands list on each plugin
        if command in plugins[pluginID].commands():
            cmdList = plugins[pluginID].commands()
            # check if the command is registered in the plugin's command list
            for i in range(0, len(cmdList)):
                if command == cmdList[i]:
                    if not plugins[pluginID].sendData(deviceID, command,
                                                      param):
                        return '{"return": "SendDataFalse"}'
        else:
            # fallback code
            return '{"return": "UnknownCommandError"}'

        return '{"return": "success"}'

    else:
        # if the key (deviceID) is not found
        return '{"return": "KeyError"}'

```

This part of the code is used to handle HTTP requests through the web browser. It's mainly used for debugging purposes, but some are for communicating between programs. In this case, the WindowsControl plugin needs another program to send its address, so the other program will pass the address using the "sendCommand" function. The @ symbol at the "@app.route()" is a decorator function. It is used by flask to run certain steps (a function one line below the decorator line) when the user requested certain address.



```python
#  ----- DIALOGFLOW INTENT HANDLING ----- #
# This part of the code handles the input from Google Assistant.

# Actions to do if "toggleOnOff" intent is triggered by voice or text
# via Google Assistant
@assist.action('toggleOnOff', mapping={'bool': 'boolean', 'obj': 'any'})
def dflowToggle(bool, obj):
    log("MAIN", 0, "Received \"toggleOnOff\" intent from Dialogflow.")
    keys = jsonDeviceList.json().keys()
    for i in keys:
        # Check if the device name that the user wanted to send command to\
        # exists in the devices.json
        if jsonDeviceList.json()[i]["name"].lower() == obj.lower():
            # If the data is sent successfully
            if sendData(jsonDeviceList.json()[i]["type"], i, bool, ""):
                log("toggleOnOff", 0, "Command sent successfully")
                speech = "Ok. the {} is {}".format(obj, bool)
                return ask(speech)
            else:
                log("toggleOnOff", 2, "Command not sent.")
                return ask(dflowErrMsg)
    log("toggleOnOff", 2, "Command not sent (UnknownDevice:{})".format(obj))
    return ask("Sorry, I don't know a device called {}".format(obj))

# Actions to do if "appOpen" intent is triggered by voice or text
# via Google Assistant
@assist.action('appOpen', mapping={'app': 'appName', 'action': 'action',
                                   'device': 'deviceName'})
def dflowOpenApp(app, action, device):
    log("MAIN", 0, "Received \"appOpen\" intent from Dialogflow.")
    keys = jsonDeviceList.json().keys()
    for i in keys:
        if jsonDeviceList.json()[i]["name"].lower() == device.lower():
            response = sendDataStr(jsonDeviceList.json()[i]["type"], i, action,
                                   app)
            try:
                responseJSON = json.loads(response)
            except Exception as e:
                log("appOpen", 1, "Failed to parse response: {}".format(e))
                return ask(dflowErrMsg)
            else:
                log("appOpen", 0, "Successfully parsed response.")
                # If the command ran successfully
                if responseJSON["return"] == "1":
                    log("appOpen", 0, "Action ran successfully.")
                    return ask("Ok.")
                else:
                    # This part handles the error message that will be told to
                    # the user when it encountered errors.
                    if responseJSON["msg"] == "ProcAlreadyRunning":
                        log("appOpen", 2, "Only one instance of {} is allowed"
                            .format(app))
                        return ask("I can't run multiple instances of {}."
                                   .format(app))
                    elif responseJSON["msg"] == "ProgNotFound":
                        log("appOpen", 2, "Unknown Program: {}".format(app))
                        return ask("I don't know a program called {}."
                                   .format(app))
                    elif responseJSON["msg"] == "InvalidAllowMultipleVal":
                        return ask("There's something wrong with your \
                        configuration file. Check the file and restart the\
                        Windows client, then try again.")
    # Shows the response from the plugin, to further investigate the error if
    # an error happened
    log("appOpen", 2, "Response: {}".format(response))
    log("appOpen", 2, "Action failed to run.")
    return ask("I can't do that action for some reason. Check \
    the console for more information.")
```

This part of the code handles the user input (from Google Assistant). @assist.action() is a decorator function used, so Flask-Assistant can do the actions we wanted to do.
@assist.action("appOpen", mapping={'app': 'appName', 'action': 'action', 'device': 'deviceName'})
"appOpen" is the intent specified on Dialogflow. The keys in "mapping" is used to map variables from Dialogflow webhook requests to Python variables (the key will be the Python variable; the value is the variable from Dialogflow).

modules/consolelog.py

```python
from colorama import init, Fore, Style
init()


def log(moduleName, type, string):
    # Coloring for types of console logging
    if type == 0:
        t = "{}INFO{}".format(Fore.BLUE, Style.RESET_ALL)
    elif type == 1:
        t = "{} ERR{}".format(Fore.RED, Style.RESET_ALL)
    elif type == 2:
        t = "{}WARN{}".format(Fore.LIGHTYELLOW_EX, Style.RESET_ALL)

    n = "{}{}{}".format(Fore.LIGHTCYAN_EX, moduleName, Style.RESET_ALL)
    print("{} [{}] {}".format(t, n, string))
```

This module prints the command line output and adds color to differentiate between the message type. There are three different types of command line output defined by this code, which is info, error, and warning. Those three different types have its own color. The color is handled by a package called colorama.

modules/backgroundhandler.py

```python
import threading
from modules.consolelog import log


class BackgroundHandler:
    # Initialize the object with a list that contains plugins.
    def __init__(self, plugins):
        log("BGHANDLER", 0, "Loaded {} plugins.".format(len(plugins)))
        self.__plugins = plugins

    # Runs all the plugins on the plugins list.
    def run(self):
        for plugin in self.__plugins:
            pName = plugin.getPluginID()
            log("BGHANDLER", 0, "Running {} background task...".format(pName))
            t = threading.Thread(target=plugin.backgroundTask,
                                 name=pName)
            t.start()

    # Gets the list of running threads
    def getRunningThreads(self):
        return threading.enumerate()
```

This module handles the running of the background tasks that each plugin has. It works by applying the concept of multithreading using the Threading module built into Python. “self.__plugins” is a list containing an object “Plugin”. The “run()” method will loop through the “self.__plugins” list and runs the “backgroundTask()” method that each plugin has. “getRunningThread()” method returns the current active threads.

modules/jsonhandler.py

```python
import json
from modules.consolelog import log


class JsonHandler:
    def __init__(self, path):
        try:
            self.__filePath = path
            self.__file = open(path, "r").read()
        except Exception as e:
            log("JSON", 1, e)
        else:
            self.__json = json.loads(self.__file)
            log("JSON", 0, "Loaded {}".format(self.__filePath))

    def toString(self):
        return json.dumps(self.__json)

    def json(self):
        return self.__json

    def value(self, key):
        return self.__json[key]

    def setValue(self, key, value):
        try:
            self.__json[key] = value
        except Exception as e:
            log("JSON", 1, e)
            return False
        else:
            return True
```

This module contains a class that is used to load JSON file and store it in the memory which will be used later when a program needs data from the JSON file. JSON itself is an abbreviation of “JavaScript Object Notation”. In this case, it is used for storing configuration files.

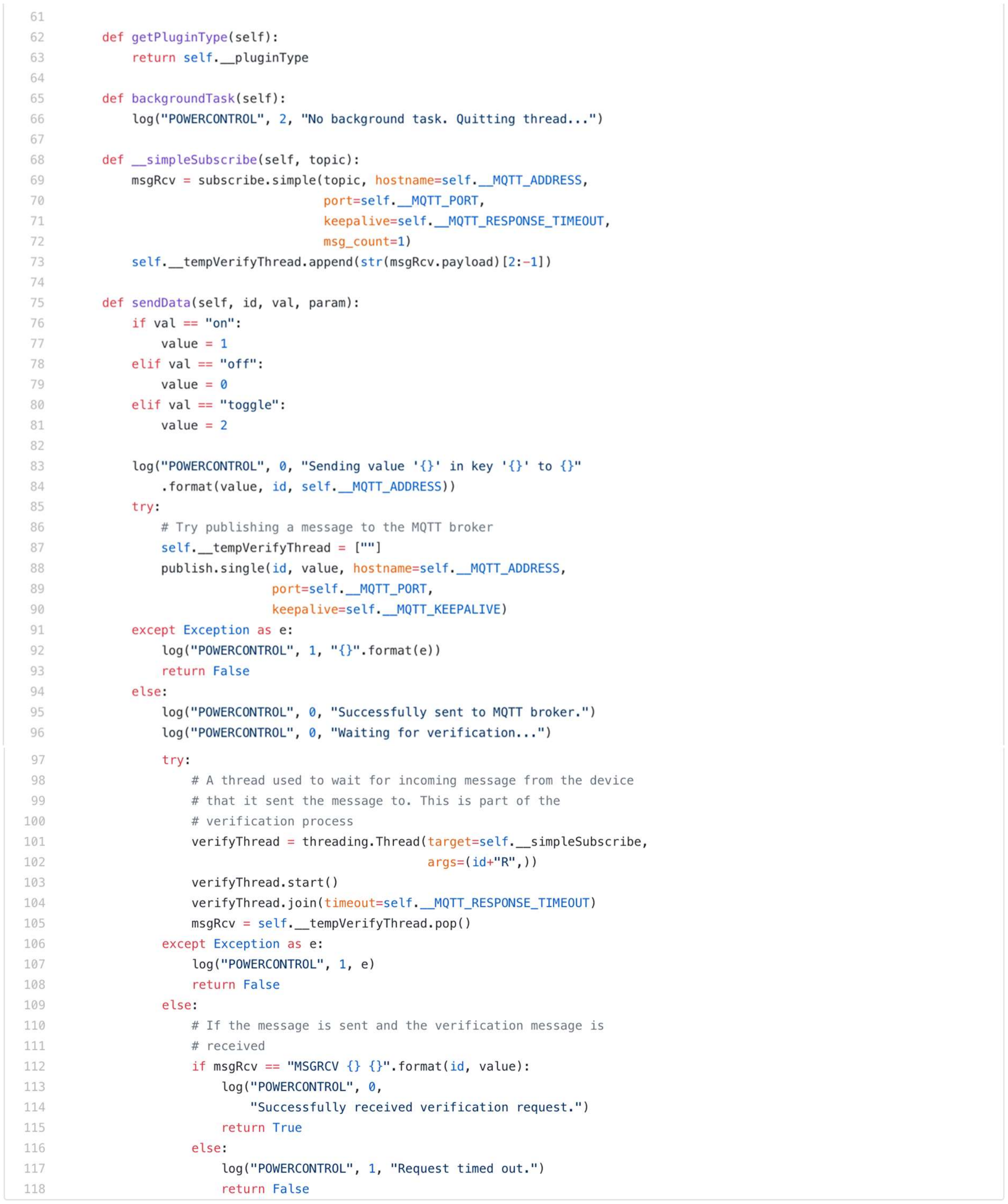
modules/pluginloader.py

```python
import importlib
import os
from modules.consolelog import log


class PluginLoader:
    # Gets the list of python files in the directory.
    def __getPlugins(self, pluginDir):
        for (dirpath, dirnames, filenames) in os.walk(pluginDir):
            x = []
            x.extend(filenames)
            x = x[::-1]
            xa = []
            for i in range(0, len(x)):
                if ".py" in x[i]:
                    x[i] = x[i].replace(".py", "")
                    xa.append(x[i])
            return xa

    # A method to construct the plugin and put the plugins in a list
    def __loadPlugin(self, pluginDir, pluginNameList):
        plugins = []
        for i in range(len(pluginNameList)):
            log("MAIN", 0, "Loading plugin {}".format(pluginNameList[i]))
            pluginFile = str(pluginDir) + "." + str(pluginNameList[i])
            currentPlugin = importlib.import_module(pluginFile, ".")
            plugins.append(currentPlugin.Plugin())
        return plugins

    def __init__(self, pluginDir):
        self.__pluginNameList = self.__getPlugins(pluginDir)
        self.__pluginList = self.__loadPlugin(pluginDir, self.__pluginNameList)

    def getPlugins(self):
        return self.__pluginList

    def getPluginList(self):
        return self.__pluginNameList
```

This module is the core of the modularity part of this program. This module is used for plugin related task. It loads plugins from the specified plugins directory. This is possible with a module built into Python called importlib. Importlib is a module that can import modules dynamically. Rather than having all modules imported manually (for example: import plugin1; import plugin2; etc.), it will look for python files in the plugins folder, then imports it automatically. The "__getPlugins()" method is used to get the list of Python files (.py) in the directory. The "__loadPlugin()" method is used to get the class inside the Python files and construct it into a list. Since the Python files has the same class inside of it which is "Plugin", It can just loop through the list of plugin files and construct the object and put it in a list. Those 2 functions then will be called when the class "PluginLoader" is initialized by the "__init__()" method. The other two methods "getPlugins()" and "getPluginList()" will be called in the main.py file.

plugins/powercontrol.py

```python
from datetime import datetime as dt
import json
import threading
from modules.consolelog import log
# import paho.mqtt.client as mqtt
import paho.mqtt.publish as publish
import paho.mqtt.subscribe as subscribe


class Plugin:
    # function to open the configuration file
    def __openConfig(self):
        try:
            file = open("plugins/powercontrol/config.json", "r").read()
        except (FileNotFoundError):
            log("POWERCONTROL", 1, "Config file not found.")
            exit()
        else:
            log("POWERCONTROL", 0, "Loaded config.json")
            return json.loads(file)

    # plugin initialzation
    def __init__(self):
        self.__pluginIdentifier = "powerControl"
        self.__commands = ["on", "off", "toggle"]

        # loads the config.json and set each keys to a variable
        self.__configFile = self.__openConfig()
        self.__MQTT_ADDRESS = self.__configFile["MQTT_ADDRESS"]
        self.__MQTT_PORT = self.__configFile["MQTT_PORT"]
        self.__MQTT_KEEPALIVE = self.__configFile["MQTT_KEEPALIVE"]
        self.__MQTT_RESPONSE_TIMEOUT = self.__configFile[
            "MQTT_RESPONSE_TIMEOUT"]

        currentDT = dt.now().strftime("%Y/%m/%d %H:%M")
        log("POWERCONTROL", 0, "PowerControl initialized at {}"
            .format(currentDT))

        # temporary variable for thread returns
        self.__tempVerifyThread = []

    def commands(self):
        return self.__commands

    def getPluginID(self):
        return self.__pluginIdentifier
```



```
 61
 62     def getPluginType(self):
 63         return self.__pluginType
 64
 65     def backgroundTask(self):
 66         log("POWERCONTROL", 2, "No background task. Quitting thread...")
 67
 68     def __simpleSubscribe(self, topic):
 69         msgRcv = subscribe.simple(topic, hostname=self.__MQTT_ADDRESS,
 70                                   port=self.__MQTT_PORT,
 71                                   keepalive=self.__MQTT_RESPONSE_TIMEOUT,
 72                                   msg_count=1)
 73         self.__tempVerifyThread.append(str(msgRcv.payload)[2:-1])
 74
 75     def sendData(self, id, val, param):
 76         if val == "on":
 77             value = 1
 78         elif val == "off":
 79             value = 0
 80         elif val == "toggle":
 81             value = 2
 82
 83         log("POWERCONTROL", 0, "Sending value '{}' in key '{}' to {}"
 84             .format(value, id, self.__MQTT_ADDRESS))
 85         try:
 86             # Try publishing a message to the MQTT broker
 87             self.__tempVerifyThread = [""]
 88             publish.single(id, value, hostname=self.__MQTT_ADDRESS,
 89                            port=self.__MQTT_PORT,
 90                            keepalive=self.__MQTT_KEEPALIVE)
 91         except Exception as e:
 92             log("POWERCONTROL", 1, "{}".format(e))
 93             return False
 94         else:
 95             log("POWERCONTROL", 0, "Successfully sent to MQTT broker.")
 96             log("POWERCONTROL", 0, "Waiting for verification...")
 97             try:
 98                 # A thread used to wait for incoming message from the device
 99                 # that it sent the message to. This is part of the
100                 # verification process
101                 verifyThread = threading.Thread(target=self.__simpleSubscribe,
102                                                 args=(id+"R",))
103                 verifyThread.start()
104                 verifyThread.join(timeout=self.__MQTT_RESPONSE_TIMEOUT)
105                 msgRcv = self.__tempVerifyThread.pop()
106             except Exception as e:
107                 log("POWERCONTROL", 1, e)
108                 return False
109             else:
110                 # If the message is sent and the verification message is
111                 # received
112                 if msgRcv == "MSGRCV {} {}".format(id, value):
113                     log("POWERCONTROL", 0,
114                         "Successfully received verification request.")
115                     return True
116                 else:
117                     log("POWERCONTROL", 1, "Request timed out.")
118                     return False
```

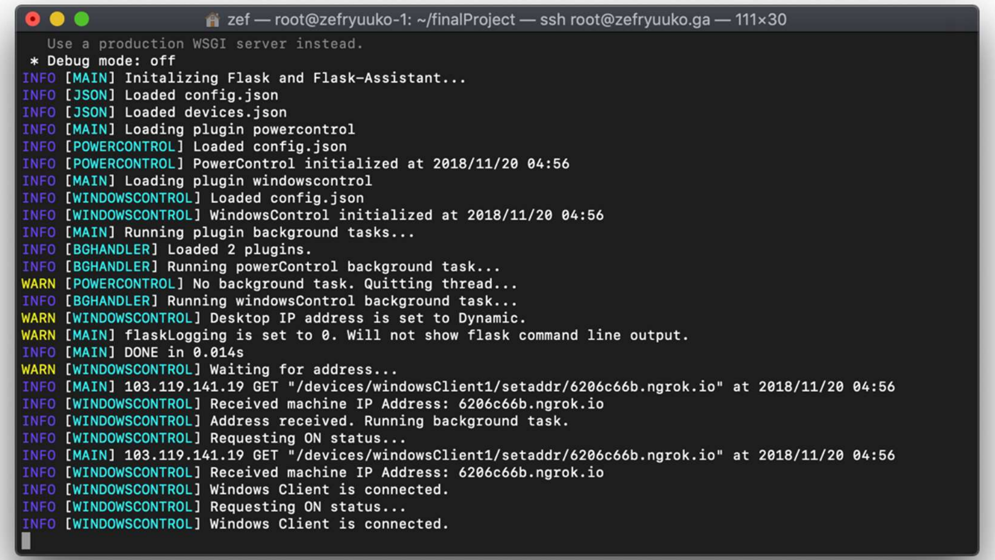
This plugin is made to forward what the user wants to do, to devices using the MQTT (Mosquitto) protocol, for example: lights, power sockets, etc. In this project, it is used to send data to an Arduino. Arduino is a microcontroller that can be programmed easily. When "sendData()" method gets the data (arguments) from the "sendData()" on the main.py file, it will try to connect to the MQTT broker (a program that handles MQTT message publishing and subscribing) and sends the message to the broker (line 88). The cons of using MQTT, there is no way to determine if the message is received by another device by default. To solve this problem, the other device has to send a message back to the program, so that the program knows that the message has been sent successfully. The program waits for message (line 101) until a certain timeout value stored on the powercontrol's config.json file. If the program received a message, then it's safe to say that the message went through, and vice versa.

plugins/windowscontrol.py

```python
from datetime import datetime as dt
from modules.consolelog import log
import json
import time
import urllib.request


class Plugin:
    # A method to open the configuration file.
    def __openConfig(self):
        try:
            file = open("plugins/windowscontrol/config.json", "r").read()
        except (FileNotFoundError):
            log("WINDOWSCONTROL", 1, "Config file not found.")
            exit()
        except Exception as e:
            log("WINDOWSCONTROL", 1, "Error: {}".format(e))
        else:
            log("WINDOWSCONTROL", 0, "Loaded config.json")
            return json.loads(file)

    def __init__(self):
        self.__pluginIdentifier = "windowsControl"
        self.__commands = ["run", "kill", "check", "setaddr"]

        self.__configFile = self.__openConfig()
        self.__isDynamicAddress = self.__configFile["isDynamicAddress"]
        self.__desktopAddress = self.__configFile["desktopAddress"]
        self.__pingInterval = self.__configFile["pingInterval"]

        currentDT = dt.now().strftime("%Y/%m/%d %H:%M")
        log("WINDOWSCONTROL", 0, "WindowsControl initialized at {}"
            .format(currentDT))

    def commands(self):
        return self.__commands

    def getPluginID(self):
        return self.__pluginIdentifier

    def getPluginType(self):
        return self.__pluginType

    def backgroundTask(self):
        # If the PC is using dynamic address
        if self.__isDynamicAddress == 1:
            isAddrFound = False
            self.__desktopAddress = ""
            log("WINDOWSCONTROL", 2, "Desktop IP address is set to Dynamic.")
            log("WINDOWSCONTROL", 2, "Waiting for address...")
            while not isAddrFound:
                # Wait until the windows client sends it's address
                if self.__desktopAddress != "":
                    log("WINDOWSCONTROL", 0,
                        "Address received. Running background task.")
                    isAddrFound = True

            url = "http://{}/{}".format(self.__desktopAddress, "ping")
            # Check the connection at X minutes. (Interval set on config.json)
            while True:
                url = url.replace(" ", "%20")
                log("WINDOWSCONTROL", 0, "Requesting ON status...")
                try:
                    response = urllib.request.urlopen(url).read()[2:-1]
                except Exception as e:
                    log("WINDOWSCONTROL", 1, "ERROR: {}".format(e))
                else:
                    log("WINDOWSCONTROL", 0, "Windows Client is connected.")
                    self.__desktopAddress = response
                    time.sleep(self.__pingInterval*60)

    def sendData(self, id, value, param):
        if value == "run" or value == "kill":
            url = "http://{addr}/{cmd}/{app}".format(addr=self.__desktopAddress,
                                                      cmd=value, app=param)
            url = url.replace(" ", "%20")
            # Requests response from the windows client app.
            log("WINDOWSCONTROL", 0, "Requesting response from {}".format(url))
            response = json.loads(str(urllib.request.urlopen(url).read())[2:-1])
            return json.dumps(response)
        elif value == "setaddr":
            self.__desktopAddress = param
            log("WINDOWSCONTROL", 0, "Received machine IP Address: {}"
                .format(param))
```

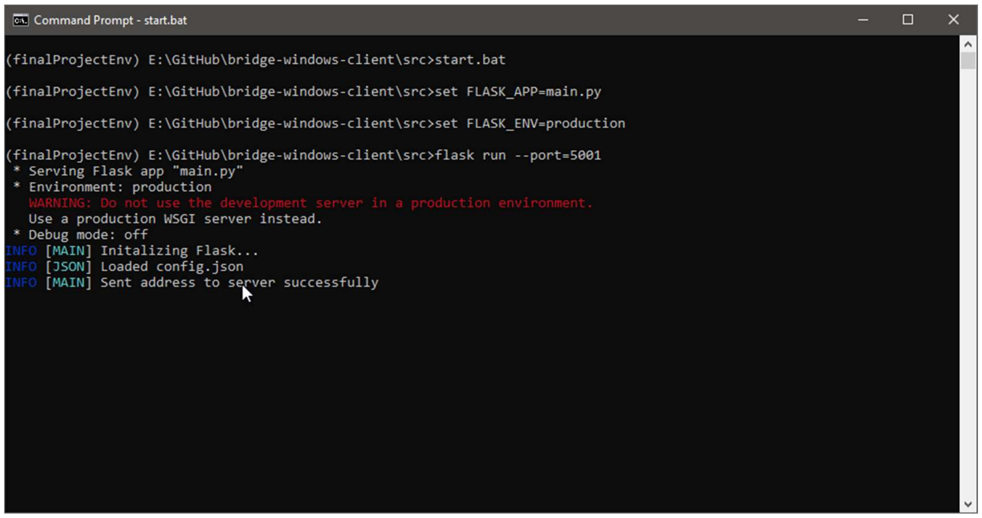
This plugin is made to communicate between this program and a client program running Windows. The purpose of this is to tell the client to run or kill certain process (application) on a computer running Windows. The system is more or less the same compared to powercontrol.py. But what makes it different is the way this two programs communicate. Rather than using MQTT, this plugin and the client communicates with HTTP requests.

4. Evidence



```
zef — root@zefryuuko-1: ~/finalProject — ssh root@zefryuuko.ga — 111×30
   Use a production WSGI server instead.
 * Debug mode: off
INFO [MAIN] Initalizing Flask and Flask-Assistant...
INFO [JSON] Loaded config.json
INFO [JSON] Loaded devices.json
INFO [MAIN] Loading plugin powercontrol
INFO [POWERCONTROL] Loaded config.json
INFO [POWERCONTROL] PowerControl initialized at 2018/11/20 04:56
INFO [MAIN] Loading plugin windowscontrol
INFO [WINDOWSCONTROL] Loaded config.json
INFO [WINDOWSCONTROL] WindowsControl initialized at 2018/11/20 04:56
INFO [MAIN] Running plugin background tasks...
INFO [BGHANDLER] Loaded 2 plugins.
INFO [BGHANDLER] Running powerControl background task...
WARN [POWERCONTROL] No background task. Quitting thread...
INFO [BGHANDLER] Running windowsControl background task...
WARN [WINDOWSCONTROL] Desktop IP address is set to Dynamic.
WARN [MAIN] flaskLogging is set to 0. Will not show flask command line output.
INFO [MAIN] DONE in 0.014s
WARN [WINDOWSCONTROL] Waiting for address...
INFO [MAIN] 103.119.141.19 GET "/devices/windowsClient1/setaddr/6206c66b.ngrok.io" at 2018/11/20 04:56
INFO [WINDOWSCONTROL] Received machine IP Address: 6206c66b.ngrok.io
INFO [WINDOWSCONTROL] Address received. Running background task.
INFO [WINDOWSCONTROL] Requesting ON status...
INFO [MAIN] 103.119.141.19 GET "/devices/windowsClient1/setaddr/6206c66b.ngrok.io" at 2018/11/20 04:56
INFO [WINDOWSCONTROL] Received machine IP Address: 6206c66b.ngrok.io
INFO [WINDOWSCONTROL] Windows Client is connected.
INFO [WINDOWSCONTROL] Requesting ON status...
INFO [WINDOWSCONTROL] Windows Client is connected.
```

Figure 1: Application running on a virtual private server, and established connection with a Windows client program on another network.



```
Command Prompt - start.bat

(finalProjectEnv) E:\GitHub\bridge-windows-client\src>start.bat

(finalProjectEnv) E:\GitHub\bridge-windows-client\src>set FLASK_APP=main.py

(finalProjectEnv) E:\GitHub\bridge-windows-client\src>set FLASK_ENV=production

(finalProjectEnv) E:\GitHub\bridge-windows-client\src>flask run --port=5001
 * Serving Flask app "main.py"
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
INFO [MAIN] Initalizing Flask...
INFO [JSON] Loaded config.json
INFO [MAIN] Sent address to server successfully
```

Figure 2: Windows client program sends it's address to the main program.

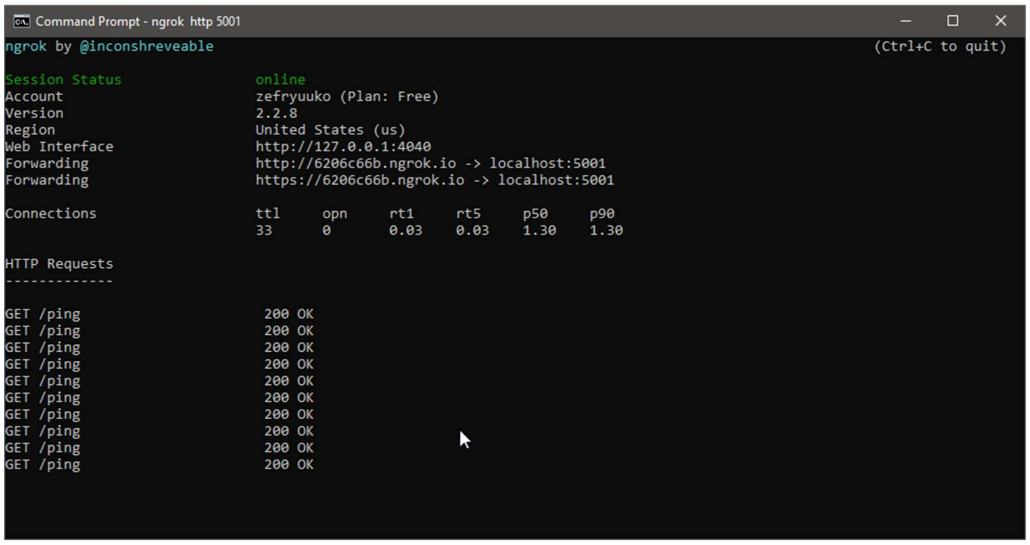
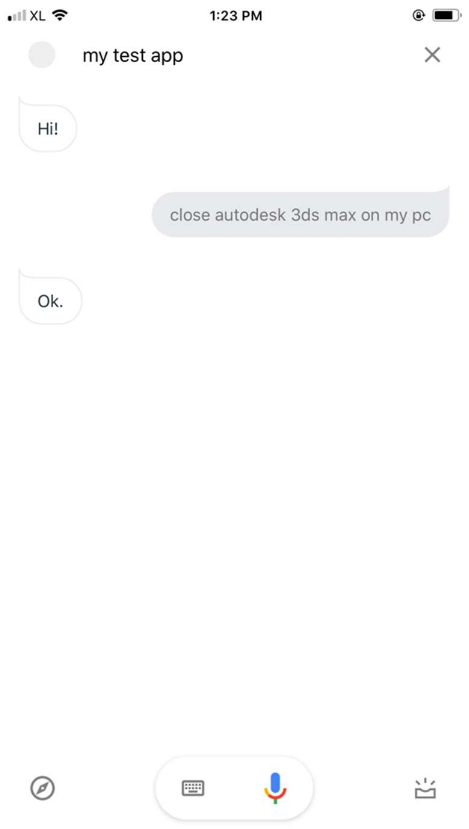

```
Command Prompt - ngrok  http 5001
ngrok by @inconshreveable                                                          (Ctrl+C to quit)

Session Status                online
Account                       zefryuuko (Plan: Free)
Version                       2.2.8
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://6206c66b.ngrok.io -> localhost:5001
Forwarding                    https://6206c66b.ngrok.io -> localhost:5001

Connections                   ttl     opn     rt1     rt5     p50     p90
                              33      0       0.03    0.03    1.30    1.30

HTTP Requests
-------------

GET /ping                      200 OK
GET /ping                      200 OK
GET /ping                      200 OK
GET /ping                      200 OK
GET /ping                      200 OK
GET /ping                      200 OK
GET /ping                      200 OK
GET /ping                      200 OK
GET /ping                      200 OK
GET /ping                      200 OK
```

Figure 3: ngrok forwards local port 5001 to externally visible address on port 80 (HTTP)



Hi!

close autodesk 3ds max on my pc

Ok.

Figure 4: Google Assistant controlling a Windows PC.

5. Sources

Documentations:

- https://codelabs.developers.google.com/codelabs/actions-1/#0
- https://dialogflow.com/docs/getting-started
- https://docs.python.org/3/reference/index.html
- https://pypi.org/project/paho-mqtt/
- http://flask.pocoo.org/docs/1.0/
- https://flask-assistant.readthedocs.io/en/latest/
- https://pypi.org/project/colorama/
- https://dashboard.ngrok.com/get-started

Other sources (Stack Overflow, YouTube, etc.) is listed on

- https://github.com/zefryuuko/pdm-final-project/blob/master/references.txt
- https://github.com/zefryuuko/pdm-final-project/blob/master/src-windowsclient/references.txt
- https://github.com/zefryuuko/pdm-final-project/blob/master/src-arduino/references.txt