IIOT DEVICE MANAGEMENT AT UGA

by

KEITH JAMES SCHNEIDER

(Under the Direction of WenZhan Song)

ABSTRACT

The Sensorweb research laboratory has done some amazing research regarding cybersecurity and Industrial Internet of things research. While we continue doing this research it is important to create and maintain workflows that allow for ease of use and create a standard operating procedure for device management. The sensorweb lab will be able to easily manage its devices using a continuous integration/continuous development solution as well as open-source software such as Thingsboard.io to generate new RaspberryPi images, create already interpreted executable files of their python code and manage their devices.

INDEX WORDS: Industrial Internet of Things, Microcontroller, Continuous Integration, Continuous Development, Python, Edge Computing

IIOT DEVICE MANAGEMENT AT UGA

by

KEITH JAMES SCHNEIDER

BS Electrical Engineering, United States Military Academy, 2016

A report Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2023

IIOT DEVICE MANAGEMENT AT UGA

by

KEITH JAMES SCHNEIDER

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

OVERVIEW

When developing Industrial Internet of things (IIoT) devices there are several considerations that need to be made with two of the primary considerations being: How can we scale the production of this, and how can we easily manage these devices? The primary goal of this project was to answer both of those questions for the BedDot project that is in development at the Sensorweb research lab. Another problem that needs to be addressed in a research university setting is to create an easily replicable solution that can be taught and learned easily by new researchers. To do this, I created a solution that is integrated with GitHub and Thingsboard.io. GitHub being the largest version control manager of git in the world and Thingsboard is an easily configurable open-source Internet of things device management platform. In this paper I will be providing a guide on the use of both GitHub Actions and the Thingsboard platform for the use of future researchers at UGA. First though I will give a brief overview of two applications and their merits.

1. GITHUB ACTIONS

The integration of GitHub in this solution solves several key problems in scaling and management of IIoT devices, especially devices that are built to run Python programs. The Continuous Integration/Continuous Development (CI/CD) solution offered by GitHub is called GitHub Actions and integrating GitHub actions into team workflows is crucial to maintaining a rapid development setting. There is a large ecosystem of open-source applications called “actions” that can be used to automate various tasks to manipulate your repository in some way. In this solution there is one primary action used [1] that was created to emulate both an ARM

CPU and the RaspberryPi operating system (RasPiOS) so you can automatically generate RasPiOS images that are manipulated in a manner that your project requires. The ability to manipulate the image prior to it ever being ran on a device allows you to automatically add all the software and packages the Sensorweb lab required as well as easily allowing the easy deployment of multiple devices at once from the same image so there is minimal time wasted copying files from computers to the IIoT device. The other advantage to being able to emulate both an ARM CPU and RasPiOS is that the GitHub action server can build an executable binary file from a python program so the IIoT device can run an already built single executable file rather than a python program. Running an executable file saves a significant amount of compute resources compared to the python interpreter that comes installed on the RaspberryPi saving in some instances 10-20 minutes of interpretation time that the device might need to start up the program. This executable was built with PyInstaller, another open-source tool. To use PyInstaller you must be in the exact same environment as the environment the built file will run on which is the other advantage of using this GitHub actions marketplace solution – The emulation of the RasPiOS image and using an ARM CPU. Finally, since there is a custom built image you can manipulate boot and runtime services creating an easy and replicable solution to have your programs run at startup. In summary – GitHub actions is used to generate RasPiOS images that are preloaded with all software from your repository and can contain prebuilt executable files for python code with services pre-installed to automatically manage the software running on the device.

2. THINGSBOARD

There are several considerations to make when considering a device management solution. After consulting with the Sensorweb research lab we decided to use Thingsboard.io.

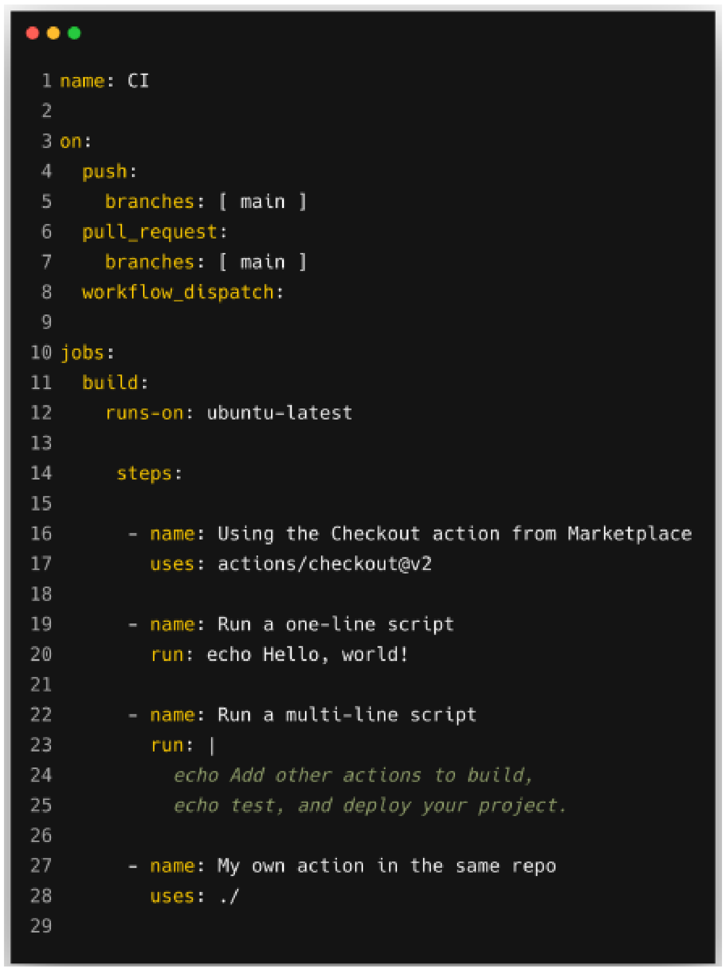
This is an open-source free-to-use platform that allows a user to set up their own server to run the management software and grants the ability to perform over-the-air software updates, manage tenants/customers of the device, allow tenants/customers to manage their own devices and issue remote procedure calls to devices. This solution was an obvious choice due to its ease-of-use and scalability features to provision devices and profiles for different tenants.

CHAPTER 2

GITHUB ACTIONS GUIDE

1. OVERVIEW

GitHub is one of the most famous hosting services for software development and version control using Git. It has a plethora of features that need no introduction, with a minor exception of one of its newer more powerful features, GitHub Actions. GitHub Actions was launched in 2018 and was created to help software developers to automate workflows - an obviously beneficial feature for a project with this scope. One of the most important features of GitHub Actions is the marketplace available to use custom actions that do many different desirable functions. Some examples of different functions include building docker containers, deploying those containers, and emulation of processors and different operating systems. At its most basic functionality however it brings automation directly into every developers repository with an easy to learn YAML syntax.



```
1  name: CI
2
3  on:
4    push:
5      branches: [ main ]
6    pull_request:
7      branches: [ main ]
8    workflow_dispatch:
9
10 jobs:
11   build:
12     runs-on: ubuntu-latest
13
14      steps:
15
16       - name: Using the Checkout action from Marketplace
17         uses: actions/checkout@v2
18
19       - name: Run a one-line script
20         run: echo Hello, world!
21
22       - name: Run a multi-line script
23         run: |
24           echo Add other actions to build,
25           echo test, and deploy your project.
26
27       - name: My own action in the same repo
28         uses: ./
29
```

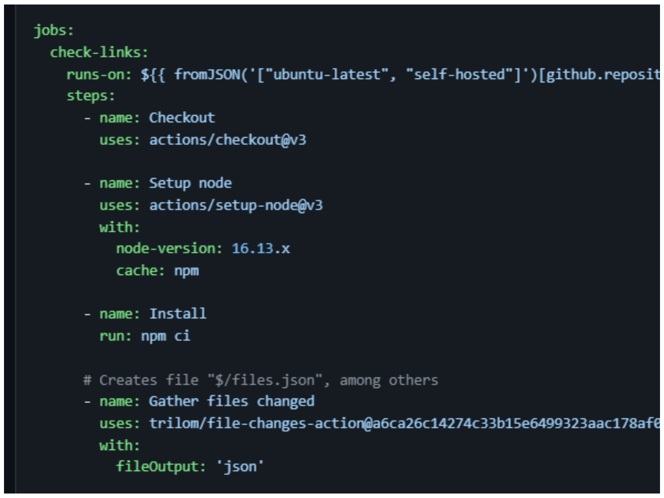
Fig. 1. A simple GitHub Actions ”Hello World” workflow.

Fig. 1, is a simple script that prints ”Hello, world!” in an Ubuntu server owned by GitHub. It may help to step through this program line-by-line so you can better understand YAML syntax. Line 1 is just naming the workflow - this feature is useful if you have multiple workflows which could allow easy differentiation between different scripts and allows different workflows to run in parallel completing different tasks at different times instead of running them all in sequence. Lines 3-8 are describing the types of Git related events the workflow should run on, this workflow runs on git pushes and pull requests to the main branch as well as on user request with the workflow dispatch trigger. You could also have the code run on a timed trigger on a schedule (a ”Cronjob”) with Cron syntax. Lines 10-28 define the actual jobs that will be running. On line

11 the first job is ”build” - this is just the name of the job. On line 12 the ”runs-on” chooses the runner the job will run on. A runner is a server that either the developer can set-up in a very simple to follow process on any device or you can use a GitHub hosted server (also called a runner) using Ubuntu, Linux or Mac OS. Line 14 begins the different steps the job will complete. Line 16 checks out the repository code onto the runner and is a necessary step on all workflows. Line 19 and 20 is the actual ”Hello World” script and it is just an echo command printing “Hello World” to the terminal. In their simplest form these workflows are bash scripts that can have some high powered functionality with line 20 being the only line in this script actually running a command. Line 23 shows how you can use the pipeline symbol to create a multi-line command.

3. GITHUB ACTIONS MARKETPLACE

The GitHub actions marketplace is a collection of open-source community created tools to help create more complex workflows while only needing to add a few lines of code to your workflow. The most popular actions are things like Linters, Security scanners for leaked credentials, and building Docker containers.



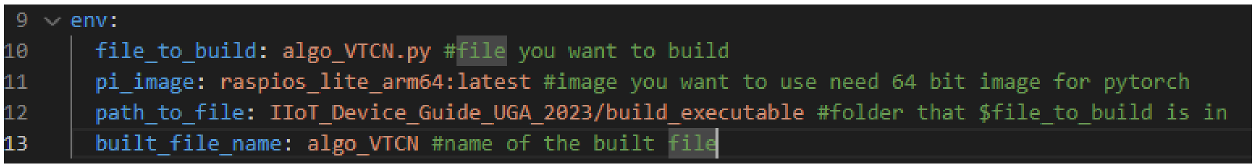
```yaml
jobs:
  check-links:
    runs-on: ${{ fromJSON('["ubuntu-latest", "self-hosted"]')[github.reposit
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Setup node
        uses: actions/setup-node@v3
        with:
          node-version: 16.13.x
          cache: npm

      - name: Install
        run: npm ci

      # Creates file "$/files.json", among others
      - name: Gather files changed
        uses: trilom/file-changes-action@a6ca26c14274c33b15e6499323aac178af0
        with:
          fileOutput: 'json'
```

Fig. 2: An example workflow using various actions to checkout their code and build a nodejs app

Several actions from the marketplace will be used in this project they are: actions/checkout@main, actions/upload-artifact@main,actions, /download-artifact@main, and pguyot/arm-runneraction@main. The first three actions are very simple and used in most workflows. In order they, checkout the repository code onto the runner so the code can be manipulated, upload any artifacts created during the workflow, and download those uploaded artifacts. The arm-runner-action is the most powerful of the four and allows for an incredibly simple implementation to build custom RasPiOS images and executable files for a Raspberry Pi. The arm-runner-action runs commands that allow the user to emulate an ARM processor as well as a host of common ARM operating systems including all raspios/rasbian images. Emulating the processor as well as the operating system is essential to using pyinstaller as it is not a cross compiler so the desired operating system and processor must be used to build the executable file.

4. TEMPLATE FILES

There are three template workflow files in this repository; executable.yml, startup_service.yml and all_together.yml. I will go through the executable and startup_service workflows first and at the end of this paper explain the all_together workflow after giving an overview of thingsboard. I will be skipping over the initial setup lines as seen in lines 1-9 in Fig. 1. You must define events for the workflow to trigger on for each workflow in those lines at the top of the file. There are more than 30 different possible events you could use and the most common are: push, pull, workflow dispatch, and schedule. For more information on these events and their uses see https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows.

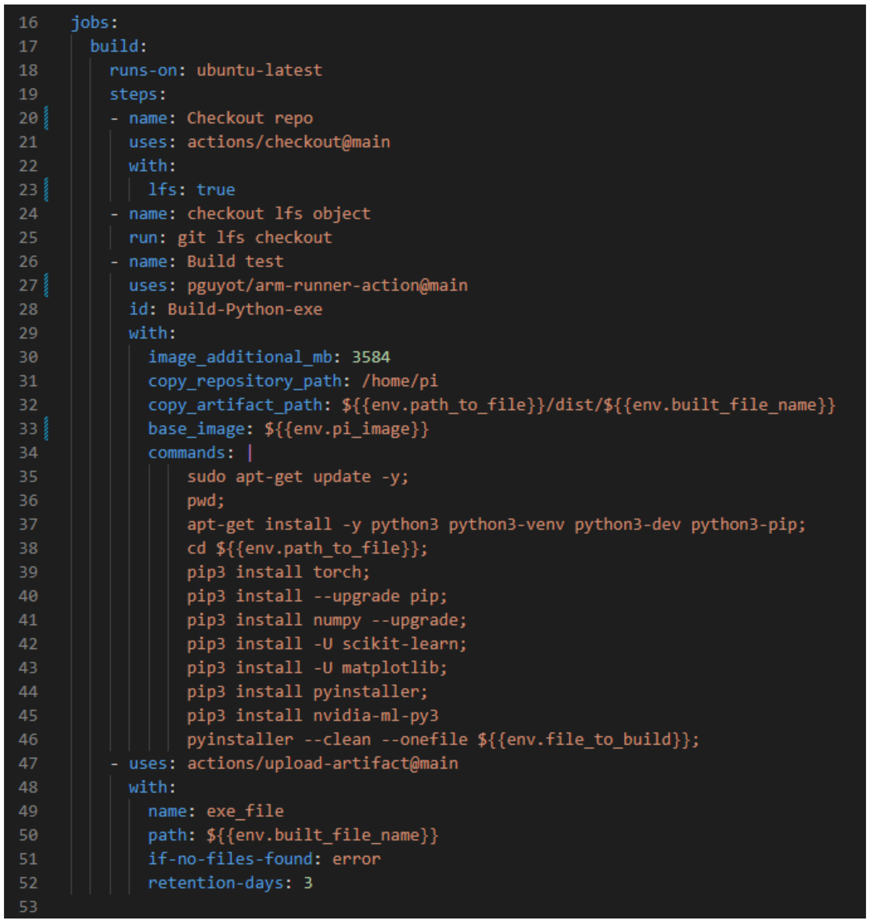


```
9  env:
10   file_to_build: algo_VTCN.py #file you want to build
11   pi_image: raspios_lite_arm64:latest #image you want to use need 64 bit image for pytorch
12   path_to_file: IIoT_Device_Guide_UGA_2023/build_executable #folder that $file_to_build is in
13   built_file_name: algo_VTCN #name of the built file
```

Fig. 3. Environment variables for a workflow

At the start of each job just below the initial setup you must name your environment variables. In fig 3. I have named 4 environment variables, the words in blue are the variable names and in the orange is the variable values. You can access these variables by using the syntax seen in line 32 or 33 of Fig. 4.

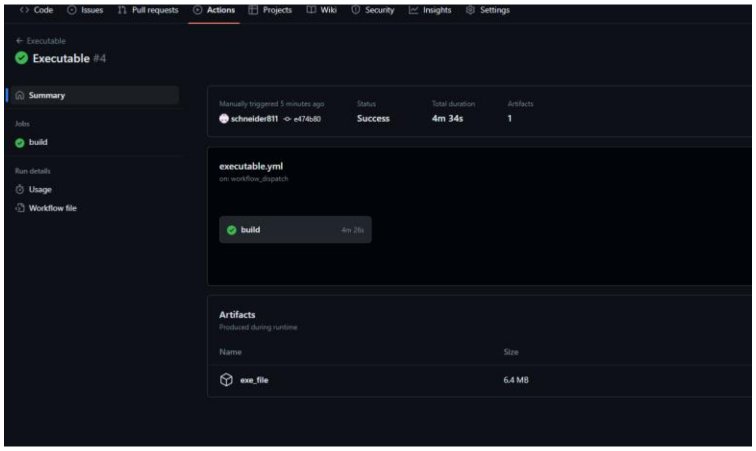
A. Executable.yml



```
16  jobs:
17    build:
18      runs-on: ubuntu-latest
19      steps:
20      - name: Checkout repo
21        uses: actions/checkout@main
22        with:
23          lfs: true
24      - name: checkout lfs object
25        run: git lfs checkout
26      - name: Build test
27        uses: pguyot/arm-runner-action@main
28        id: Build-Python-exe
29        with:
30          image_additional_mb: 3584
31          copy_repository_path: /home/pi
32          copy_artifact_path: ${{env.path_to_file}}/dist/${{env.built_file_name}}
33          base_image: ${{env.pi_image}}
34          commands: |
35              sudo apt-get update -y;
36              pwd;
37              apt-get install -y python3 python3-venv python3-dev python3-pip;
38              cd ${{env.path_to_file}};
39              pip3 install torch;
40              pip3 install --upgrade pip;
41              pip3 install numpy --upgrade;
42              pip3 install -U scikit-learn;
43              pip3 install -U matplotlib;
44              pip3 install pyinstaller;
45              pip3 install nvidia-ml-py3
46              pyinstaller --clean --onefile ${{env.file_to_build}};
47      - uses: actions/upload-artifact@main
48        with:
49          name: exe_file
50          path: ${{env.built_file_name}}
51          if-no-files-found: error
52          retention-days: 3
53
```

Fig. 4. An example script to build an executable from a python file.

Going step-by-step through this file lines 16-19 defines the jobs and the operating system you wish to use for the job. Lines 20-25 checkout the repository code so you can use or manipulate any of the code in your repository. Line 26-33 is the action and its commands that allow you to emulate a RasPiOS on the Linux server and execute the commands from line 34-46 as if you are executing them from a Raspberry Pi device. As stated previously, this is necessary to use pyinstaller as the application to build the desired executable file. When building the executable make sure your package manager is up to date and your python is up to date (lines 35-37). Then install the dependencies of your program onto the operating system as seen in lines 39-45. Once all the dependencies are installed you can just run pyinstaller as described on their website. Once pyinstaller is done running the action pguyot/arm-runner-action@main will automatically add the built file as an artifact (line 32) and GitHub will upload that artifact for you to download in your repository (lines 47-52). The file used in this example is just some example code that the Sensorweb lab uses and could be easily swapped for a different program. To swap to a different program, the environment variables at the top of the file need to be changed and lines 39-45 need to be changed to include the dependencies of the new file. To download the file, go to the run and click the filename under 'artifacts' as seen in Fig. 5.



Manually triggered 5 minutes ago	Status	Total duration	Artifacts
schneider811 -o- e474b80	Success	4m 34s	1

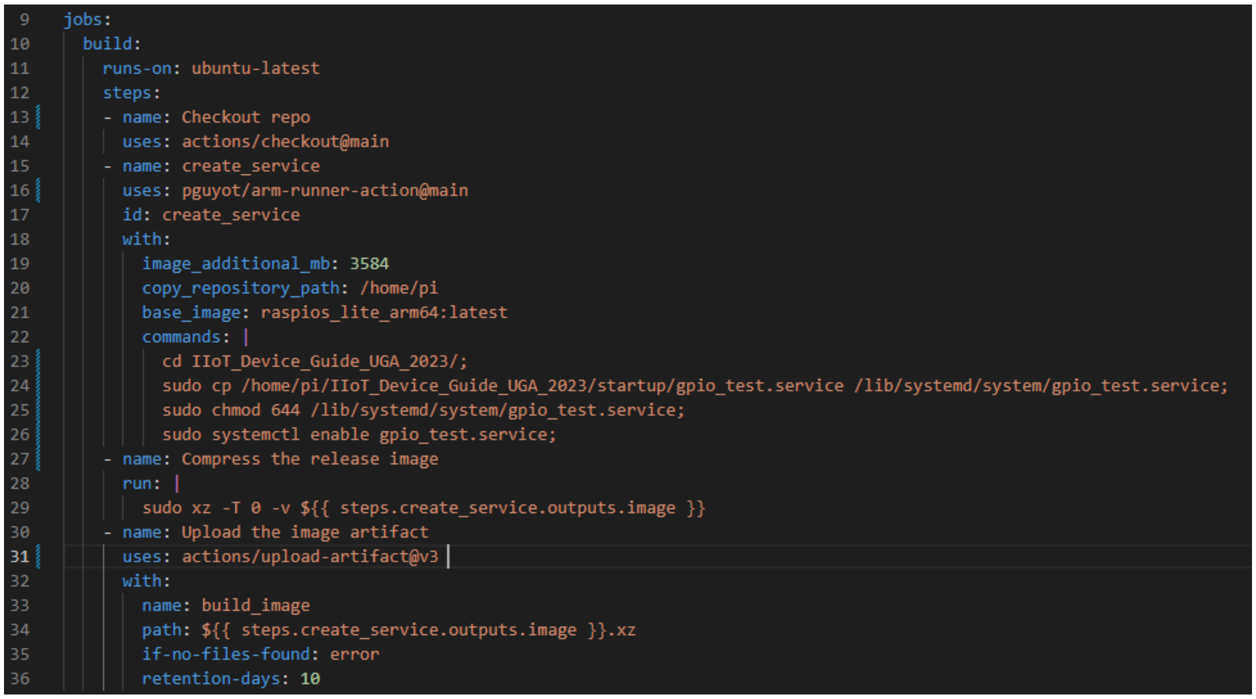
executable.yml
on: workflow_dispatch

Artifacts
Produced during runtime

Name	Size
exe_file	6.4 MB

Fig. 5

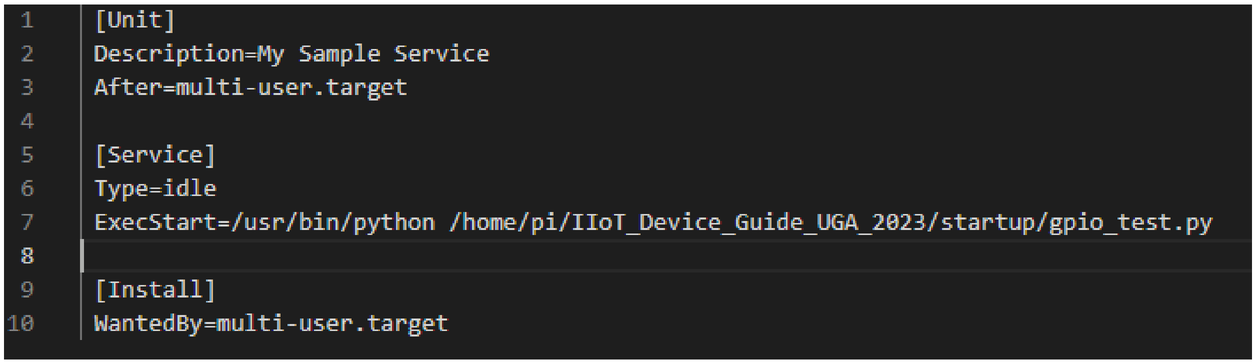
B. Startup_service.yml



```yaml
9   jobs:
10    build:
11      runs-on: ubuntu-latest
12      steps:
13      - name: Checkout repo
14        uses: actions/checkout@main
15      - name: create_service
16        uses: pguyot/arm-runner-action@main
17        id: create_service
18        with:
19          image_additional_mb: 3584
20          copy_repository_path: /home/pi
21          base_image: raspios_lite_arm64:latest
22          commands: |
23            cd IIoT_Device_Guide_UGA_2023/;
24            sudo cp /home/pi/IIoT_Device_Guide_UGA_2023/startup/gpio_test.service /lib/systemd/system/gpio_test.service;
25            sudo chmod 644 /lib/systemd/system/gpio_test.service;
26            sudo systemctl enable gpio_test.service;
27      - name: Compress the release image
28        run: |
29          sudo xz -T 0 -v ${{ steps.create_service.outputs.image }}
30      - name: Upload the image artifact
31        uses: actions/upload-artifact@v3
32        with:
33          name: build_image
34          path: ${{ steps.create_service.outputs.image }}.xz
35          if-no-files-found: error
36          retention-days: 10
```

Fig. 6 Startup_service.yml

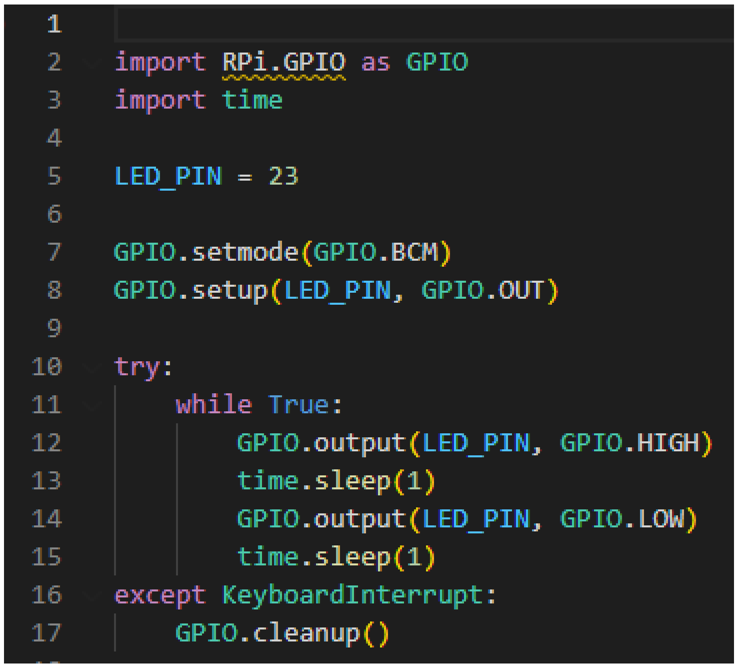
This workflow is very similar to the previous workflow except for the commands run after calling the arm-runner-action. In lines 23-26 this workflow is copying a file defined in the repository (Fig. 7) into the system files for the emulated raspberry pi. Then the server is giving the service permissions to run and enabling the service. This service is a simple script that will blink an LED on startup for your raspberry pi.



```ini
[Unit]
Description=My Sample Service
After=multi-user.target

[Service]
Type=idle
ExecStart=/usr/bin/python /home/pi/IIoT_Device_Guide_UGA_2023/startup/gpio_test.py

[Install]
WantedBy=multi-user.target
```

Fig. 7 Gpio_test.service file (line 24 fig 5.)

The code in Fig. 6 is just the standard format to define a service to run on a Linux machine. And this service is just running a simple LED blink function as can be seen in Fig. 8.



```python

import RPi.GPIO as GPIO
import time

LED_PIN = 23

GPIO.setmode(GPIO.BCM)
GPIO.setup(LED_PIN, GPIO.OUT)

try:
    while True:
        GPIO.output(LED_PIN, GPIO.HIGH)
        time.sleep(1)
        GPIO.output(LED_PIN, GPIO.LOW)
        time.sleep(1)
except KeyboardInterrupt:
    GPIO.cleanup()
```

Fig. 8 gpio_test.py

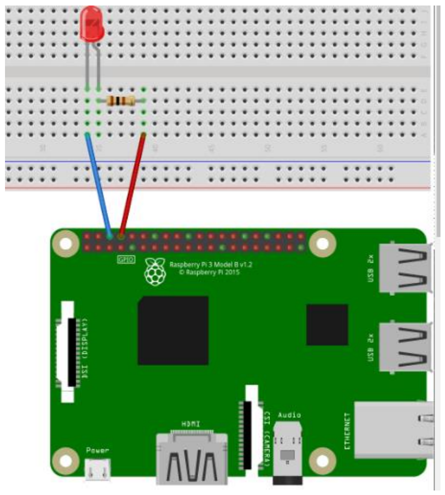


Fig. 9 Wiring scheme for gpio_test.py

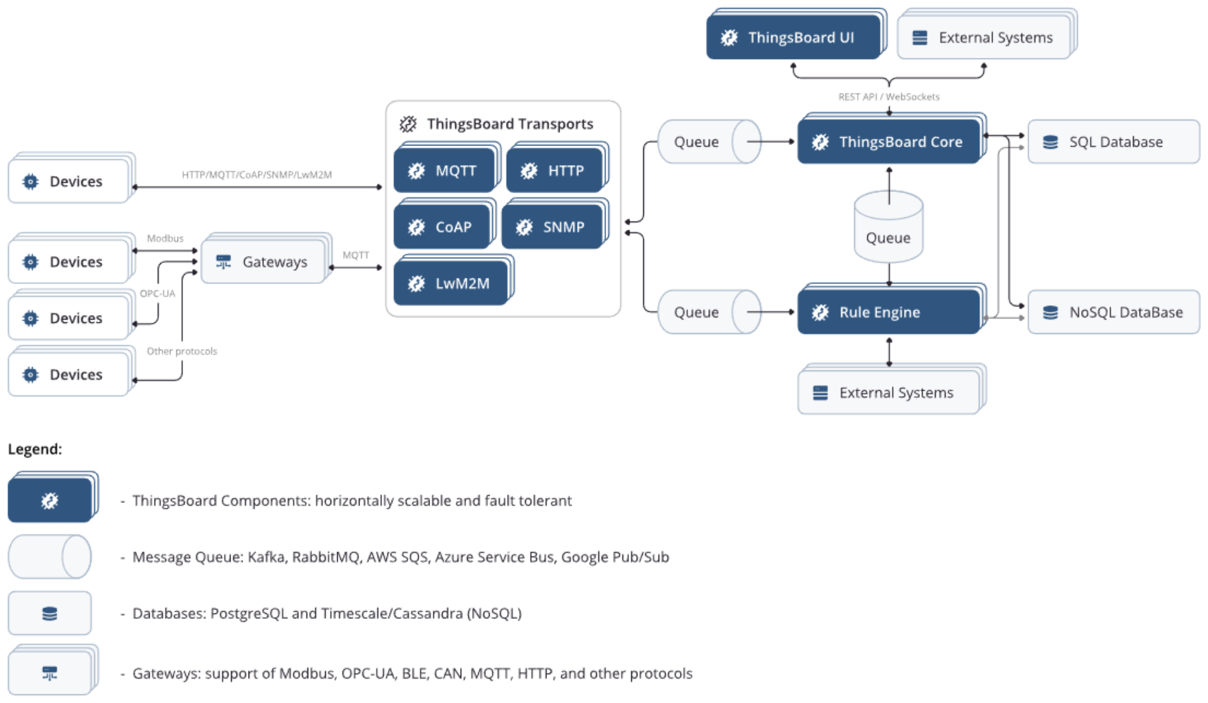
To test this workflow just run it through the workflow dispatch trigger to generate a new image and connect an LED to RaspberryPi GPIO pin 23 with a 330 Ohm resistor to not burn out the LED. The red wire in Fig. 8 is connected to pin 23 and the blue wire is connected to ground.

Once the workflow is done running you can download the image and use RasPi imager to flash the image onto an SD card and the service will automatically run on startup without you having to issue any commands to the device.

CHAPTER 3

THINGSBOARD OVERVIEW

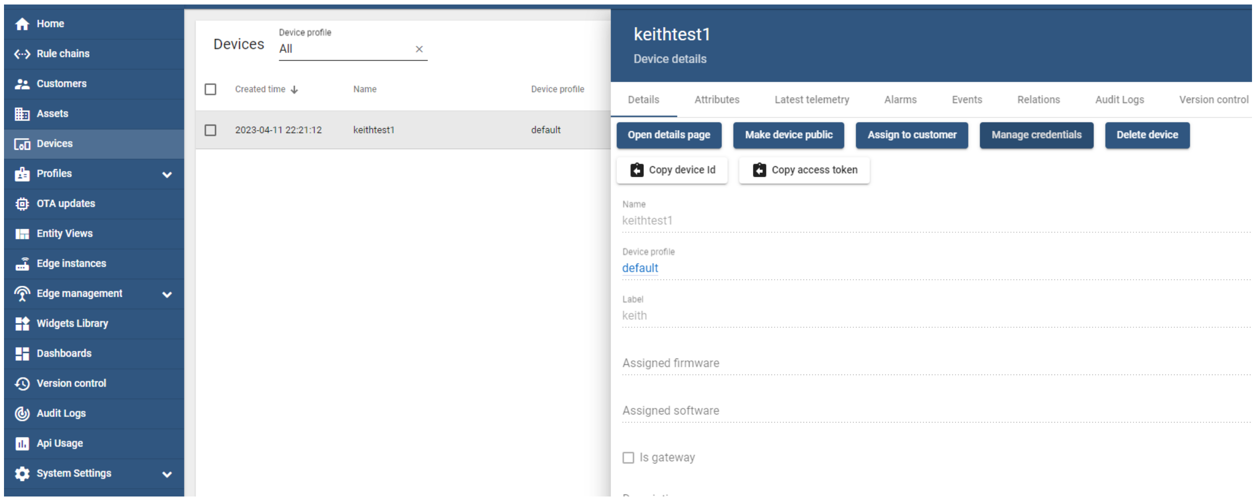
This portion of this report will be kept rather short as it is much easier to explain over video and show specific examples of how this website works so I will just briefly describe some of the features and how you can use them.



ThingsBoard UI
External Systems
REST API / WebSockets
ThingsBoard Transports
MQTT
HTTP
CoAP
SNMP
LwM2M
Queue
ThingsBoard Core
SQL Database
Devices
HTTP/MQTT/CoAP/SNMP/LwM2M
Modbus
Gateways
MQTT
OPC-UA
Other protocols
Rule Engine
NoSQL DataBase
External Systems
Legend:
- ThingsBoard Components: horizontally scalable and fault tolerant
- Message Queue: Kafka, RabbitMQ, AWS SQS, Azure Service Bus, Google Pub/Sub
- Databases: PostgreSQL and Timescale/Cassandra (NoSQL)
- Gateways: support of Modbus, OPC-UA, BLE, CAN, MQTT, HTTP, and other protocols

Fig. 10 Thingsboard Architecture

The sensorweb lab has built a thingsboard server that is accessible via the internet at: sensorweb.us:9090. In order to gain access to the website contact someone within the Sensorweb lab and they can provide you with an access code to create an account and test some of the functionality. The server architecture can be seen in Fig. 10 On the left you have your devices, in our case they are Raspberry Pi devices. These devices communicate via different protocols, in my video demonstration I chose the HTTP protocol as it seemed to be the simplest and require the least amount of work to implement. Once the device sends its information to the sensorweb Thingsboard server there is a check to see if the information being passed contains any rules that have been added to the rule engine or it just passes the information through to the SQL database on the server and is accessible through the GUI.



Devices

Device profile: All

Created time ↓	Name	Device profile
2023-04-11 22:21:12	keithtest1	default

keithtest1

Device details

Name: keithtest1

Device profile: default

Label: keith

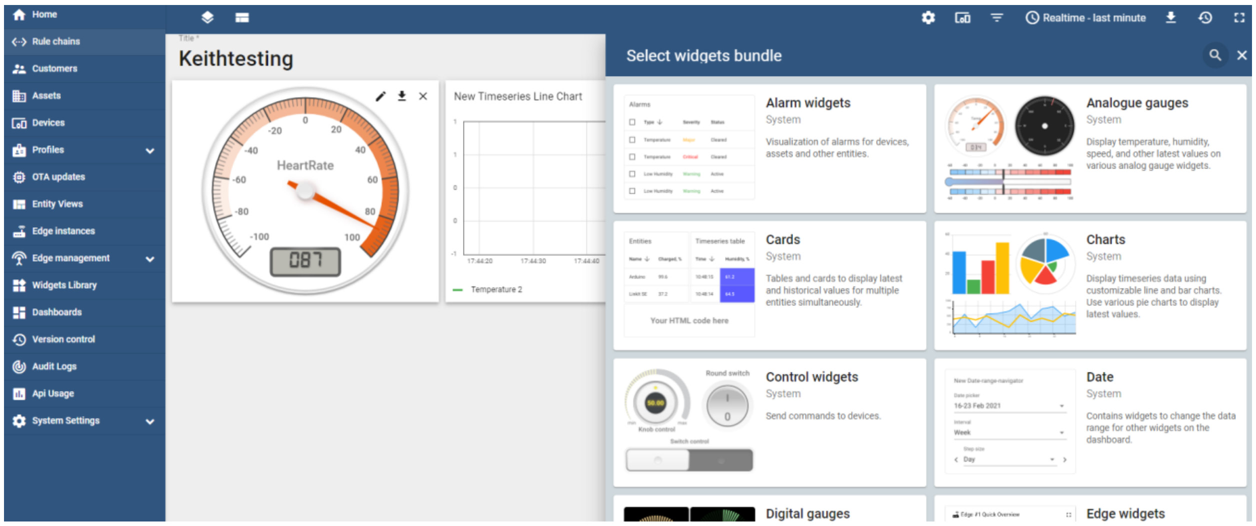
Assigned firmware

Assigned software

☐ Is gateway

Fig. 11. The device management page of Thingsboard

Once you have access to Thingsboard this is the primary page to manage your devices and do things such as manage its credentials, assigning firmware/software, view alarm activity and viewing other various details of the device. Once the device is provisioned you can use its access token in your programs so that it is able to communicate to the server and send information in a JSON format to be parsed by the server and added to the SQL database automatically. There will be more on this in the final section of this paper once we bring together the integration of GitHub actions with Thingsboard.



Visualization of alarms for devices, assets and other entities.

Display temperature, humidity, speed, and other latest values on various analog gauge widgets.

Tables and cards to display latest and historical values for multiple entities simultaneously.

Display timeseries data using customizable line and bar charts. Use various pie charts to display latest values.

Send commands to devices.

Contains widgets to change the data range for other widgets on the dashboard.

Fig. 12. Dashboard overview

Figure 12 is just a simple dashboard overview. Thingsboard has many different built in dashboards that allow you to easily select your data and display it in a GUI so all of the devices can be monitored from one location.

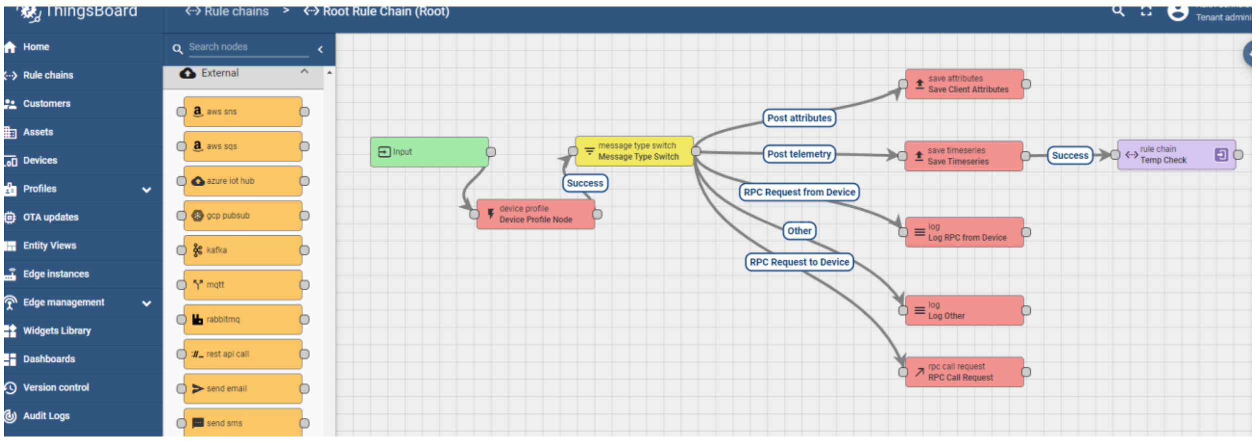


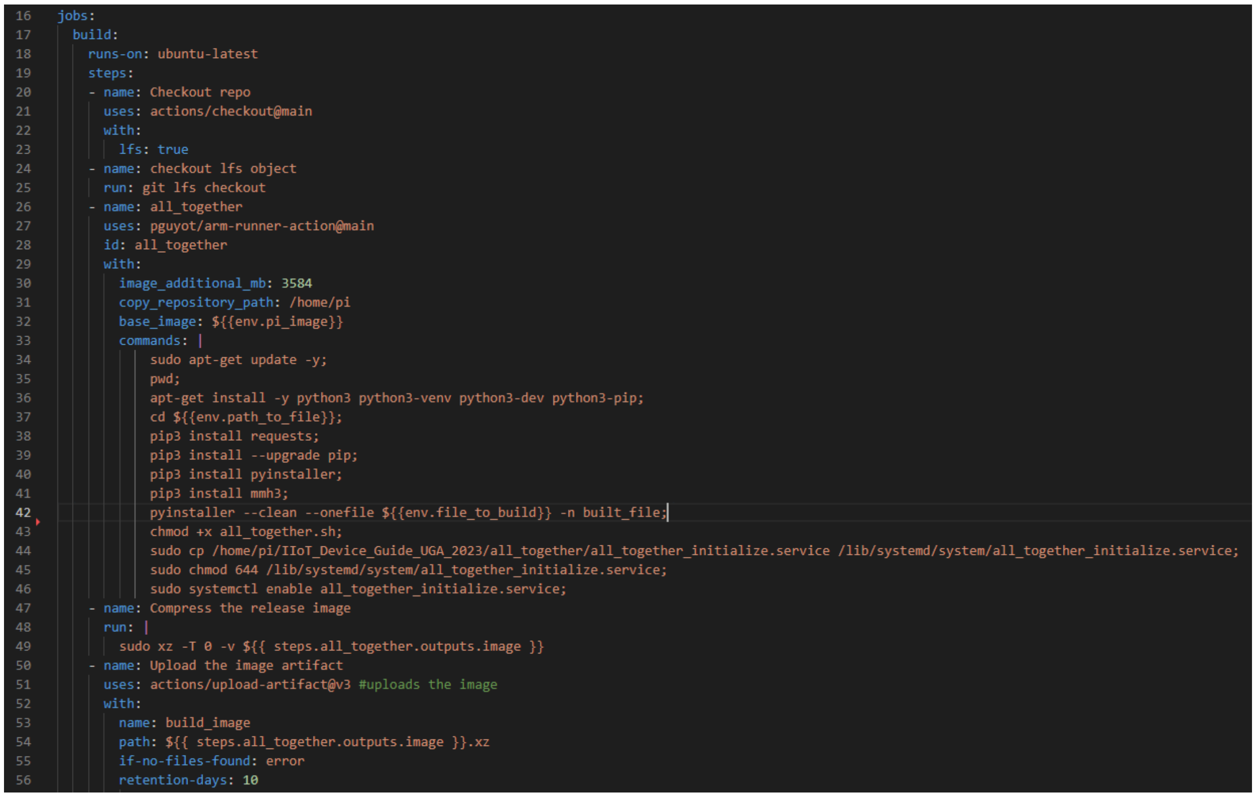
Fig. 13. Rule Chain overview

Fig. 13 is just a quick overview of the default rule chain assigned to a device in this guide. This is the same as the default for all initial root chains when creating a thingsboard server except I added an alarm rule chain where a guide on how to do so can be accessed here [2] The video for this portion of the guide can be found at [3].

CHAPTER 4

CONNECTING THE WORKFLOWS

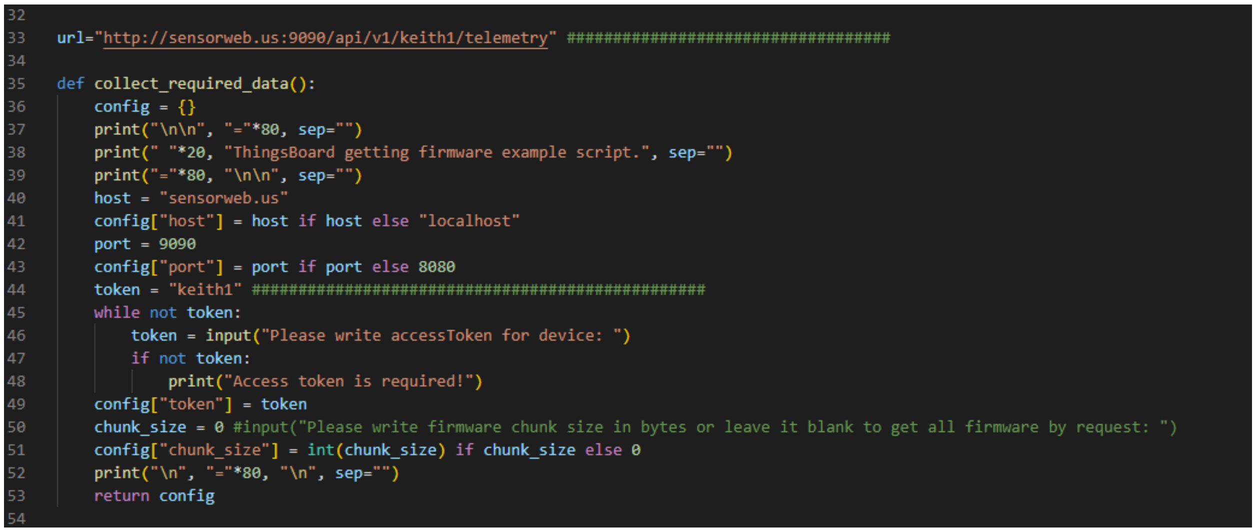
The final step in this implementation is to connect the two different workflows. This is done in the all_together.yml file as previously discussed in chapter 3.



```
16  jobs:
17    build:
18      runs-on: ubuntu-latest
19      steps:
20      - name: Checkout repo
21        uses: actions/checkout@main
22        with:
23          lfs: true
24      - name: checkout lfs object
25        run: git lfs checkout
26      - name: all_together
27        uses: pguyot/arm-runner-action@main
28        id: all_together
29        with:
30          image_additional_mb: 3584
31          copy_repository_path: /home/pi
32          base_image: ${{env.pi_image}}
33          commands: |
34              sudo apt-get update -y;
35              pwd;
36              apt-get install -y python3 python3-venv python3-dev python3-pip;
37              cd ${{env.path_to_file}};
38              pip3 install requests;
39              pip3 install --upgrade pip;
40              pip3 install pyinstaller;
41              pip3 install mmh3;
42              pyinstaller --clean --onefile ${{env.file_to_build}} -n built_file;
43              chmod +x all_together.sh;
44              sudo cp /home/pi/IIoT_Device_Guide_UGA_2023/all_together/all_together_initialize.service /lib/systemd/system/all_together_initialize.service;
45              sudo chmod 644 /lib/systemd/system/all_together_initialize.service;
46              sudo systemctl enable all_together_initialize.service;
47      - name: Compress the release image
48        run: |
49          sudo xz -T 0 -v ${{ steps.all_together.outputs.image }}
50      - name: Upload the image artifact
51        uses: actions/upload-artifact@v3 #uploads the image
52        with:
53          name: build_image
54          path: ${{ steps.all_together.outputs.image }}.xz
55          if-no-files-found: error
56          retention-days: 10
```

Fig. 14 All_together.yml file

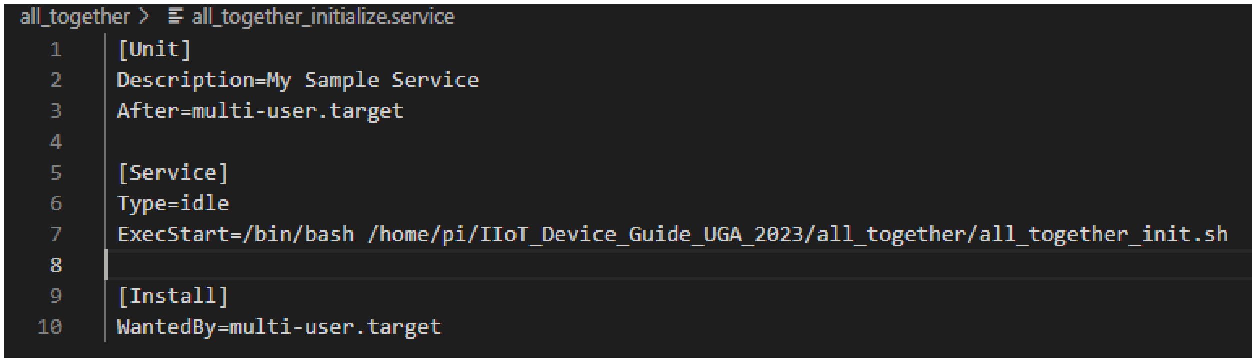
This YML file is a combination of the executable.yml and startup_service.yml files. In the workflow the server collects the required packages in lines 34-41 then builds the file in line 42. The file in line 42 is the primary file that can be used to manage over-the-air updates, collect data from sensors and send that data to the Thingsboard server. This file is an adapted version of the HTTP connection file thingsboard provides to check the server for software updates.



```python
32
33  url="http://sensorweb.us:9090/api/v1/keith1/telemetry" ###################################
34
35  def collect_required_data():
36      config = {}
37      print("\n\n", "="*80, sep="")
38      print(" "*20, "ThingsBoard getting firmware example script.", sep="")
39      print("="*80, "\n\n", sep="")
40      host = "sensorweb.us"
41      config["host"] = host if host else "localhost"
42      port = 9090
43      config["port"] = port if port else 8080
44      token = "keith1" #################################################
45      while not token:
46          token = input("Please write accessToken for device: ")
47          if not token:
48              print("Access token is required!")
49      config["token"] = token
50      chunk_size = 0 #input("Please write firmware chunk size in bytes or leave it blank to get all firmware by request: ")
51      config["chunk_size"] = int(chunk_size) if chunk_size else 0
52      print("\n", "="*80, "\n", sep="")
53      return config
54
```

Fig. 15 Excerpt from beginning of file

The first program in the file is the most important one for our purposes, it sets up the device and server information for communication to the thingsboard server. In order to be able to use this file lines 33, 40, 42, and 44 need to be edited to the information that pertains to your thingsboard server. If you are using the sensorweb thingsboard server, you only need to change the access token in line 44 for the device you have provisioned so that the server is assigning the sensor readings to the correct device. What may seem a bit more complicated is how the built file is being called to run on startup and checked to see if there are multiple instances of the program running and only allowing one to be run at a time. The first step in this process of running the built program is creating the all_together_initialize.service in lines 44-46.



all_together > all_together_initialize.service

```ini
[Unit]
Description=My Sample Service
After=multi-user.target

[Service]
Type=idle
ExecStart=/bin/bash /home/pi/IIoT_Device_Guide_UGA_2023/all_together/all_together_init.sh

[Install]
WantedBy=multi-user.target
```

Fig. 16 all_together_initialize.service file.

The file in Fig. 16 runs a shell script that can be seen in fig. 17. This script just creates a crontab that runs another shell script (Fig. 18) at start up and every minute. The script is written in this way so that every time it is called at start it overwrites the crontab (line 3) and then appends lines 4 to the crontab it just wrote. The script that it runs in Fig. 18 checks if the desired service is already running, if the service is running it just prints that it is running already and if it is not running the script executes the command to begin the script. This creates a constant check every minute the device is on so that if the connection to the server is lost or the program stops running it will automatically begin running again after one minute. In summary, Github creates and enables the service seen in Fig 16. When the Raspberry Pi is turned on, that service (Fig. 16) runs creating the crontab on the device seen in Fig. 17. The crontab just checks to see if the built file is currently running and if it is not running it will issue the command to begin running the executable version of the file seen in Fig. 15.

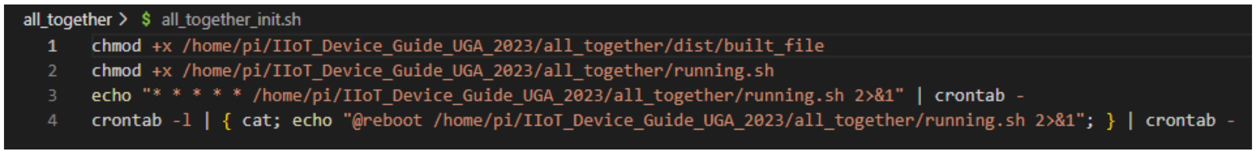

```
all_together > $ all_together_init.sh
1   chmod +x /home/pi/IIoT_Device_Guide_UGA_2023/all_together/dist/built_file
2   chmod +x /home/pi/IIoT_Device_Guide_UGA_2023/all_together/running.sh
3   echo "* * * * * /home/pi/IIoT_Device_Guide_UGA_2023/all_together/running.sh 2>&1" | crontab -
4   crontab -l | { cat; echo "@reboot /home/pi/IIoT_Device_Guide_UGA_2023/all_together/running.sh 2>&1"; } | crontab -
```

Fig. 17 all_together_init.sh

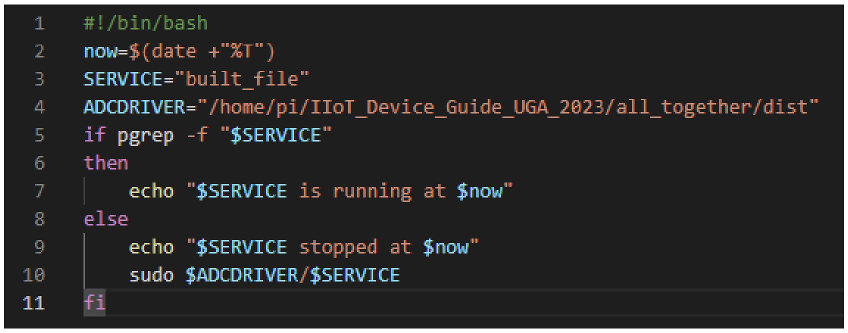

```
#!/bin/bash
now=$(date +"%T")
SERVICE="built_file"
ADCDRIVER="/home/pi/IIoT_Device_Guide_UGA_2023/all_together/dist"
if pgrep -f "$SERVICE"
then
    echo "$SERVICE is running at $now"
else
    echo "$SERVICE stopped at $now"
    sudo $ADCDRIVER/$SERVICE
fi
```

Fig. 18 running.sh

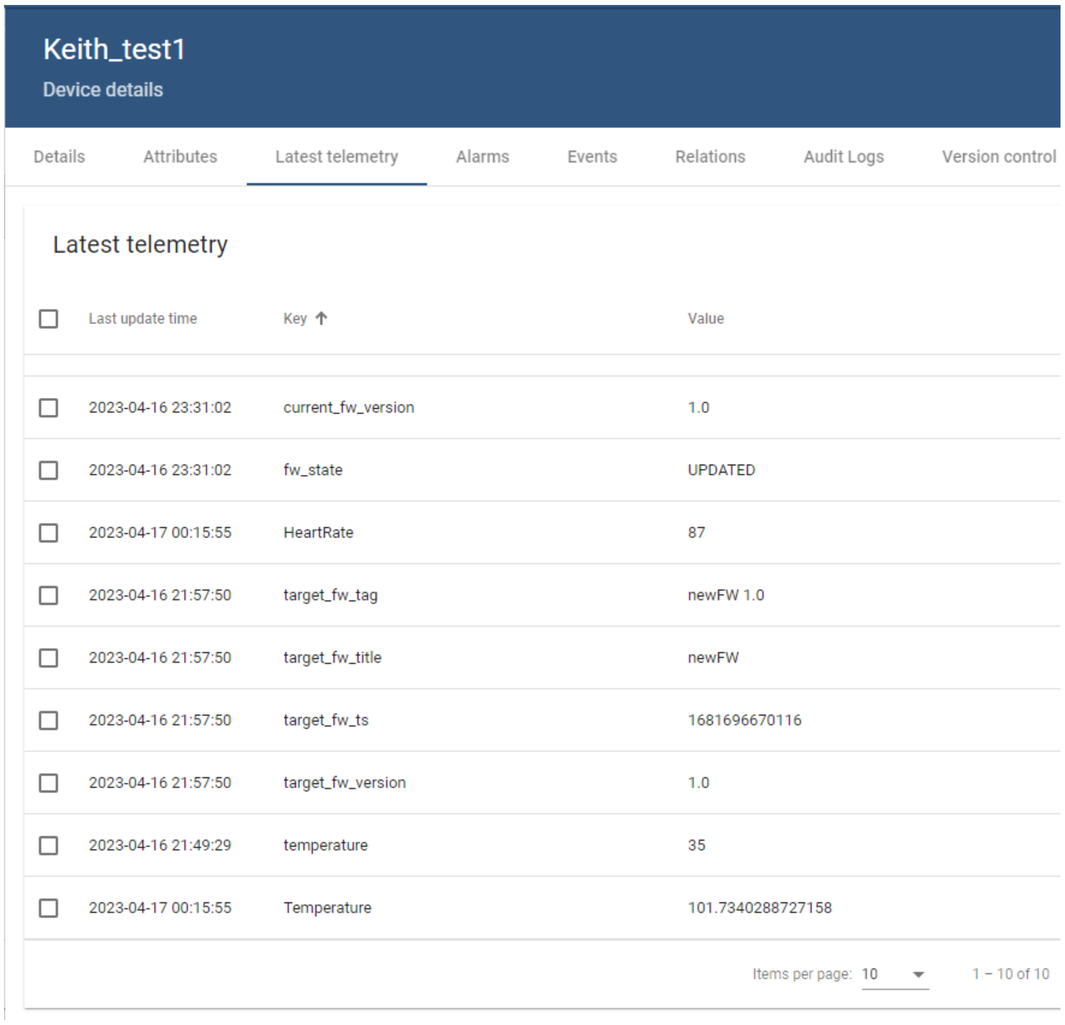
For a more in-depth and practical demonstration of all of this implementation I highly recommend you visit the repository and watch the accompanying videos located at [4] and [5].

To test this workflow make sure to edit lines 33, 40, 42, and 44 with the applicable information and then run the all_together.yml and download the image artifact that the run creates and run the image on your raspberry pi.

CHAPTER 5

RESULTS

Once the device is implemented on thingsboard and pushing data there are several new things to look at and features that are available to the user.



Latest telemetry

Last update time	Key ↑	Value
2023-04-16 23:31:02	current_fw_version	1.0
2023-04-16 23:31:02	fw_state	UPDATED
2023-04-17 00:15:55	HeartRate	87
2023-04-16 21:57:50	target_fw_tag	newFW 1.0
2023-04-16 21:57:50	target_fw_title	newFW
2023-04-16 21:57:50	target_fw_ts	1681696670116
2023-04-16 21:57:50	target_fw_version	1.0
2023-04-16 21:49:29	temperature	35
2023-04-17 00:15:55	Temperature	101.7340288727158

Items per page: 10 1 – 10 of 10

Fig. 19 Device Management page

Now that the device has been integrated into thingsboard you can access the device on the device management page and see the latest telemetry and update the software/firmware through an over-the-air update.

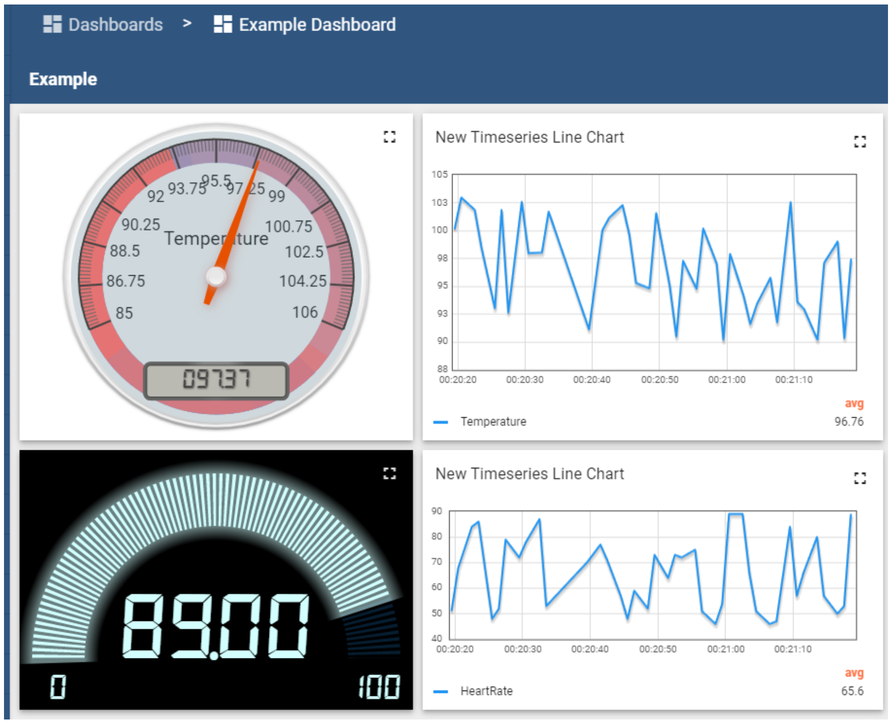
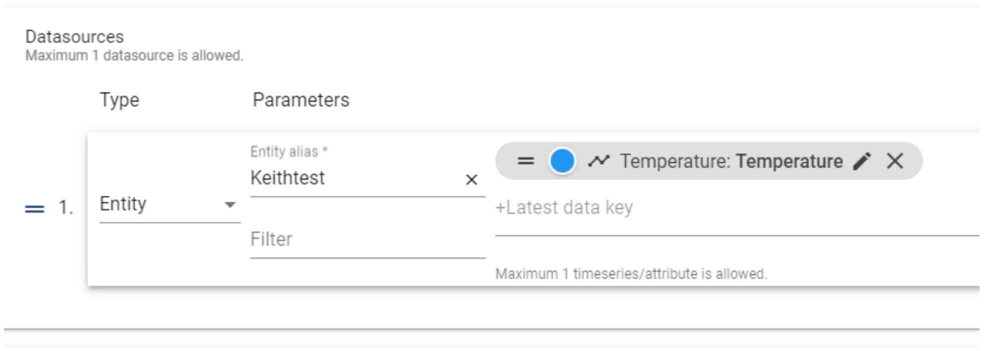

Dashboards > Example Dashboard
Example
Temperature
09737
New Timeseries Line Chart
Temperature
avg
96.76
89.00
0
100
New Timeseries Line Chart
HeartRate
avg
65.6

Fig. 20 Example dashboard for monitoring Patient Vital Signs



Datasources
Maximum 1 datasource is allowed.

Type	Parameters	
1. Entity	Entity alias * Keithtest	Temperature: Temperature
	Filter	+Latest data key
		Maximum 1 timeseries/attribute is allowed.

Fig. 21 How to create a new widget on the dashboard

Also once the device is provisioned and pushing information to the management server the user can now create their own dashboard and customize it easily through either the available open source widgets or creating their own. Fig. 20 shows an example dashboard containing the test heartrate and temperature sensor readings that are being pushed to the server. You can select from many different widgets to have it displayed as either a gauge such as the two displays on the left or as line charts showing the time series data for any range of time. In order to create a widget you just edit the dashboard to add a new widget, create a new entity alias, select Single Entity, Device and then the device with the data you want to display on the dashboard.

CHAPTER 6
CONCLUSION

In my attempts to bring together these two systems to create a cohesive and functional implementation I discovered many bugs/errors that were incredibly difficult to troubleshoot and solve. The thing that is likely to cause the most errors is the executable.yml and pyinstaller. Pyinstaller is an already fragile application that can require very specific installations of packages and dependencies to be able to build the binary. When you couple that with having to conduct test runs through a GitHub server that is emulating a Rasbian image and ARM CPU can lead to a lot of troubleshooting time. The simplest advice I can give to anyone trying to build files automatically in this manner is that whatever package or dependency is giving you errors in the implementation you should try to revert to an older installation of the package that is hopefully compatible with Pyinstaller. Additionally, the several step implementation described in chapter 4 is done this way because it was not possible to remove the service and just implement the crontab. The way that RasPi imager works will wipe previously created crontabs when it is formatting the built image onto the SD card but it will not reformat the service folders that are created in the workflow. Ultimately, CI/CD and device management tools such as the ones described in this paper all come with their flaws and bugs that need to be worked out. Once those bugs have been fixed though the types of applications available through the open-source community allow for some powerful and highly customizable workflows. These workflows can unlock many new features of programs you are already using and can increase productivity in any type of research that involves coding, not just the specific device management solution presented in this paper.

REFERENCES

[1] https://github.com/pguyot/arm-runner-action

[2] https://thingsboard.io/docs/user-guide/rule-engine-2-0/tutorials/create-clear-alarms/

[3] https://www.youtube.com/watch?v=2cG1O7QNw_Q

[4] https://youtu.be/oG-rT_H-thw

[5] https://github.com/schneider811/IIoT_Device_Guide_UGA_2023