Zebra DevCon 2023

ZEBRA

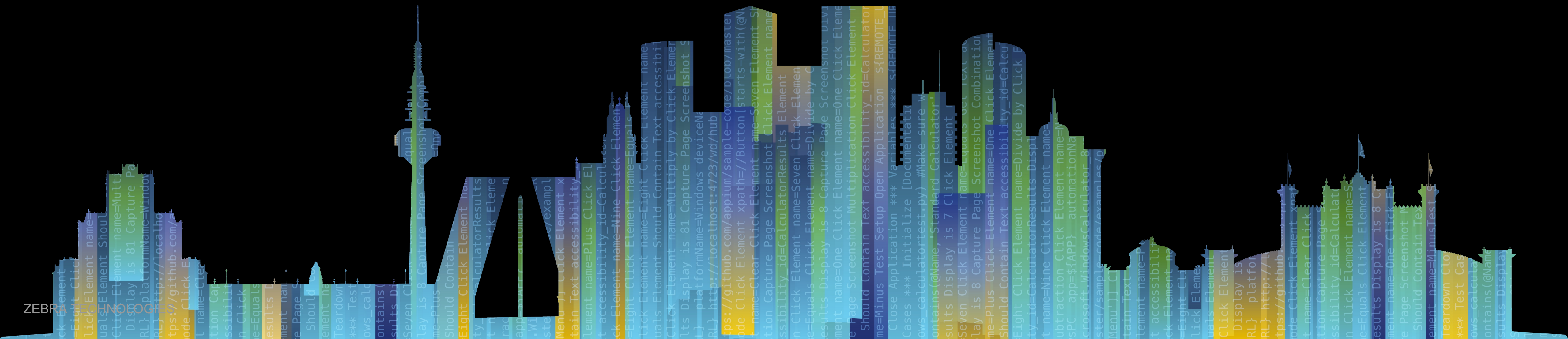
Embedded scripting in Python & NodeJS

Gary Crean

Senior Software Engineer

Zebra
DevCon 2023

User Applications (DA apps): Overview



User Applications

User Applications

- Runs on the RFID reader, No external control software needed
- Previous embedded applications were written in either C++ or Java
- Difficult to maintain without a build environment setup
- Support libraries and generic configuration had to be implemented every time

User Applications

User Applications (DA apps): DA Library

- DA Apps must make use of the DA Library to be able to send messages across.
- DA library abstracts the underlying connections between the ZIoTC components.
- The DA modules are available in below languages
  - Python 3.9
  - NodeJS
- The apps must be packaged as deb files, like an embedded User App.
- The apps can be installed via Web Console/ZIoTC management interface.

User Applications

Features and Highlights

Zebra DevCon 2023

- Supported on FX7500, FX9600, ATR7000.
- Enables connectivity to the cloud platforms to provide IOT capabilities to the reader.
- Supports independent interfaces for Management, control, data and monitoring.
- Supports Data retention during network disconnects.
- Supports various pre-defined but configurable radio operating modes.
- Supports multiple modes of deployment for fully cloud, on-prem, hybrid modes of operation.
- Support two simultaneous data paths.
- Supports sending different data to different data paths using the DA app framework.
- Supports a User-App framework called DA framework for writing custom applications using Python or NodeJS.

User Applications

Connectivity

- Zebra Data Services (ZDS)
- Message Queuing Telemetry Transport (MQTT)
- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- HTTP Post
- IBM Watson IoT
- TCP
- Websocket
- Microsoft Azure
- Keyboard HID Emulation

User Applications

Reader Management/Monitoring/Control

- Management functionalities supported
  - Get Info
    - Status
    - Network
    - Region
  - Configure reader
    - Reader Profile
    - Endpoints
    - Events
    - GPIO-LED
  - Manage User Apps
  - Update Firmware

- Control functionalities supported
  - Control
    - Start
    - Stop
    - Mode

- Monitoring Events supported
  - Heartbeats
  - GPI
  - Error
  - Warnings
  - Firmware Update Progress

User Applications

Controlling GPOs and LED

- Provides an easy-to-use rules-based mechanism to control the reader GPOs and LED
- User can configure:
  - The default state of GPOs and LED
  - Event of Interest upon which a GPO and LED control action can be performed
  - Conditions to be met for the action to take place
  - The action to perform: the LED and GPO state/blink etc.

User Applications

Overview

Zebra DevCon 2023

- The following methods are available in the DA library for applications to use.
  - ziotc.ZIOTC() : Initializes the library. This will establish connections between the script and the other IoT Connector components
  - ziotcObject.reg_new_msg_callback() : Registers a callback function to be called when a message is received.
  - ziotcObject.reg_pass_through_callback() : Registers a callback function to be called on a control message
  - ziotcObject.enableGPIEvents() : Allows callback to receive GPI Events
  - ziotcObject.loop.run_forever(): This will cause any messages arriving to flow through the callback function
  - ziotcObject.send_next_msg(msg_type, msg_out): This will send the message out to the Reader Gateway to be handled appropriately. Following message types are supported.
    - ZIOTC_MSG_TYPE_DATA
    - ZIOTC_MSG_TYPE_CTRL
    - ZIOTC_MSG_TYPE_GPO

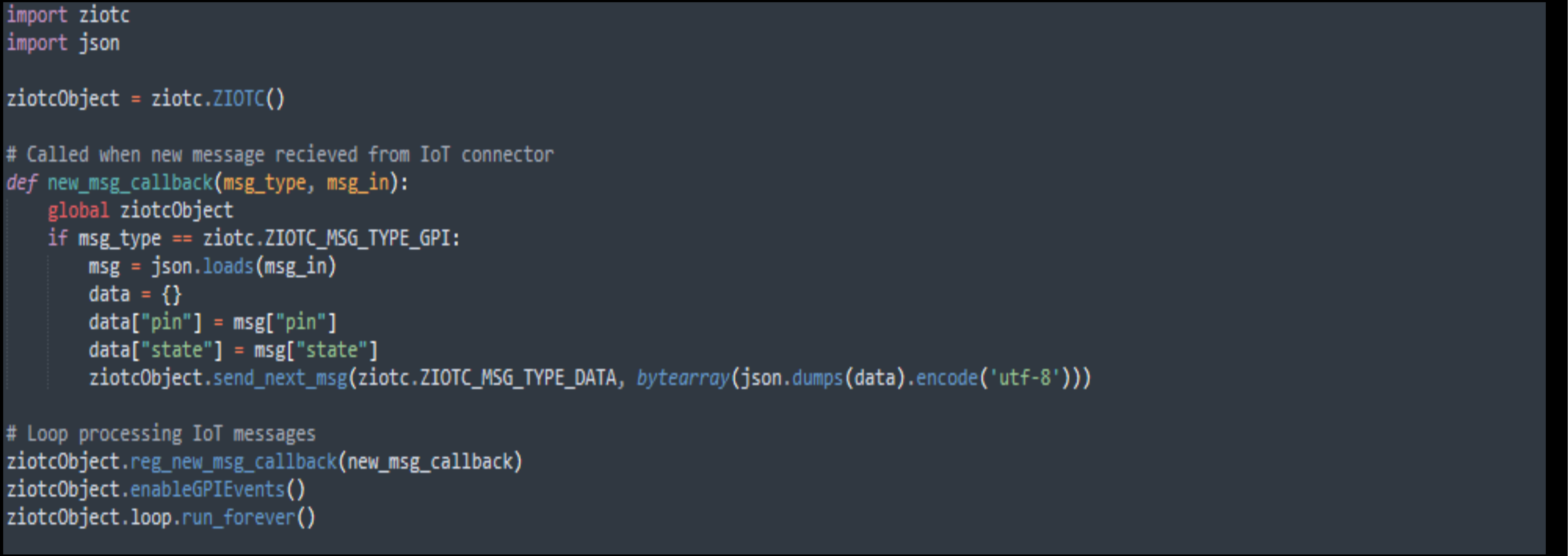
User Applications

Simple Python Application

```
def new_msg_callback(msg_type, msg_in):
    if msg_type == ziotc.ZIOTC_MSG_TYPE_TAG_INFO_JSON:
        msg_in_json = json.loads(msg_in.decode('utf-8'))
        tag_id_hex = msg_in_json["data"]["idHex"]
        ts = msg_in_json["timestamp"]
        tag = { “tag” : {} }
        tag[“id”] = tag_id_hex
        tag[“timestamp”] = ts
        ziotcObject.send_next_msg(ziotc.ZIOTC_MSG_TYPE_DATA, bytearray(json.dumps(tag).encode('utf-8’)))

ziotcObject = ziotc.ZIOTC()
ziotcObject.reg_new_msg_callback(new_msg_callback)
ziotcObject.loop.run_forever()
```

User Applications

Simple Python Application to Monitor GPI

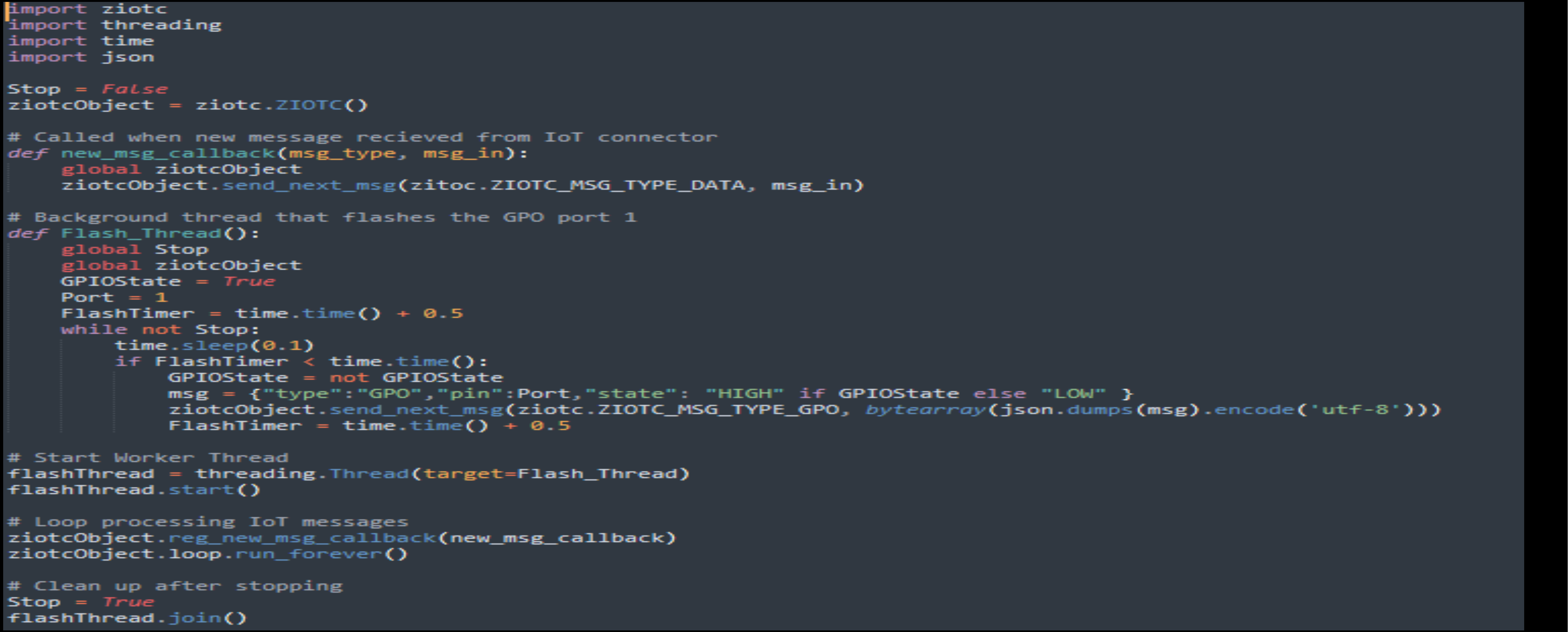


```
import ziotc
import json

ziotcObject = ziotc.ZIOTC()

# Called when new message recieved from IoT connector
def new_msg_callback(msg_type, msg_in):
    global ziotcObject
    if msg_type == ziotc.ZIOTC_MSG_TYPE_GPI:
        msg = json.loads(msg_in)
        data = {}
        data["pin"] = msg["pin"]
        data["state"] = msg["state"]
        ziotcObject.send_next_msg(ziotc.ZIOTC_MSG_TYPE_DATA, bytearray(json.dumps(data).encode('utf-8')))

# Loop processing IoT messages
ziotcObject.reg_new_msg_callback(new_msg_callback)
ziotcObject.enableGPIEvents()
ziotcObject.loop.run_forever()
```

User Applications

Simple Python Application to Flash GPO

Zebra DevCon 2023

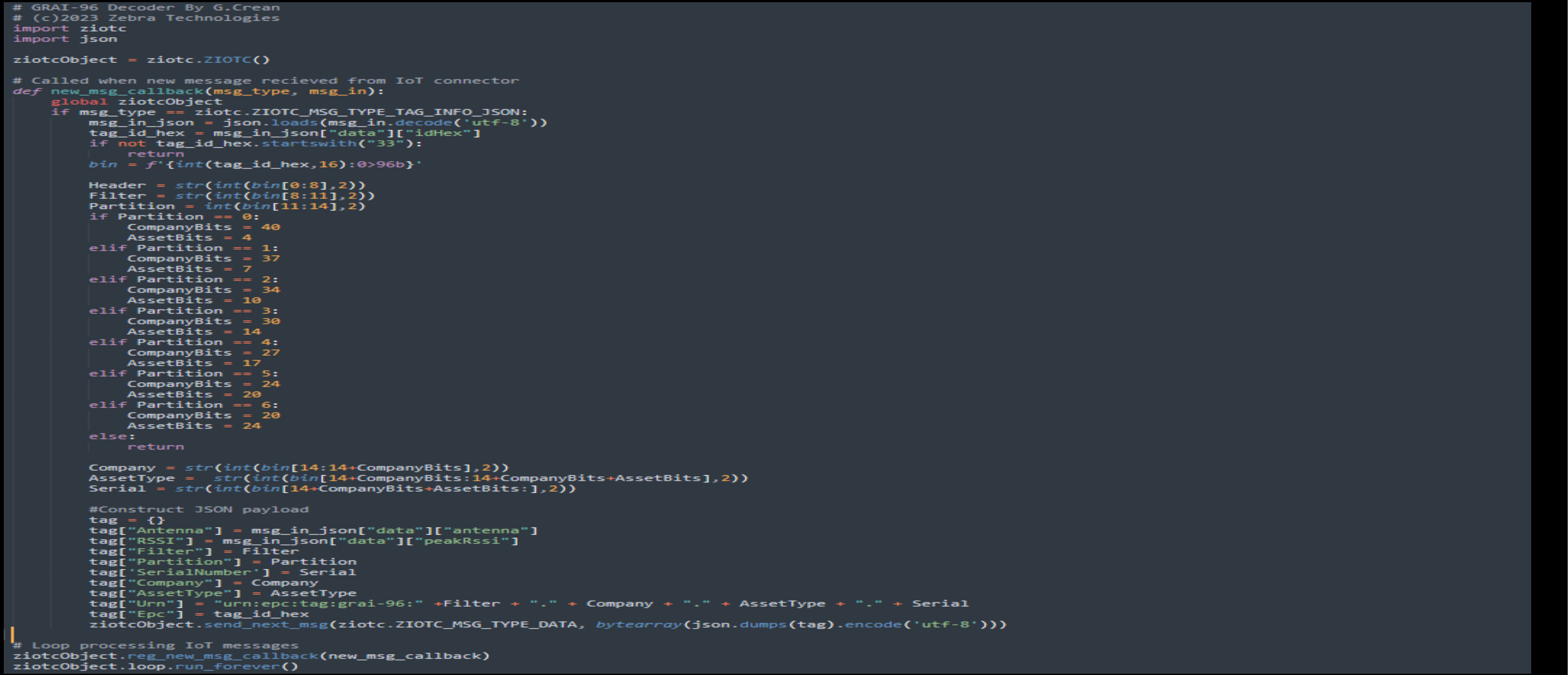


```
import ziotc
import threading
import time
import json

Stop = False
ziotcObject = ziotc.ZIOTC()

# Called when new message recieved from IoT connector
def new_msg_callback(msg_type, msg_in):
    global ziotcObject
    ziotcObject.send_next_msg(zitoc.ZIOTC_MSG_TYPE_DATA, msg_in)

# Background thread that flashes the GPO port 1
def Flash_Thread():
    global Stop
    global ziotcObject
    GPIOState = True
    Port = 1
    FlashTimer = time.time() + 0.5
    while not Stop:
        time.sleep(0.1)
        if FlashTimer < time.time():
            GPIOState = not GPIOState
            msg = {"type":"GPO","pin":Port,"state": "HIGH" if GPIOState else "LOW" }
            ziotcObject.send_next_msg(ziotc.ZIOTC_MSG_TYPE_GPO, bytearray(json.dumps(msg).encode('utf-8')))
            FlashTimer = time.time() + 0.5

# Start Worker Thread
flashThread = threading.Thread(target=Flash_Thread)
flashThread.start()

# Loop processing IoT messages
ziotcObject.reg_new_msg_callback(new_msg_callback)
ziotcObject.loop.run_forever()

# Clean up after stopping
Stop = True
flashThread.join()
```

User Applications

Simple Python Application to decode GRAI-96

Zebra
DevCon 2023



```python
# GRAI-96 Decoder By G.Crean
# (c)2023 Zebra Technologies
import ziotc
import json

ziotcObject = ziotc.ZIOTC()

# Called when new message recieved from IoT connector
def new_msg_callback(msg_type, msg_in):
    global ziotcObject
    if msg_type == ziotc.ZIOTC_MSG_TYPE_TAG_INFO_JSON:
        msg_in_json = json.loads(msg_in.decode('utf-8'))
        tag_id_hex = msg_in_json["data"]["idHex"]
        if not tag_id_hex.startswith("33"):
            return
        bin = f'{int(tag_id_hex,16):0>96b}'

        Header = str(int(bin[0:8],2))
        Filter = str(int(bin[8:11],2))
        Partition = int(bin[11:14],2)
        if Partition == 0:
            CompanyBits = 40
            AssetBits = 4
        elif Partition == 1:
            CompanyBits = 37
            AssetBits = 7
        elif Partition == 2:
            CompanyBits = 34
            AssetBits = 10
        elif Partition == 3:
            CompanyBits = 30
            AssetBits = 14
        elif Partition == 4:
            CompanyBits = 27
            AssetBits = 17
        elif Partition == 5:
            CompanyBits = 24
            AssetBits = 20
        elif Partition == 6:
            CompanyBits = 20
            AssetBits = 24
        else:
            return

        Company = str(int(bin[14:14+CompanyBits],2))
        AssetType =  str(int(bin[14+CompanyBits:14+CompanyBits+AssetBits],2))
        Serial = str(int(bin[14+CompanyBits+AssetBits:],2))

        #Construct JSON payload
        tag = {}
        tag["Antenna"] = msg_in_json["data"]["antenna"]
        tag["RSSI"] = msg_in_json["data"]["peakRssi"]
        tag["Filter"] = Filter
        tag["Partition"] = Partition
        tag['SerialNumber'] = Serial
        tag["Company"] = Company
        tag["AssetType"] = AssetType
        tag["Urn"] = "urn:epc:tag:grai-96:" +Filter + "." + Company + "." + AssetType + "." + Serial
        tag["Epc"] = tag_id_hex
        ziotcObject.send_next_msg(ziotc.ZIOTC_MSG_TYPE_DATA, bytearray(json.dumps(tag).encode('utf-8')))

# Loop processing IoT messages
ziotcObject.reg_new_msg_callback(new_msg_callback)
ziotcObject.loop.run_forever()
```

User Applications

Rest API Interface for management

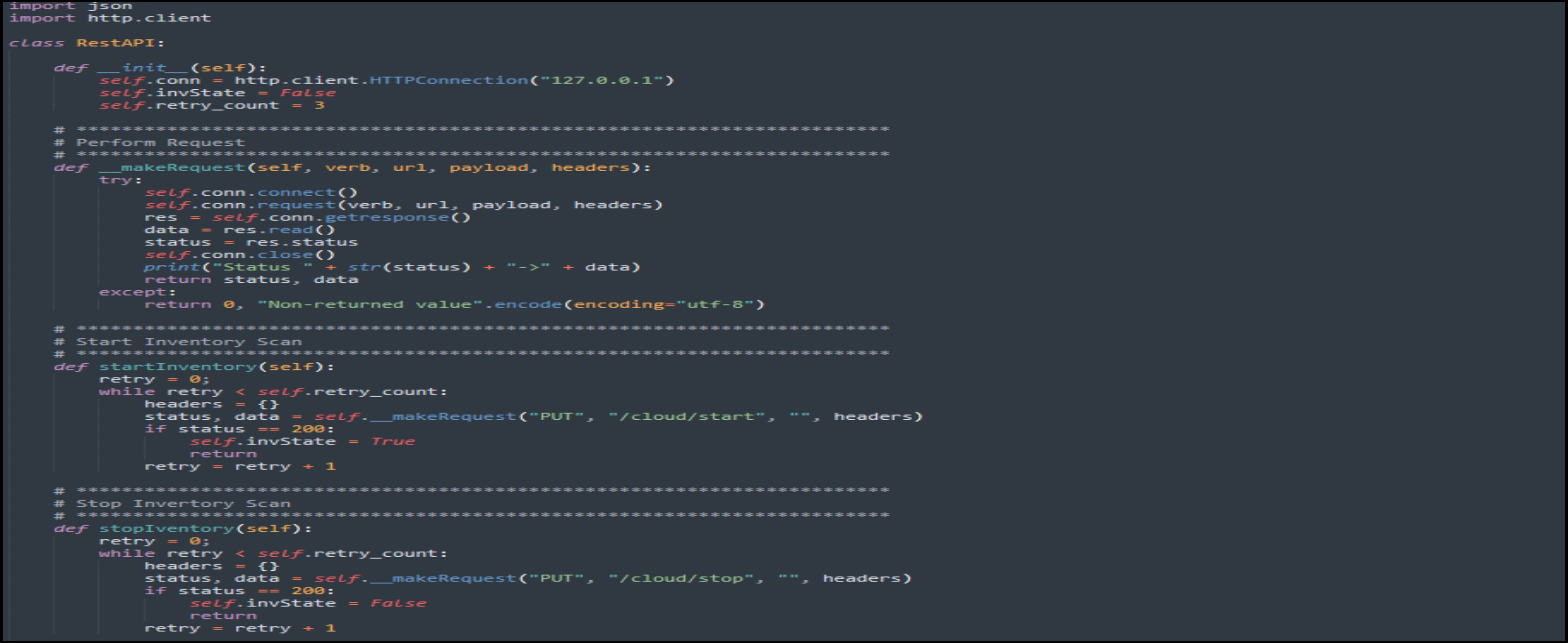
Zebra
DevCon 2023

- The Local Rest API is used to configure the RFID device
- The Local Rest API can also be used to interrogate the RFID device
- Local Rest API interface must be enabled in the Web Console
- Local Rest API's calls from an embedded application do not need authenticating

https://zebradevs.github.io/rfid-ziotc-docs/api_ref/local_rest/index.html

User Applications

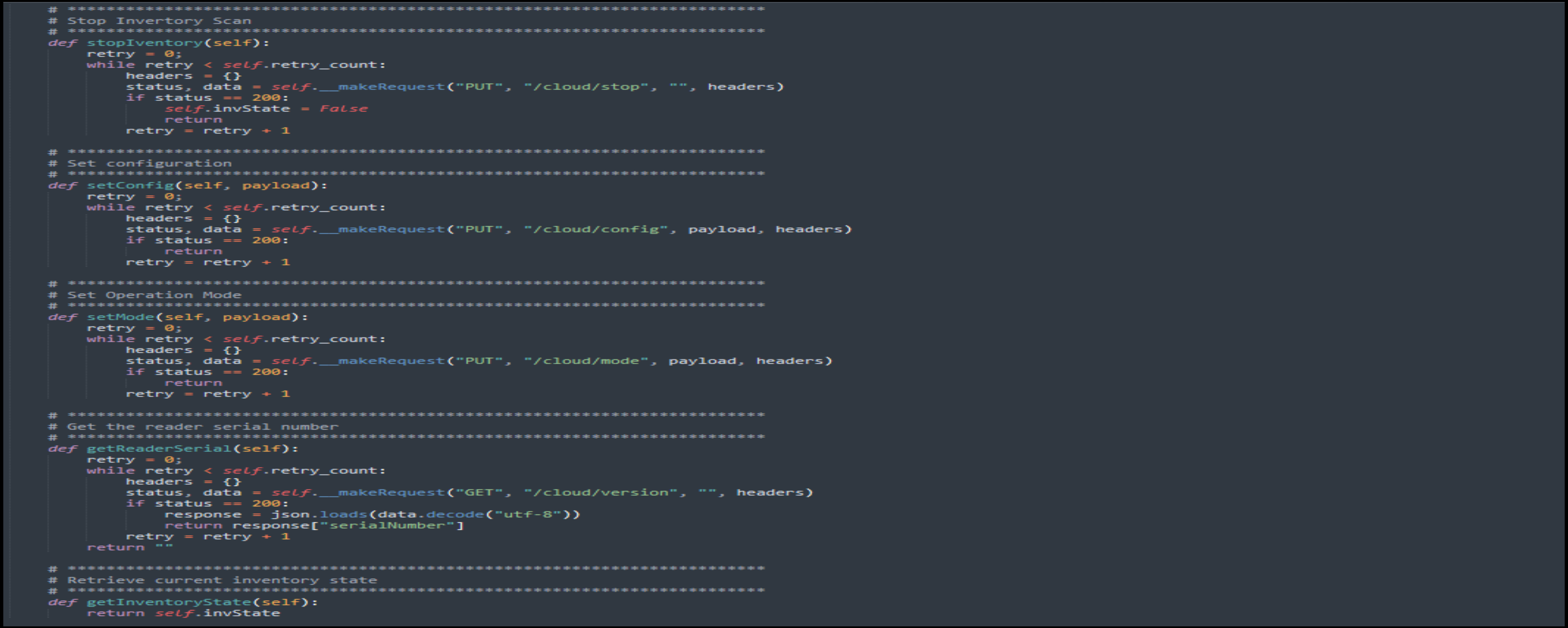
Rest API Interface for management



```python
import json
import http.client

class RestAPI:

    def __init__(self):
        self.conn = http.client.HTTPConnection("127.0.0.1")
        self.invState = False
        self.retry_count = 3

    # ************************************************************************
    # Perform Request
    # ************************************************************************
    def __makeRequest(self, verb, url, payload, headers):
        try:
            self.conn.connect()
            self.conn.request(verb, url, payload, headers)
            res = self.conn.getresponse()
            data = res.read()
            status = res.status
            self.conn.close()
            print("Status " + str(status) + "->" + data)
            return status, data
        except:
            return 0, "Non-returned value".encode(encoding="utf-8")

    # ************************************************************************
    # Start Inventory Scan
    # ************************************************************************
    def startInventory(self):
        retry = 0;
        while retry < self.retry_count:
            headers = {}
            status, data = self.__makeRequest("PUT", "/cloud/start", "", headers)
            if status == 200:
                self.invState = True
                return
            retry = retry + 1

    # ************************************************************************
    # Stop Invertory Scan
    # ************************************************************************
    def stopIventory(self):
        retry = 0;
        while retry < self.retry_count:
            headers = {}
            status, data = self.__makeRequest("PUT", "/cloud/stop", "", headers)
            if status == 200:
                self.invState = False
                return
            retry = retry + 1
```

User Applications

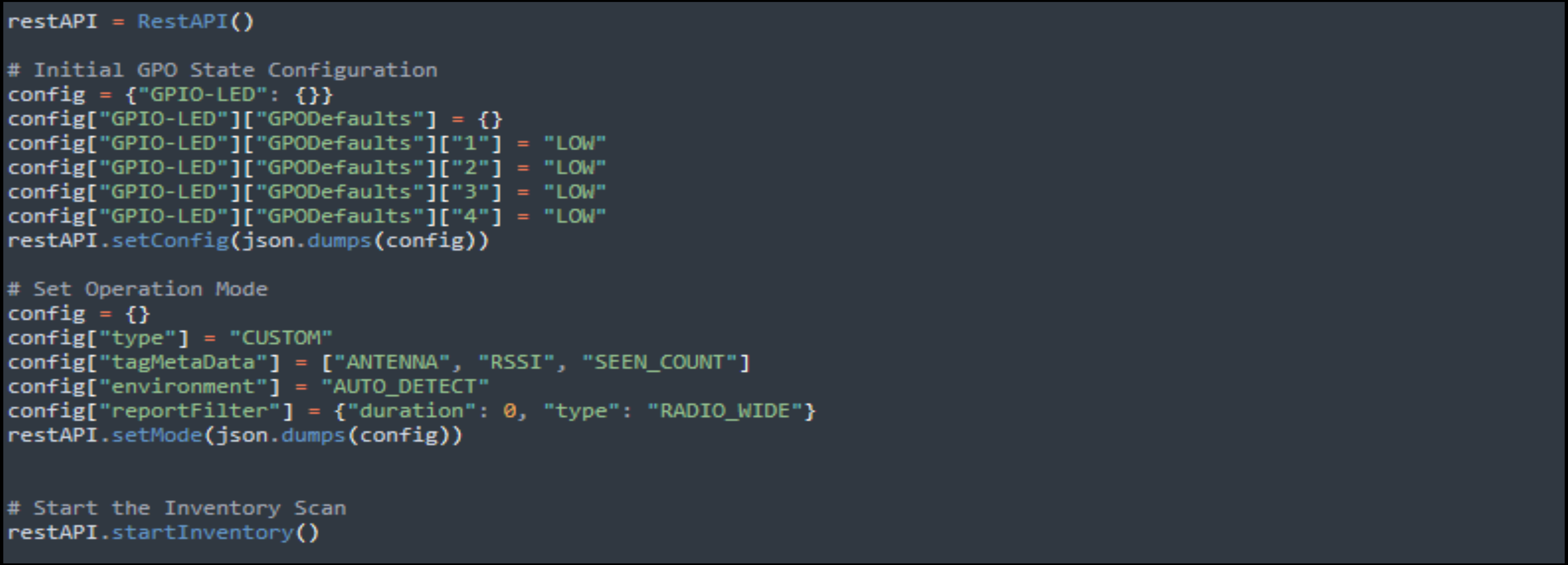
Rest API Interface for management



```
# ************************************************************************
# Stop Invertory Scan
# ************************************************************************
def stopIventory(self):
    retry = 0;
    while retry < self.retry_count:
        headers = {}
        status, data = self.__makeRequest("PUT", "/cloud/stop", "", headers)
        if status == 200:
            self.invState = False
            return
        retry = retry + 1

# ************************************************************************
# Set configuration
# ************************************************************************
def setConfig(self, payload):
    retry = 0;
    while retry < self.retry_count:
        headers = {}
        status, data = self.__makeRequest("PUT", "/cloud/config", payload, headers)
        if status == 200:
            return
        retry = retry + 1

# ************************************************************************
# Set Operation Mode
# ************************************************************************
def setMode(self, payload):
    retry = 0;
    while retry < self.retry_count:
        headers = {}
        status, data = self.__makeRequest("PUT", "/cloud/mode", payload, headers)
        if status == 200:
            return
        retry = retry + 1

# ************************************************************************
# Get the reader serial number
# ************************************************************************
def getReaderSerial(self):
    retry = 0;
    while retry < self.retry_count:
        headers = {}
        status, data = self.__makeRequest("GET", "/cloud/version", "", headers)
        if status == 200:
            response = json.loads(data.decode("utf-8"))
            return response["serialNumber"]
        retry = retry + 1
    return ""

# ************************************************************************
# Retrieve current inventory state
# ************************************************************************
def getInventoryState(self):
    return self.invState
```

User Applications

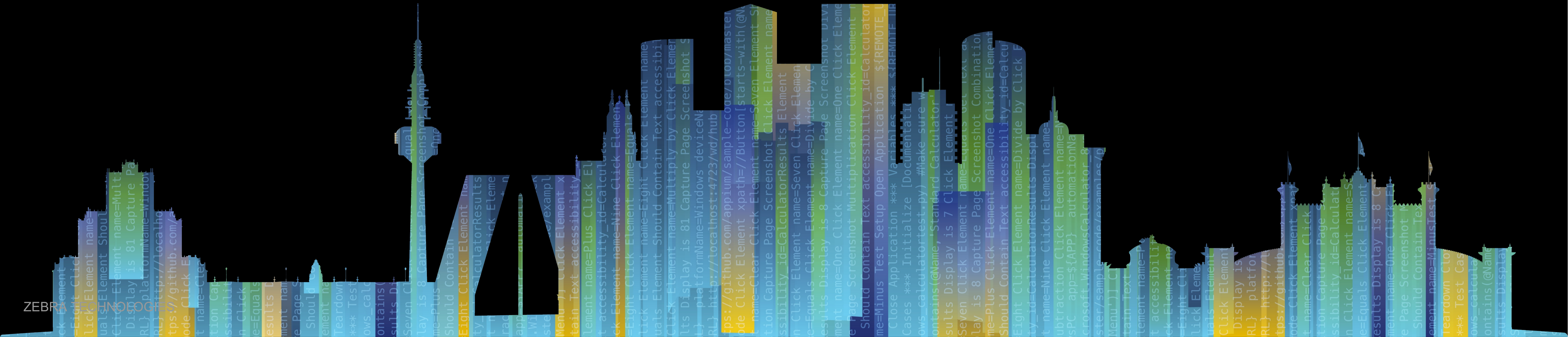
Rest API Interface for management



```
restAPI = RestAPI()

# Initial GPO State Configuration
config = {"GPIO-LED": {}}
config["GPIO-LED"]["GPODefaults"] = {}
config["GPIO-LED"]["GPODefaults"]["1"] = "LOW"
config["GPIO-LED"]["GPODefaults"]["2"] = "LOW"
config["GPIO-LED"]["GPODefaults"]["3"] = "LOW"
config["GPIO-LED"]["GPODefaults"]["4"] = "LOW"
restAPI.setConfig(json.dumps(config))

# Set Operation Mode
config = {}
config["type"] = "CUSTOM"
config["tagMetaData"] = ["ANTENNA", "RSSI", "SEEN_COUNT"]
config["environment"] = "AUTO_DETECT"
config["reportFilter"] = {"duration": 0, "type": "RADIO_WIDE"}
restAPI.setMode(json.dumps(config))


# Start the Inventory Scan
restAPI.startInventory()
```

Zebra
DevCon 2023

Packaging the application



Packaging

Zebra
DevCon 2023

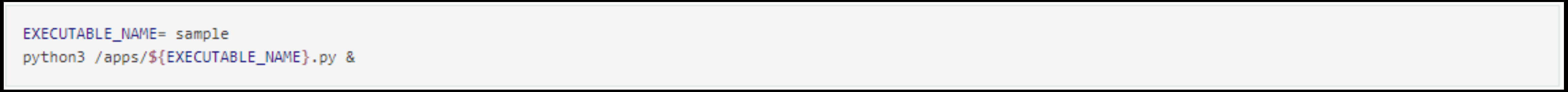
- Applications are shipped in Debian packages
- The Debian package must contain a start_ and stop_ script
- The Debian package also contains a control file
- Installation can be either through Web Console or Reader Management software

https://zebradevs.github.io/rfid-ziotc-docs/user_apps/packaging_and_deployment.html

Packaging

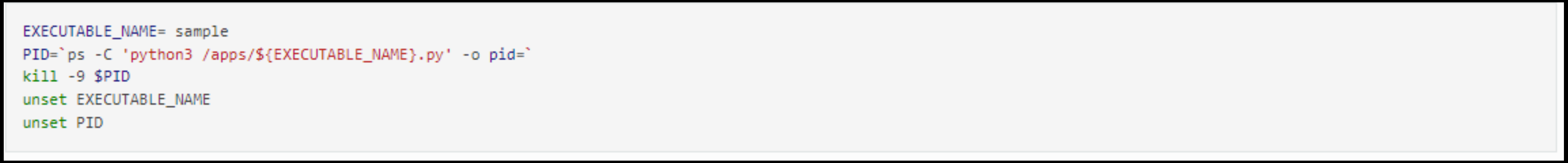
Example Start and Stop scripts

- start_sample.sh



```
EXECUTABLE_NAME= sample
python3 /apps/${EXECUTABLE_NAME}.py &
```

- stop_sample.sh

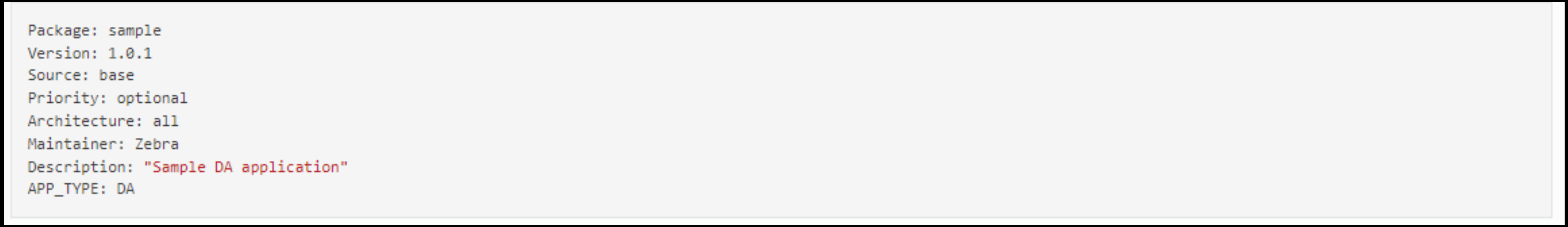


```
EXECUTABLE_NAME= sample
PID=`ps -C 'python3 /apps/${EXECUTABLE_NAME}.py' -o pid=`
kill -9 $PID
unset EXECUTABLE_NAME
unset PID
```

Packaging

Example control file

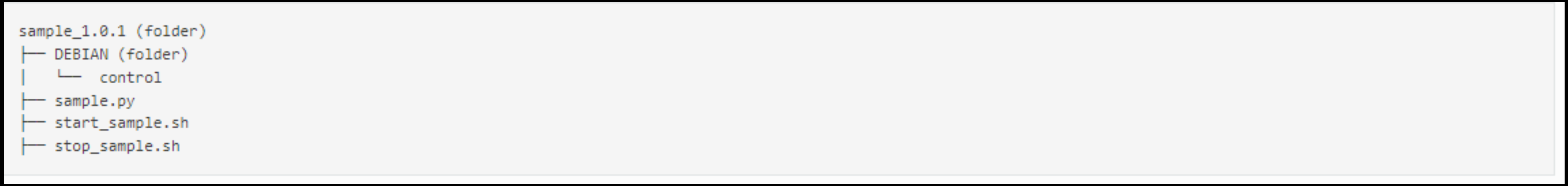
- Control



```
Package: sample
Version: 1.0.1
Source: base
Priority: optional
Architecture: all
Maintainer: Zebra
Description: "Sample DA application"
APP_TYPE: DA
```

Packaging

File Structure and building



```
sample_1.0.1 (folder)
├── DEBIAN (folder)
|    └──  control
├── sample.py
├── start_sample.sh
├── stop_sample.sh
```

- Building (Linux Only)

  dpkg-deb –build –Zgzip sample_1.0.1/

Zebra
DevCon 2023

Resources

Resources

Zebra DevCon 2023

- Zebra IoT Connector - https://zebradevs.github.io/rfid-ziotc-docs/
- Zebra Devs GitHub - https://github.com/zebradevs
- Zebra Devs Examples - https://github.com/ZebraDevs/RFID_ZIOTC_Examples

Questions

Zebra
DevCon 2023

ZEBRA

Thank You