Kingdom of Saudi Arabia
Ministry of Higher Education
Princess Nourah bint Abdulrahman University
Faculty of Computer & Informaion Science

Graduation Project I

المملكة العربية السعودية
وزارة التعليم العالي
جامعة الأميرة نورة بنت عبد الرحمن
كلية علوم الحاسب والمعلومات

مشروع تخرج ١

ONLINE HOME AUTOMATION CONTROL SYSTEM

By:

436200058 Abeer Ahmed Ezzat
436200063 Doha Nidal AlZouhbi
437004005 Mona Saud AlKhathlan
437004100 Nouf Abdullah AlDajani
437004875 Reem Ali AlGhamdi
436006939 Sarah Khalid AlHussain

Supervised By:
Dr. Abeer Mahmoud

A Graduation Project Report Submitted to
College of Computer and Information Sciences at PNU
in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science
in
Computer Sciences

CCIS, PNU
Riyadh, KSA
1440 - 1441H

Acknowledgment

First and foremost, we thank Allah the Almighty for always blessing us and for helping us complete this project on time. We would like to express our special thanks of gratitude to our project supervisor Dr. Abeer Mahmoud for her continuous support, motivation, and for always commenting on our best qualities and enhancing them. Without her direction and proper guidance, this project would not have succeeded. We thank her for her unwavering support. Not to forget our loving families and supporting friends for helping us sticking to the schedule. We also would like to thank Princess Nourah bint Abdulrahman University. Last but not least we want to thank anyone who has helped us by providing information, insight, comments or facilities that we needed to complete our project. Our heartfelt thanks to everyone we have mentioned.

Contents

List of Tables

List of Figures

List of Symbols & Abbreviations

Symbols & Abbreviations	Meaning
API	Application programming Interface
GPIO	General Purpose Input Output
GUI	Graphical User Interface
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
LED	Light Emitting Diode
REST	Representational State Transfer
SDLC	Software Development Life Cycle
SQL	Structured Query Language
UI	User Interface

Table 1: List of Symbols & Abbreviations

Keywords

Keyword	Definition
Raspberry Pi	low cost, credit-card sized computer[1].
Linear solenoid	type of electromagnetic actuator that converts an electrical signal into a magnetic field producing a linear motion[2].

Table 2: Keywords

Abstract

The aim for this project is to control light buttons, air conditioners, television or other home appliance regardless of the person's location. The methodology is simple: an android app will send controlling requests to a web server. Raspberry Pi will be getting all the new requests from the server, processing it accordingly and controlling the hardware components connected to it. Such a system will allow someone in the United States to turn the lights in their house in Saudi Arabia on. However, an active connection to the internet must be present all the time.

CHAPTER NO. 1
INTRODUCTION

1 Introduction

1.1 Problem statement & Significance

With the recent very rapid progress in technology and automation, and towards easier daily life tasks, there has become a need for remote control of almost all possible aspects of living. Especially the house appliances that surround us,that allow focusing on main work of each day.

Some examples we have already encountered and used in our daily lives include using apps to control a cleaning robot or adjust the heating in the house or switch the house lights on or off. For the latter, there have been many applications that can do that. However, they all work in a small set of devices and sensors. It is necessary for such applications to exist, as a service like this would be important for many people. An example is working moms who are outside the house and want to switch the lights on at a certain time to wake their children up. Another example is pet owners who need to have UV lights switched on for their pets at certain times of the day but cant do so immediately and so on.

However, the main challenge in creating a device to solve this problem is where the idea of IoT (Internet of Things) comes in; learning how to control this device through the Internet from afar, rather than being controlled by infrared rays locally as is the case with most similar applications.

1.2 Proposed Solution

This project propose building an intelligent application that is based on IoT techniques. The created app should enable the user, by clicking on the appropriate buttons, to control a physical apparatus that needs to be pushed to work, such as lights buttons. This will be done by designing and creating an Android application, then using a small laptop, called Raspberry Pi, to control a small piece that will be pushed forward (on command) to switch the light on or off, the API is a web application hosted on a server.

1.2.1 Aims

At the end of this project, we intend to achieve the following aims:

- Learn how to design a mobile application using previously learned and new knowledge
- Learn how to invoke a web API and use it in our application
- Learn Python programming language to control Raspberry Pi
- Learn Flask web micro-framework

1.2.2 Goals

At the end of this project, we expect to deliver:

- An Android application with a user friendly, simple, clear interface with buttons to control a LED and linear solenoid.
- A physical apparatus composed of the Raspberry Pi connected to and controlling the piece.
- A web application following REST architecture, managing user requests and Raspberry Pi's responses.

1.3 Project Domain & Limitation

1.3.1 Domain

Although the application will be available for all kinds of users to use, we expect that the ones who would make the most use of it would be employees who have long working hours and would need to be able to remotely control appliances in their homes, especially lights.

The critical piece in all of this application would be the linear solenoid actuator i.e. the small electrically controlled piece that would be placed very close to the light switch and would, on command, spring forward to press on the switch to turn it on or off.

1.3.2 Limitation

The main limitation of the application is that it will be able to control only a limited type of home appliance. Mainly things than could be pushed to work. A much more advanced application would be able to control most of the other appliances, such as controlling an Air conditioner if the owner is outside, or a timer controlled coffee maker.

1.4 Gantt Chart

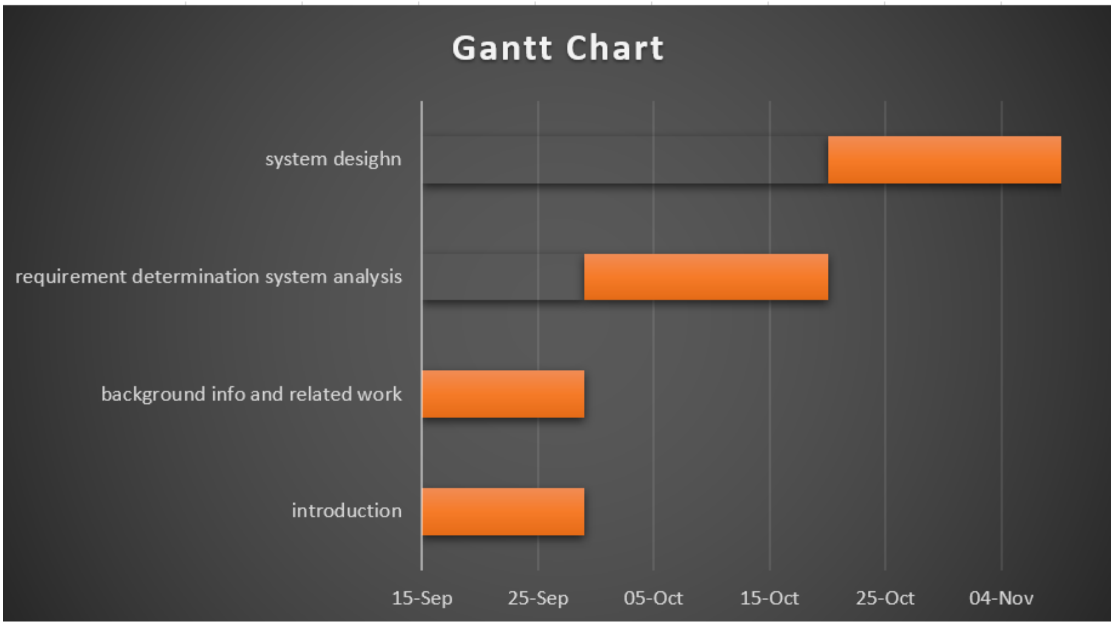


Figure 1.1: Gantt Chart

CHAPTER NO. 2
BACKGROUND INFORMATION & RELATED WORK

2 Background Information & Related Work

2.1 Background Information

2.1.1 IoT

The internet of things (IoT) is one of the most important technological developments in the last decade. Do you wonder were the term IoT came from? It was coined by Kevin in 1999 [3]. This means we can describe IoT as the ability to connect as many things by the internet without humans help by using technologies such as cloud computing, Radio Frequency Identification (RFID), wireless communication, sensors, Internet protocol, ultra-low-power processors and others [4].

2.1.1.1 IoT Architecture

It is basically what IoT is made of. It contains three layers; Perception layer, Network layer, and Application layer. So, to know these layers better we need to know the definition of them.

- **Perception layer:** is responsible for perceiving and identifying objects or things in the environment [5].
- **Network layer:** is responsible for receiving and transmitting data between layers [6].
- **Application layer:** is the interface for all previous layers used to process and transport data to provide services to the users [7].

2.1.1.2 IoT Applications

There are different kinds of applications that make life easier and more secure. These are some example of them:

- **Automobiles:** by IoT technology, it makes cars smarter and works to improve safety on the road by helping cars detect obstacles and assist braking or adapt their speed to the flow of traffic. Also it helps protect the environment by reducing fuel consumption, etc [8].

- **Smart Home:** You can control your entire home with a touch screen or your smartphone whether you are inside or outside a home, for example, you can lock or open your doors, turn on/off the light and air conditioner, etc [9].

- **Healthcare:** IoT has an important role in healthcare applications by providing the ability to easily monitor and manage patient health without having to manually visit each patient the doctor can give a remote diagnosis to provide quality care more quickly and manage the health care environment more efficiently [6].

2.1.2 Hardware

- **Raspberry Pi:** a small general purpose computer. All hardware components will be connected to it. An active connection to the internet is needed for it to fetch data from the server[1].

- **Ubuntu Web Server:** hosts the web application. Digital Ocean severs[10] were chosen for this project.

- **LED:** since the hardware components controlled depends heavily on the user needs, this project main aim will be controlling a small LED. LED stands for light-emitting diode[11]. Basically a small light source.

- **Linear Solenoid:** once the LED works, linear solenoid will be installed for demonstrating the idea[2]. It is a small component that generates a linear motion. It will be used to press in anything, such as lights, TV remote, and coffee machine.

2.1.3 Programming Languages & Frameworks

- **Python:** raspberry pi can be controlled by either c++ or python. Python was chosen because a REST API can be made using it fast.
  - **GPIO:** a library for controlling any hardware component connected to the GPIO pins[12].
  - **Flask:** a lightweight framework to build web applications.
- **Java:** mobile application are made in a native way with either swift or java.
  - **Android:** a framework for making android apps.
  - **Retrofit:** type-safe HTTP client for Android and Java[13]. It will be used to send and receive commands and status from the web server.
- **PostgreSQL:** an open-source RDBMS[14]. It will be installed on the server.

2.1.4 SDLC Model

Incremental model will be used in this project. This model is a process of software development where requirements are broken down into multiple standalone modules of software development cycle. Incremental development is done in steps from analysis design, implementation, testing / verification, maintenance[15]. The reason this model was chosen is the pieces will be installed, tested and connected to the system gradually. First a LED, then a linear solenoid and so on.

2.2 Related Work

2.2.1 Insteon - Insteon Hub

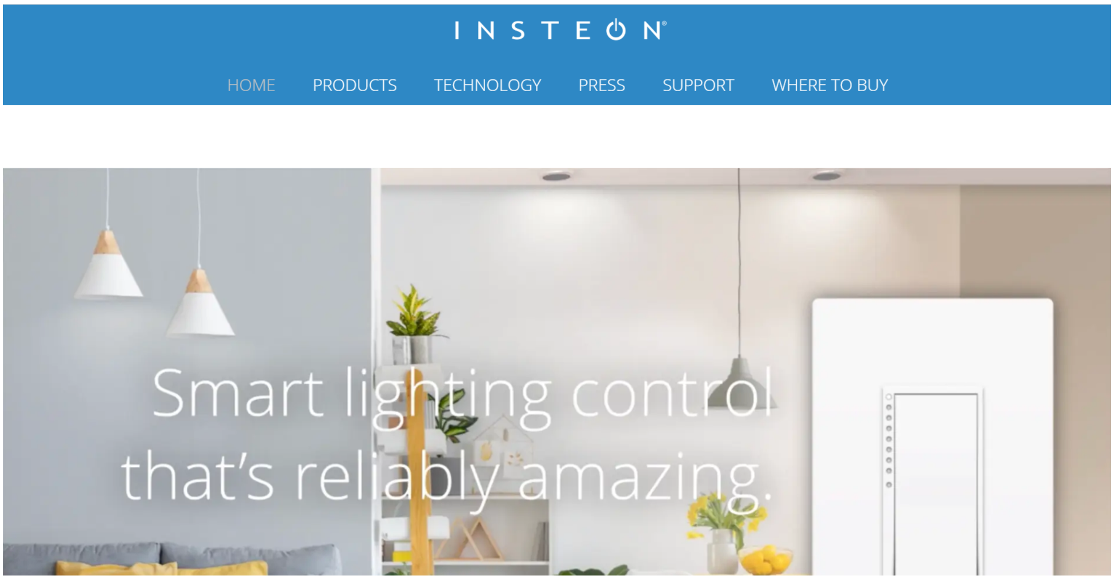


Figure 2.1: Related Work: Insteon

Insteon Hub is a simple and straightforward system that connects you to your home from any smartphone or tablet, anywhere in the world. Control Insteon light bulbs, wall switches, outlets, and thermostats at home or remotely and receive instant email or push notification alerts from motion, door and window, water leak, and smoke sensors while youre away[16]. *Figure 2.1 shows the home page for Insteon.*

- **Advantage:**

  1. Control Multiple Devices Simultaneously with a Basic Scene.
  2. Create Schedules to Turn Your Lights On and Off at Specific Times.
  3. Automatically Turn Lights On and Off with Sensors.
  4. Monitor Your Home with Email or Push Notification Alerts.

- **Disadvantage:**

1. Hub setup takes a couple of minutes and a few moments per light switch, sensor.
2. Its need to connect it to power and your home's internet router so if the internet die all devices need to start over again.
3. fixed the hub take more cost than its original price.
4. There is no database save/restore. You have to recreate all the devices, scenes, schedules if its replaced.

2.2.2 Wink - Wink Hub 2

Figure 2.2: Related Work: Wink

Wink Hub 2 is the worlds first smart home hub created for the mainstream consumer. With industry-leading smart home protocol support, enhanced connectivity features, and a sleek design, Wink Hub 2 brings hundreds of products from best-in-class brands together for a simple, intuitive experience[17]. *Figure 2.2 shows the home page for Wink.*

- **Advantage:**

  1. Support Different platforms such as iOS or Android.
  2. Once you've created an account, Wink has the ability to recognize the products within Wink Bright, guide you through a few simple steps, and then you're ready to go.
  3. Wink works with Cortana Microsofts voice assistant and Amazon Alexa.
  4. One Important features in wink, its can see what youre spending even before the bill arrives.

- **Disadvantage:**

1. One major problem with the Wink 2 hub is that the device sometimes loses connectivity and must be reset in order for it to connect again.
2. Wink app doesn't always let you access other devices' full features.
3. High price.
4. Takes 14 days to arrive.

2.2.3 Samsung Smart Things Hub

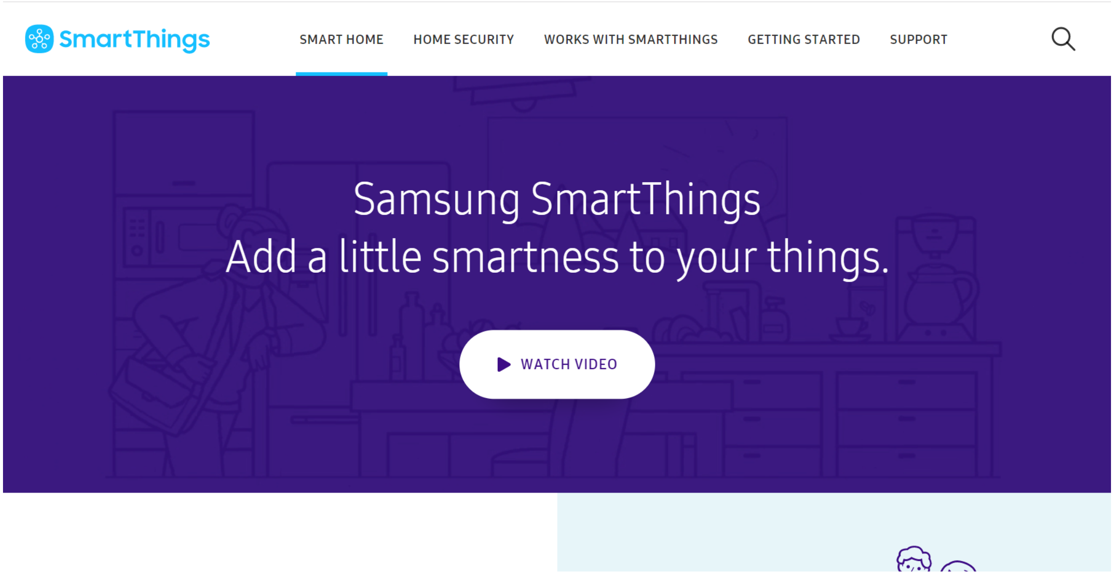


Figure 2.3: Related Work: Samsung

Smart things hub Connect wirelessly with a wide range of smart devices and make them work together[18]. *Figure 2.3 shows the home page for Samsung.*

- **Advantage:**

  1. Monitor and control connected devices in your home using a single SmartThings app for iPhone or Android.
  2. Manage connected devices in your home with SmartThings Routines for Good Morning, Goodbye, Good Night, and more.
  3. Receive alerts from connected devices when theres unexpected activity in your home.

- **Disadvantage:**

  1. Some compatible components may not work as efficiently or smoothly as you want them to, which may be inconvenient.
  2. Some users report it stops working at times.
  3. Difficult to upgrade from older hub.
  4. In US Only.

2.3 Proposed & Similar System Comparison

	Raspberry Pi	**Insteon**	**Wink hub 2**	**Samsung (smart things)**
design				
Works With Wi-Fi	yes	yes	yes	yes
Price	25$, parts are very cheap	80$, expensive parts	99$, very expensive parts	70$, expensive parts
Installation & Configuration Difficulty	hard to install but doesn't takes time to reinstall and configure	easy to install and hardly takes any time setting up even if you change your home	easy to install and hardly takes any time setting up even if you change your home	easy to install and hardly takes any time setting up even if you change your home

Table 2.1: Proposed & Similar System Comparison

Better quality can be found at Appendix A

Although similar systems already exists, our system has its own special advantages. The biggest being **hardware freedom**. In other systems, there exists a main hub receiving user command from the mobile app. So far, the ideas and implementation is identical. The previous systems require the consumer to by additional parts for it to work, such as special LED lights that needs installation or a small component controlling air conditioners. Those parts are usually limited in numbers, usage and can get very expensive fast. On the other hand, our system works with any hardware component as long as connecting it to the electrical circuit is possible.

CHAPTER NO. 3
SYSTEM ANALYSIS

3 System Analysis

3.1 Requirement Specification

3.1.1 Overview

The proposed project consists of three main systems: the android application which will be the user interface, the Raspberry Pi, which is the small computer where all the hardware pieces will be connected to, and the web server which will host the REST web application and be the connection between the android application and the Raspberry Pi.

First, the user is required to install the android app on his mobile phone. When the app is opened, the user either register or login. Once logged in, a list of raspberry pis is shown with the possibility of adding more. To add a new raspberry pi, the user must scan the QR code that belongs to the raspberry pi. Any user or raspberry added is immediately uploaded to the web server.

Once a raspberry is clicked, an activity displaying the hardwares that are connected to the Raspberry Pi is shown. It gets this list of hardwares from the webserver by using the GET method. When the user clicks on any hardware that is there, a new activity opens. In it is mentioned the name of the hardware, the status (e.g. whether it is on or off), the commands and the scheduling configuration. All of this information is obtained from the server.

Next to the commands title, there will be a small button that when clicked on will open a third activity, which gives the option of adding a new command. The command can be instantaneous (for example, switching an LED light immediately) or it can be scheduled for a later date or time. For the instantaneous command, the POST method will be used, and for scheduling the commands, the user will have the option to choose the date and time he wishes the commands to be undertaken in. Whatever the outcome of this process is, a popup message will appear to the user either confirming the success of the command the user issued, or denying it while explaining the reason for that failure.

Under the commands tab, there will be a configuration section where all the scheduled commands will appear, along with their dates and times and options to edit them or delete them. The edit option will be done by the PUT method and the deleted option by the DELETE method. All the methods work on the data in the servers database i.e. they either add a new command (POST), edit a scheduled command (PUT), or delete an existing command (DELETE).

For the Raspberry Pi, the sequence it works according to is timed. Every 5 minutes, it puts the hardware status to the server so that it can show on the users application hardware list. Also, every 30 seconds, it checks the server for any new commands posted by the user from the android application. If there are any new ones that have to be, it updates its own local database (a local queue) according to the priorities and scheduled dates and times of the commands. This local database is organized according to the time the command was issued (i.e. the instantaneous commands are put at the front of this queue because of their precedence and the scheduled ones are put in the command order) and contains the command ID, which hardware this command was issued for, when this command was issued, and whether the command was successfully done or not, all gotten from the server by the GET method except the successfully done column, which the Raspberry edits according to the hardware.

Whatever the result of the command was, the Raspberry posts the response of the command to the server. The android application gets this response from the server every 5 or 10 minutes, depending on the users choice. The response is displayed as a push notification in the users mobile phone. Depending on this response, the status and configuration information in the app will be updated to reflect the success or failure of the command response. Finally, it is important to note that any new hardware or configuration added to or connected to the Raspberry will have its information posted to the server by the Raspberry computer, where the user can view it then as soon as he opens his application to the first activity. When the response is successfully done and read by the user, the webserver deletes it from the database to save space. The webserver also deletes instantaneous commands once the user is notified the execution result.

3.1.1.1 Input

The user command issued using the Android client is the main input. Each command consist of the following:

- chosen hardware.
- configuration wanted.
- optional scheduling information.

The **hardware** is the physical component connected to raspberry pi. Each hardware has a set of possible states that it can be in. Those states are called **configuration**. The **schedule** indicates the time of day and days of week the user might want the command to run at.

3.1.1.2 Output

- a response

The raspberry pi issues a **response** indicating whether the command has been successfully done with an optional message. This response is saved in the webserver, which in turn is read by the android client periodically.

3.2 Requirement Analysis

3.2.1 Software Requirements

- **Languages**
  - Java
  - Python
  - SQL
- **Frameworks & libraries**
  - Android
  - Retrofit
  - GPIO
  - Flask
  - SQLAlchemy
- **IDE**
  - Android studio
  - Pycharm
- **Databases**
  - Postgresql
  - Sqlite
- **Web Server**
  - Nginx
  - uWsgi

3.2.2 Hardware Requirements

- **Raspberry Pi**
  - Raspberry Pi 3 B+.
  - a minimum of 2 GB of RAM.
  - a minimum of 10 GB space in SD card.
  - a monitor, a keyboard and a mouse, alternatively SSH connection could be established.
  - internet connection, either via Wi-Fi or Ethernet cable.
  - breadboard, cables, and resistors for circuit.
  - RGB LED, solenoid, or any other hardware components satisfying user needs.
- **Web Server**
  - Ubuntu 16.04+ web server, we chose digital ocean's.
  - Minimum of 1GB of RAM.
  - Minimum of 10GB of available space.
- **Android mobile phone**

3.2.3 Functional Requirements

3.2.3.1 Admin's Functionalities:

3.2.3.1.1. Admin shall be able to add new hardware to the system.

3.2.3.1.2. Admin shall be able to add new configuration to the system.

3.2.3.2 Android Client's Functionalities:

3.2.3.2.1. Android client shall allow user to login or register

3.2.3.2.2. Android client shall allow user to add a new raspberry pi to the system

3.2.3.2.3. Android client shall allow user to remove a raspberry pi from the system

3.2.3.2.4. Android client should get hardware list from webserver.

3.2.3.2.5. Android client should get scheduled commands from webserver.

3.2.3.2.6. Android client shall be able to submit a new a command, might be scheduled, to webserver.

3.2.3.2.7. Android client shall be able to delete a scheduled command from webserver.

3.2.3.2.8. Android client shall be able to edit a scheduled command from webserver.

3.2.3.2.9. Android client shall get responses from webserver automatically.

3.2.3.3 Raspberry pi's Functionalities:

3.2.3.3.1. Raspberry pi should get command list each 30 seconds.

3.2.3.3.2. Raspberry pi shall be able to update local queue.

3.2.3.3.3. Raspberry pi shall execute commands saved in queue.

3.2.3.3.4. Raspberry pi shall be able to submit a command response to server.

3.2.3.4 Web application's Functionalities:

3.2.3.4.1. Web application should delete executed immediate commands.

3.2.3.4.2. Web application should delete read responses.

3.2.4 Non-Functional Requirements

Requirement	Description
Availability	The system shall not be shut down for maintenance for more than 1 minute.
Usability	The user shall be able to learn the most important 5 functions of application in 2 hours.
Verifiability	The system shall check the user identity.
Performance	Any interface between a user and the automated system shall have a maximum response time of three seconds.
Flexibility	Any hardware component, such as linear solenoid or an infra-red controller, can be added to the system as long as it can be connected to the electric circuit.
Security	Each user can only communicate with assigned raspberry pi. In a similar manner, raspberry pi only reads commands meant for it and not for other ones.

Table 3.1: Non-functional requirements

Table 3.1 shows the non-functional requirements for the system and their description. The most important requirement is availability, because the system will fail once the user and raspberry pi were not able to communicate.

3.2.5 Structured Diagrams

3.2.5.1 Use Case Diagram

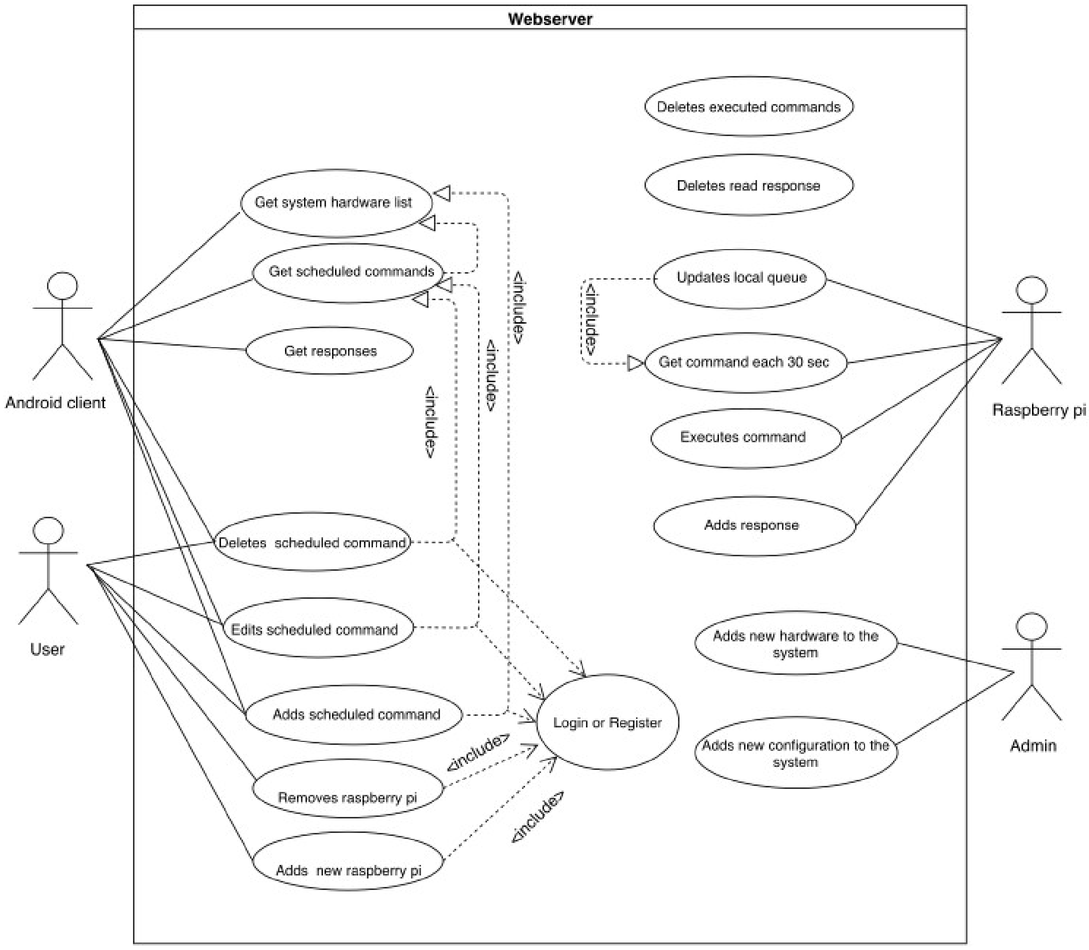


Figure 3.1: Use case diagram

3.2.5.2 Use Case Scenarios & Sequence Diagrams

Admin adds a new hardware to system

Sequence diagram: *Figure* 3.2

Goal: to add a new hardware device that the user can control to the Raspberry Pi.

Actors: Admin, Raspberry Pi, Web server.

Precondition: None.

Primary Scenario:

1. Admin physically connects the new hardware to the Raspberry Pi
2. Admin inserts new hardware attributes into hardware table in the raspberry local database.
3. Raspberry pi uploads information to the webserver using *POST* `/hardware`

Variant:

1.A. A physical defect might appear in the new hardware, the wires connecting it to the Raspberry Pi, or its ports, all resulting in the new hardware not working correctly.

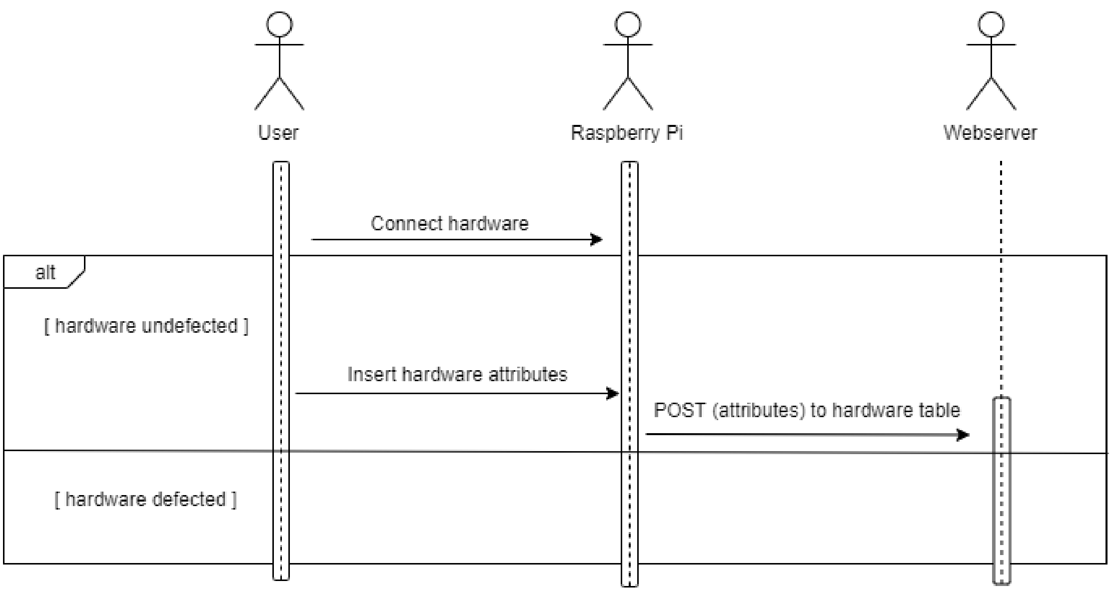


Figure 3.2: sequence diagram: Admin adds a new hardware to system

Admin adds a new configuration to system

Sequence diagram: *Figure 3.3*

Goal: to add a new configuration to a new hardware.

Actors: Admin, Raspberry Pi, Web server.

Precondition: Hardware has been added to the system successfully.

Primary Scenario:

1. Admin inserts new configuration attributes into configuration table in the raspberry local database.

2. Raspberry pi uploads information to the webserver using *POST* `/configuration`

Variant:

1.A. A physical defect might appear in the new hardware, the wires connecting it to the Raspberry Pi, or its ports, all resulting in the new hardware not working correctly.

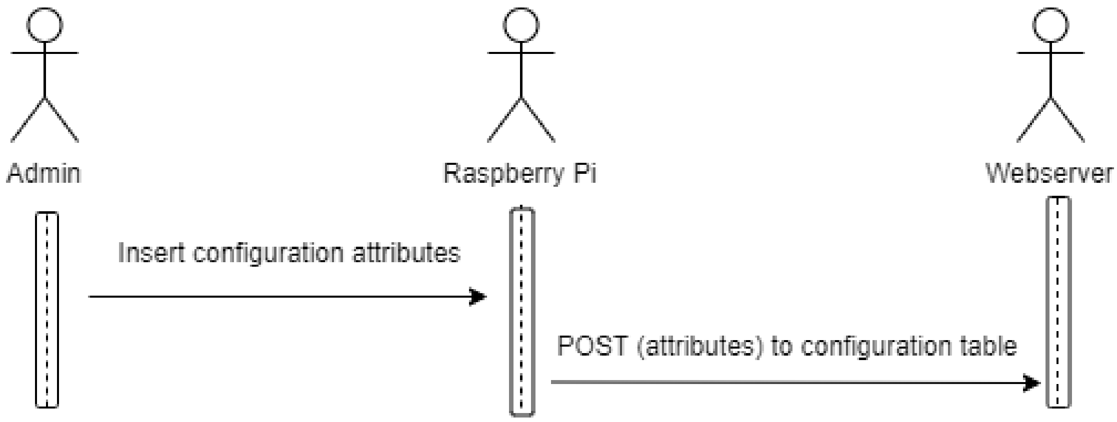


Figure 3.3: sequence diagram: Admin adds a new configuration to system

User login or register

Sequence diagram: *Figure 3.4*

Goal: to login a user if exists, or make a new user otherwise.

Actors: Android app, Web server.

Precondition: user opens android app.

Primary Scenario:

1. android app prompt user to enter credentials.
2. android POST data to webserver's endpoint /login
3. web app checks email and password. If the user already exists and credentials are correct, return token. If the the user doesn't exist create user and return token.
4. Android calls web server's endpoint /raspberry using the token in the authorization header to get the list of raspberry pi's related to the user
5. Android display this array to user in UI.

Variant:

*. user might exit android app.

2.A. android app might fail to connect to the internet

3.A. email might not match password, return error message

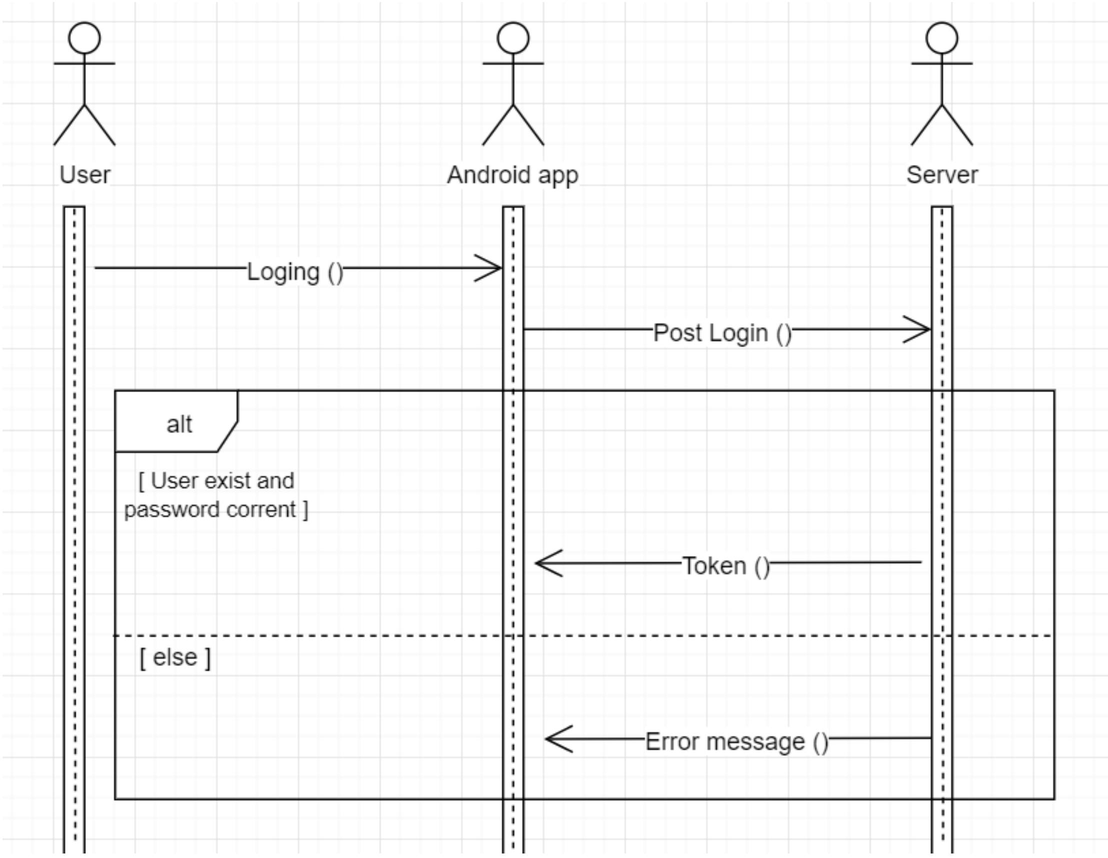


Figure 3.4: sequence diagram: User login or register

User adds new raspberry pi

Sequence diagram: *Figure 3.5*

Goal: to add a raspberry pi to the system

Actors: Android app, Web server.

Precondition: user logged in.

Primary Scenario:

1. user clicks on add new raspberry pi button.
2. android app opens camera for the user to scan the QR code on the raspberry pi.
3. once the code is scanned, the android app POST the raspberry pi id to the server's endpoint `/raspberry`.

Variant:

*. user might exit android app.

2.A. user might close camera or scan incorrect item.

3.A. android app might fail to connect to the internet

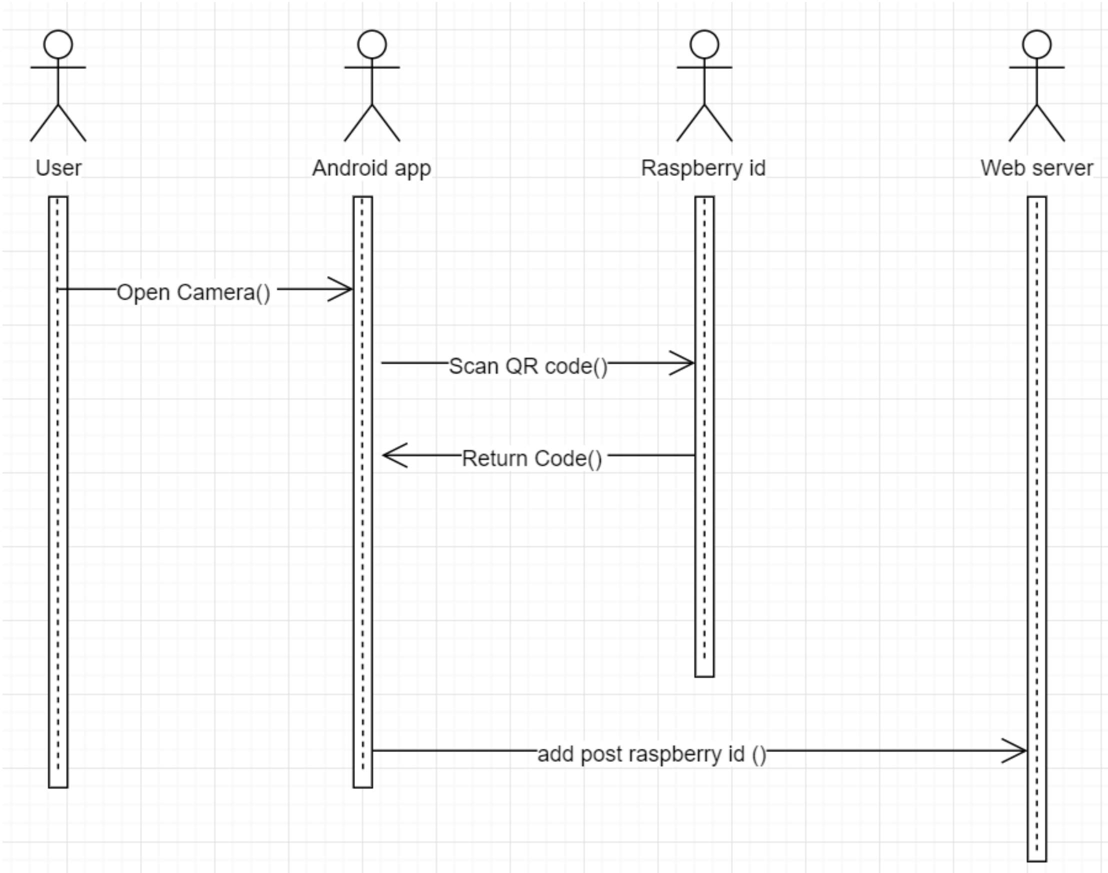


Figure 3.5: sequence diagram: User adds new raspberry pi

User removes raspberry pi

Sequence diagram: *Figure 3.6*

Goal: to remove a raspberry pi from the system.

Actors: Android app, Web server.

Precondition: user logged in.

Primary Scenario:

1. user clicks on delete raspberry pi.

2. Android client requests URL `/raspberry/{id}` with HTTP method *DELETE* where{*id*} is the raspberry pi selected id.

3. webserver receives requests and deletes the raspberry pi from user.

Variant:

*. user might exit android app.

2.A. android app might fail to connect to the internet

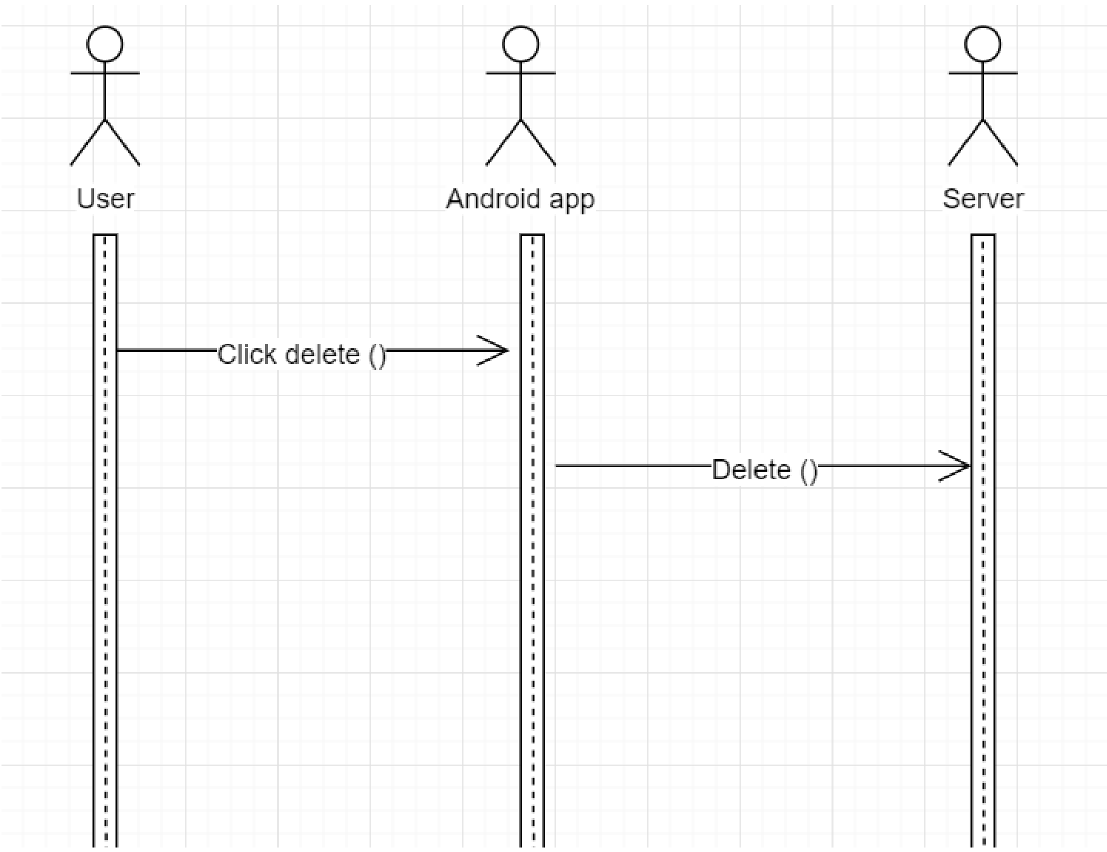


Figure 3.6: sequence diagram: User removes raspberry pi

Android app gets hardware list

Sequence diagram: *Figure 3.7*

Goal: to get all hardware in the system from webserver.

Actors: Android app, Web server.

Precondition: user logged in successfully

Primary Scenario:

1. user clicks on a raspberry pi.
2. android app calls the webserver endpoint /hardware
3. webserver fetches data from database using hardware.index()
4. webserver responses to request with an array of hardwares in the response's body
5. Android display this array to user in UI.

Variant:

*. user might exit android app.

1.A. android app might fail to connect to the internet

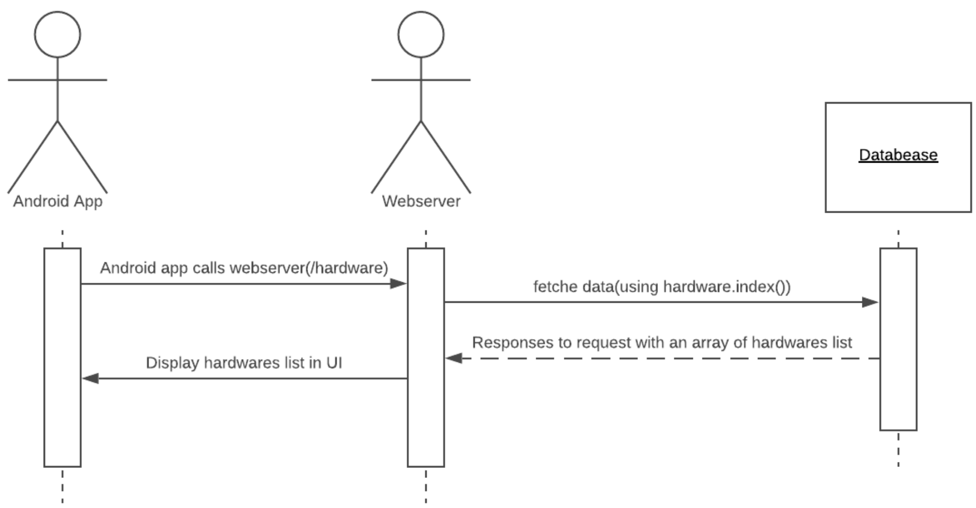


Figure 3.7: sequence diagram: Android app gets hardware list

Android app gets command list

Sequence diagram: *Figure 3.8*

Goal: to get all commands in the system related to selected hardware.

Actors: Android app, Web server.

Precondition: android app gets hardware list

Primary Scenario:

1. user selects hardware from UI.
2. android app calls the webserver endpoint `/hardware/{hardwareId}/command`, where *hardwareId* is the id of the hardware the user selected.
3. webserver fetches data from database using `command.indexByHardwareId({hardwareId})`
4. webserver responds to request with an array of commands in the response's body

Variant:

*. user might exit android app.

2.A. android app might fail to connect to the internet

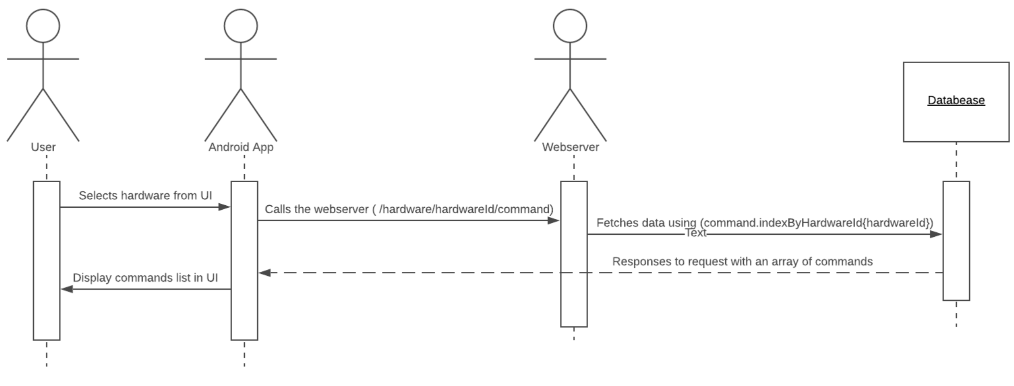


Figure 3.8: sequence diagram: Android app gets command list

Android app deletes a scheduled command

Sequence diagram: *Figure 3.9*

Goal: to delete a scheduled command from webserver related to selected hardware.

Actors: User,Android app, Web server

Precondition: android app gets all scheduled command.

Primary Scenario:

1. User clicks on delete a scheduled command.
2. Android client requests URL `/command/{id}` with HTTP method *DELETE* where{*id*} is the command selected id.
3. webserver receives requests and deletes the command.

Variant:

1.A. user might not have a schedule command.

2.A. android app might fail to connect to the internet

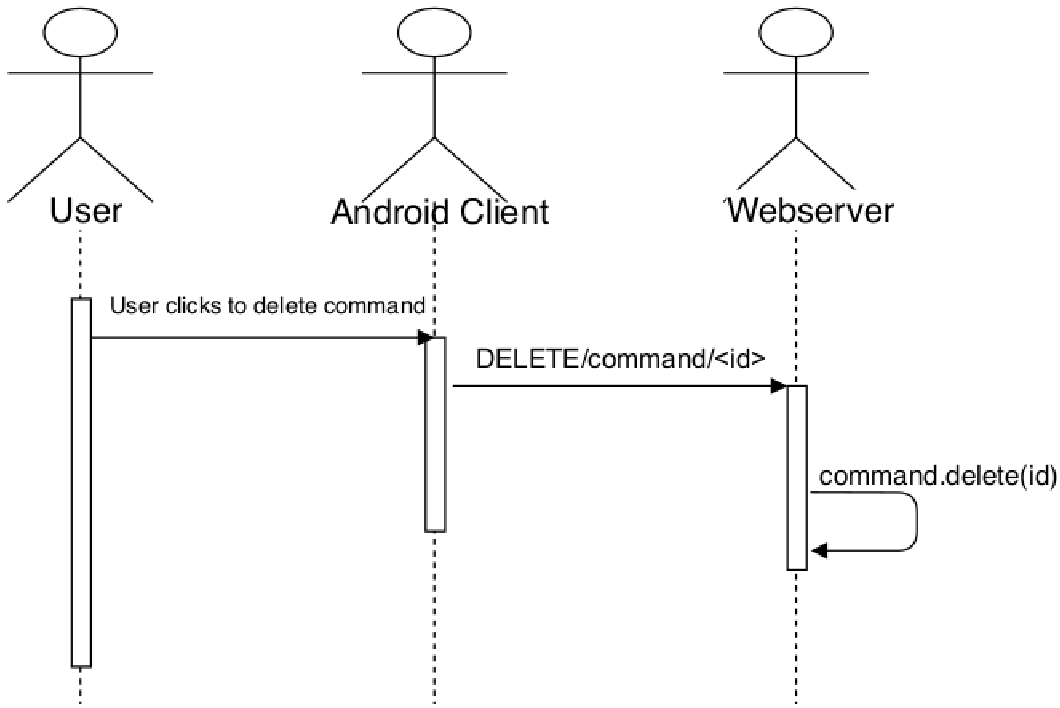


Figure 3.9: sequence diagram: Android app deletes a scheduled command

Android app edits a scheduled command

Sequence diagram: *Figure 3.10*

Goal: to edit a scheduled command from webserver related to selected hardware.

Actors: User,Android app, Web server

Precondition: android app gets all scheduled command.

Primary Scenario:

1. User clicks on editing a scheduled command.
2. Android client requests URL /command/{id} with HTTP method *PUT* where{*id*} is the command selected id.
3. webserver receives requests and edits the command.

Variant:

1.A. user might not have a schedule command.

2.A. android app might fail to connect to the internet

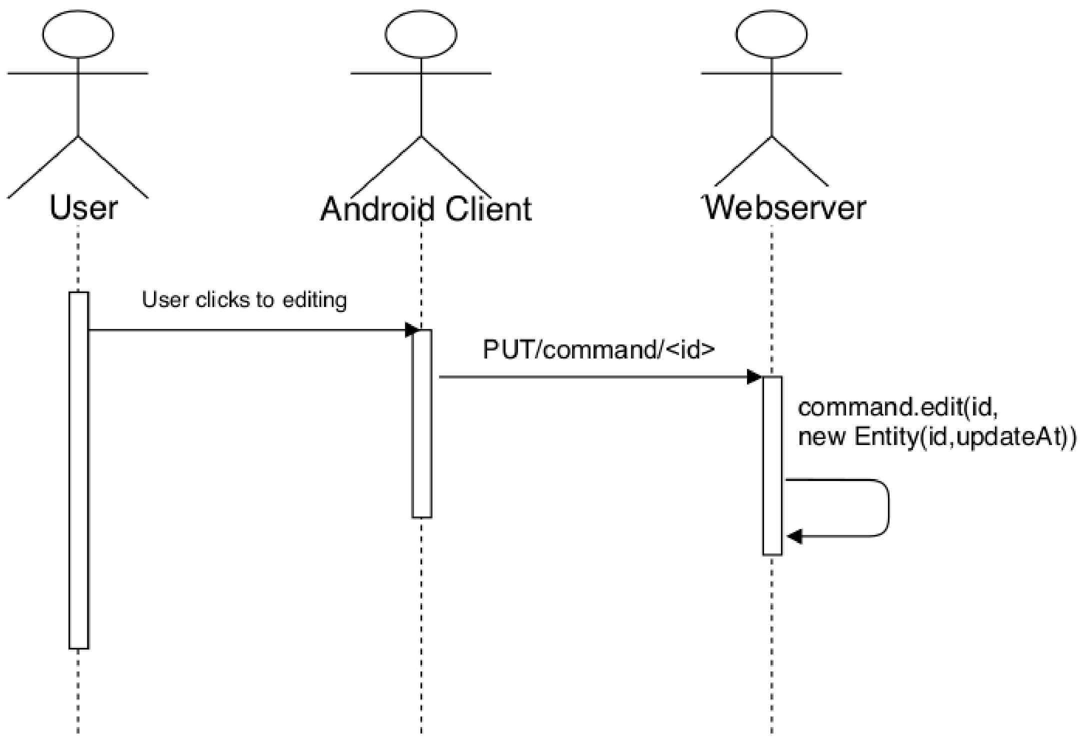


Figure 3.10: sequence diagram: Android app edits a scheduled command

Android app adds a new command

Sequence diagram: *Figure 3.11*

Goal: to add a new command to webserver.

Actors: User,Android app, Web server

Precondition: android app gets all hardwares and user chooses the hardware to add command to.

Primary Scenario:

1. User clicks on adding a new command.
2. activity appears taking the new command attributes, such as schedule or configuration.
3. Android client requests URL `/command` with HTTP method *POST*.
4. webserver receives requests and adds the command.

Variant:

*. internet might disconnect

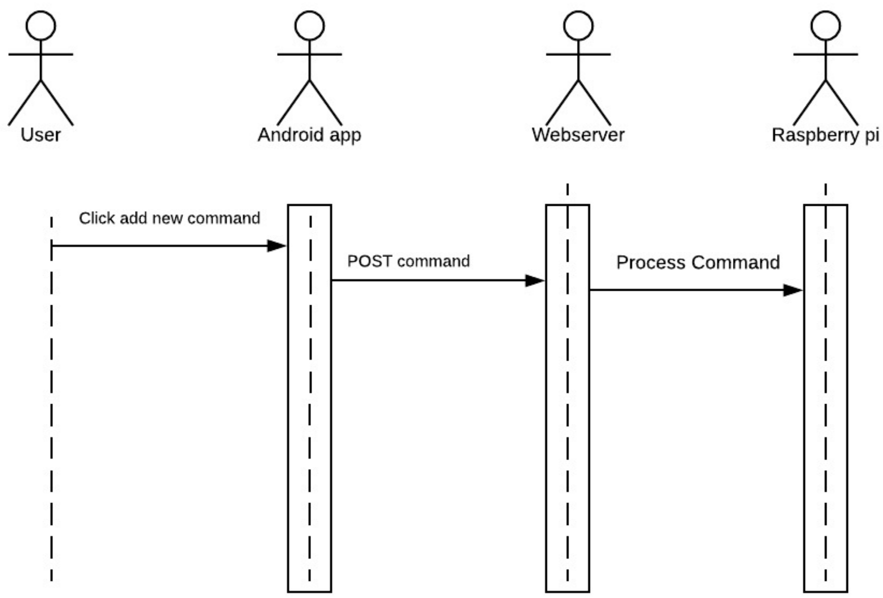


Figure 3.11: sequence diagram: Android app adds a new command

Android app gets responses

Sequence diagram: *Figure 3.12*

Goal: to get the result of the command execution to the user as a notification

Actors: Android app, Web server

Precondition: None

Primary Scenario:

1. Android client requests URL `/response` every 5 or 10 minutes.

2. if responses exist, android app shows it to user as a push notification.

Variant:

*. internet might disconnect

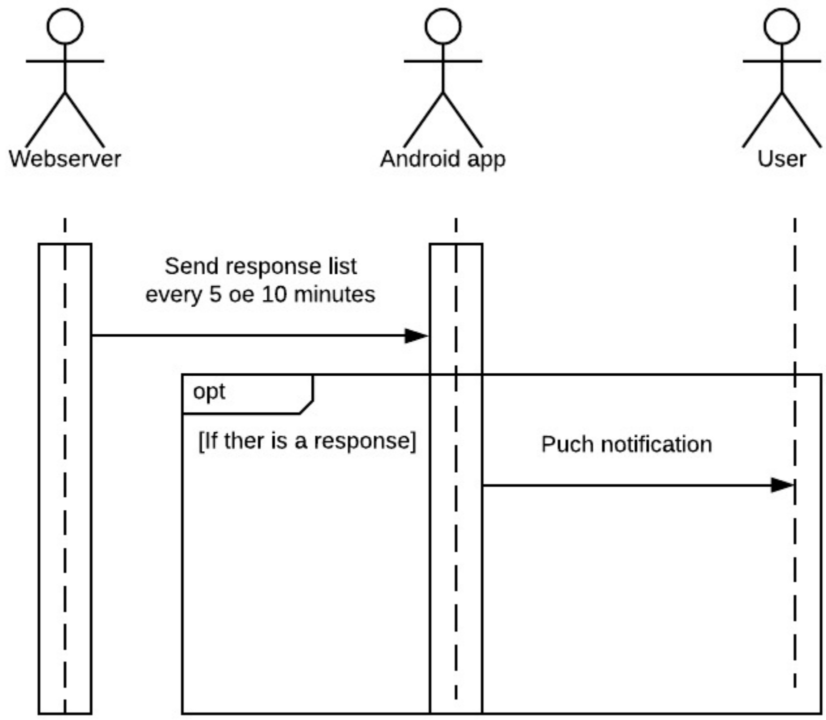


Figure 3.12: sequence diagram: Android app gets responses

Raspberry pi gets command list

Sequence diagram: *Figure 3.13*

Goal: to get all commands in the system.

Actors: Raspberry pi, Web server.

Precondition: None

Primary Scenario:

1. Raspberry pi calls the webserver endpoint `/command` to get all the commands
2. webserver fetches data from database using `command.index()`
3. webserver responds to request with an array of commands in the response's body

Variant:

*. raspberry pi might fail to connect to the internet

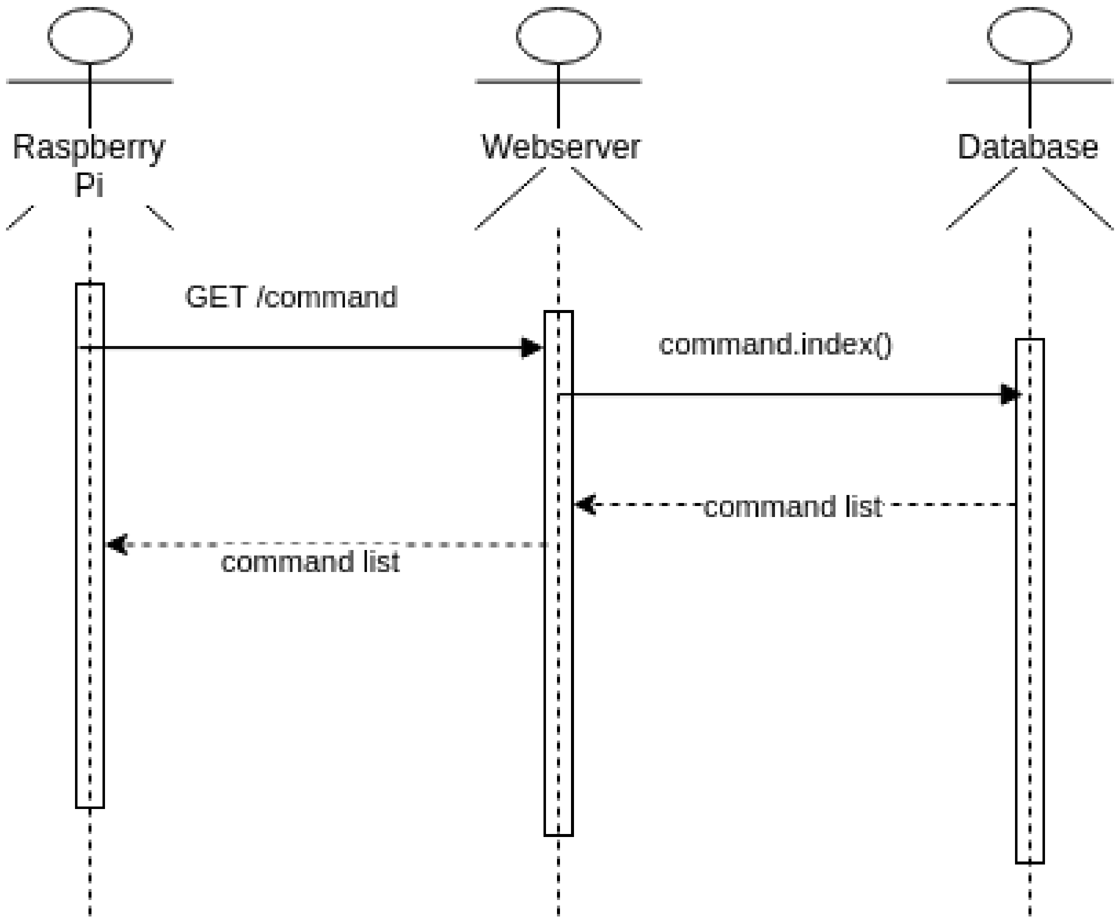


Figure 3.13: sequence diagram: Raspberry pi gets commands from server

Raspberry pi updates local queue

Sequence diagram: *Figure 3.14*

Goal: to updates local queue.

Actors: Raspberry pi.

Precondition: raspberry pi got command List from web server.

Primary Scenario:

1. Raspberry pi gets a command from list.
2. If command is immediate,i.e. has no schedule, the execution time for the processed command will be the time the command was created - accessed using command.getUpdateAt() -.
3. else -meaning command has a schedule- and today is part of the schedule days, add to queue with execution Time set to today at the schedule's time.
4. order queue by execution Time

Variant:

3.A. if today is not part of the days, discard command

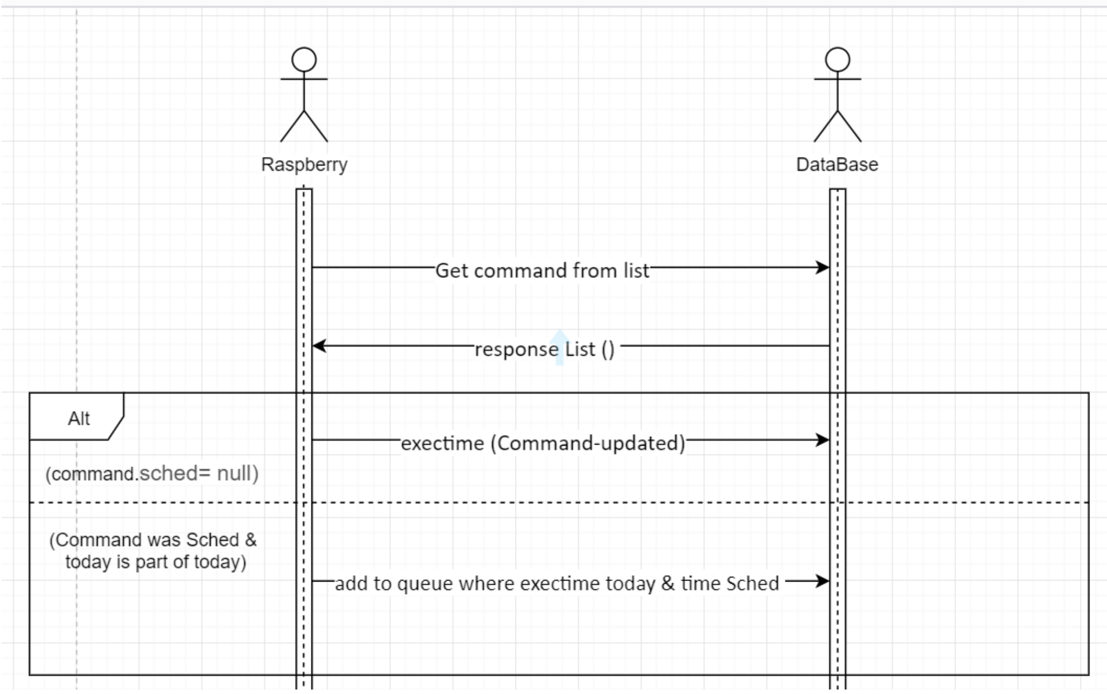


Figure 3.14: sequence diagram: Raspberry pi updates local queue

Raspberry pi executes commands

Sequence diagram: *Figure 3.15*

Goal: to execute commands saved in the local queue on a timely fashion.

Actors: Raspberry Pi.

Precondition: raspberry pi got the commands from webserver, updated local database.

Primary Scenario:

1. raspberry pi fetches first processed command saved in local queue.

2. raspberry pi calls the function `EcecuteCommand()` and pass it the command as a parameter.

3. raspberry pi look for the pin connecting the hardware and apply the desired configuration to it.

Variant:

1.A. local queue might be empty, no action is taken.

2.A. a hardware error might happen, such as disconnected cable.

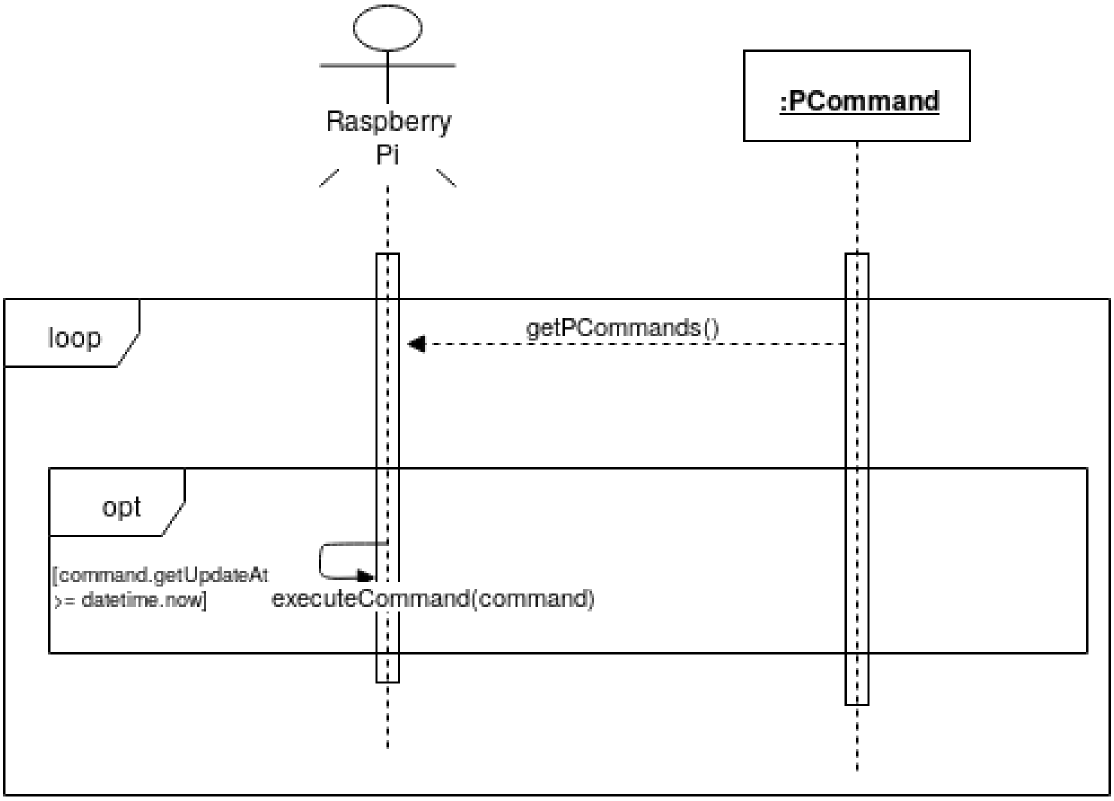


Figure 3.15: sequence diagram: Raspberry pi executes commands

Raspberry pi submits a command response to server

Sequence diagram: *Figure 3.16*

Goal: to submit the result of the command execution to the server by saving it as a response row.

Actors: Raspberry Pi, webserver.

Precondition: raspberry pi executed commands.

Primary Scenario:

1. raspberry pi determines whether the execution was successful.
2. raspberry pi includes a message if any.
3. raspberry pi requests the webserver endpoint `/response` with the method *POST*, needed parameter are added to the request body.
4. webserver receives request, a new row to the database response table is added and ready for the android client to read.

Variant:

*. internet might disconnect.

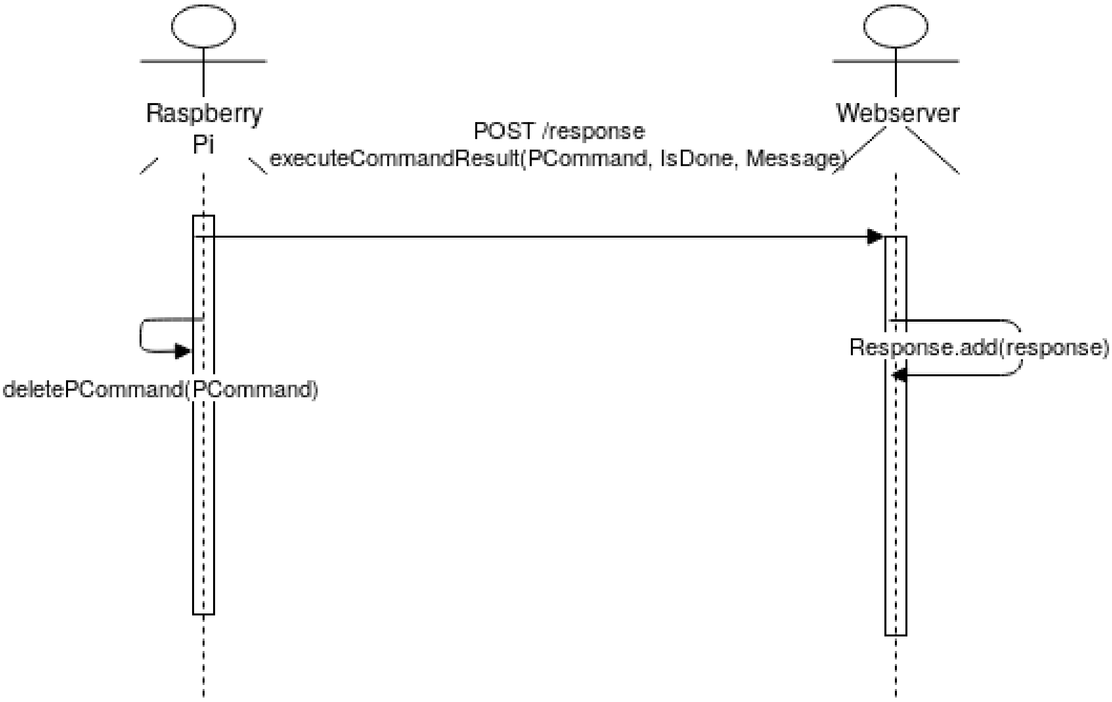


Figure 3.16: sequence diagram: Raspberry pi submits a command response to server

Web application deletes executed immediate commands

Sequence diagram: *Figure 3.17*

Goal: to delete all executed immediate commands.

Actors: Web application, database.

Precondition: None

Primary Scenario:

1. Web application join between a command table and Response table from database.
2. Web application will select the immediate command from database using `Command.scheduled=null`.
3. Web application checks if response has been read by the user. using `response.isRead()` then, deletes command.

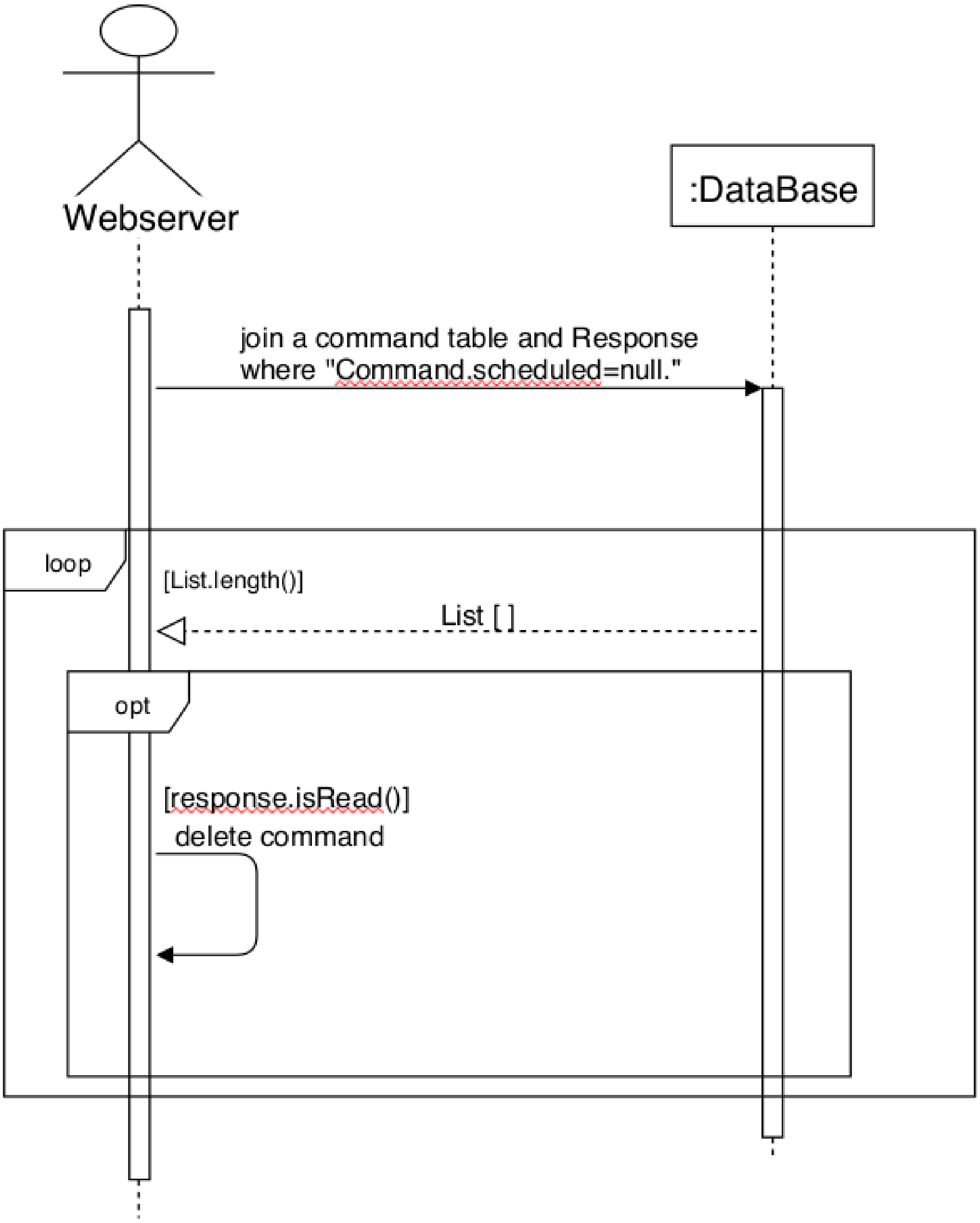


Figure 3.17: sequence diagram: Web application deletes executed immediate commands

Web application deletes executed immediate commands

Sequence diagram: *Figure 3.18*

Goal: to delete all executed immediate commands.

Actors: Web application, database.

Precondition: None

Primary Scenario:

1. Web application fetches responses from database

2. if response has been read by the user, delete it.

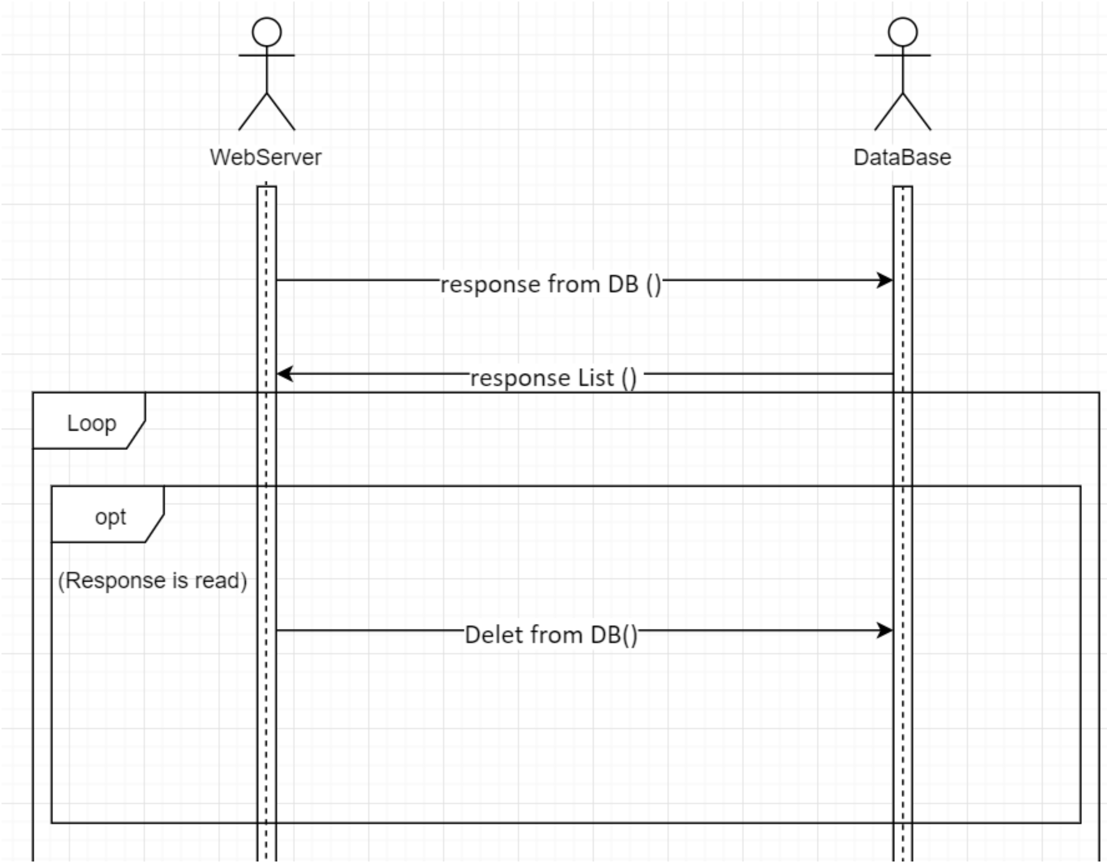


Figure 3.18: sequence diagram: Web application deletes responses

3.2.5.3 Flowchart Diagram

Figure 3.19 shows the flow chart for the android app, and figure 3.20 shows the raspberry pi's flowchart.

- **Raspberry pi**: Every 30 seconds, raspberry first requests command list from the server, the web application then fetches the list from the database and returns it. Raspberry pi updates the local queue with them. It then checks the first command in the queue, if it was due it gets executed and post the response to the server. After execution, it checks the first in queue again.

- **Android app**: when android app is first opened, it requests hardware list from the webserver. After the web app fetches the hardware list from the database, it returns it to the android app, which displays the hardwares to the user. If the user clicks in a hardware, a new activity is opened containing the hardware information and the command list for that hardware. The user then can do 3 things:

  1. Edit a certain command: android request PUT to the command chosen from the server.
  2. Delete a certain command: android request DELETE to the command chosen from the server.
  3. Add a new command: android shows a new activity prompting the user for the command information, then it requests POST command to the server.

  Regardless of the choice, the web app then process the request.

- **Android app back work**: each 5 or 10 minutes, android requests the responses from the webserver. When there is a response to be read, it shows it to the user as a push notification.

- **Web app**: when a command has no schedule and it was executed, the web app immediately deletes it from the database. Web app also deletes any read response.

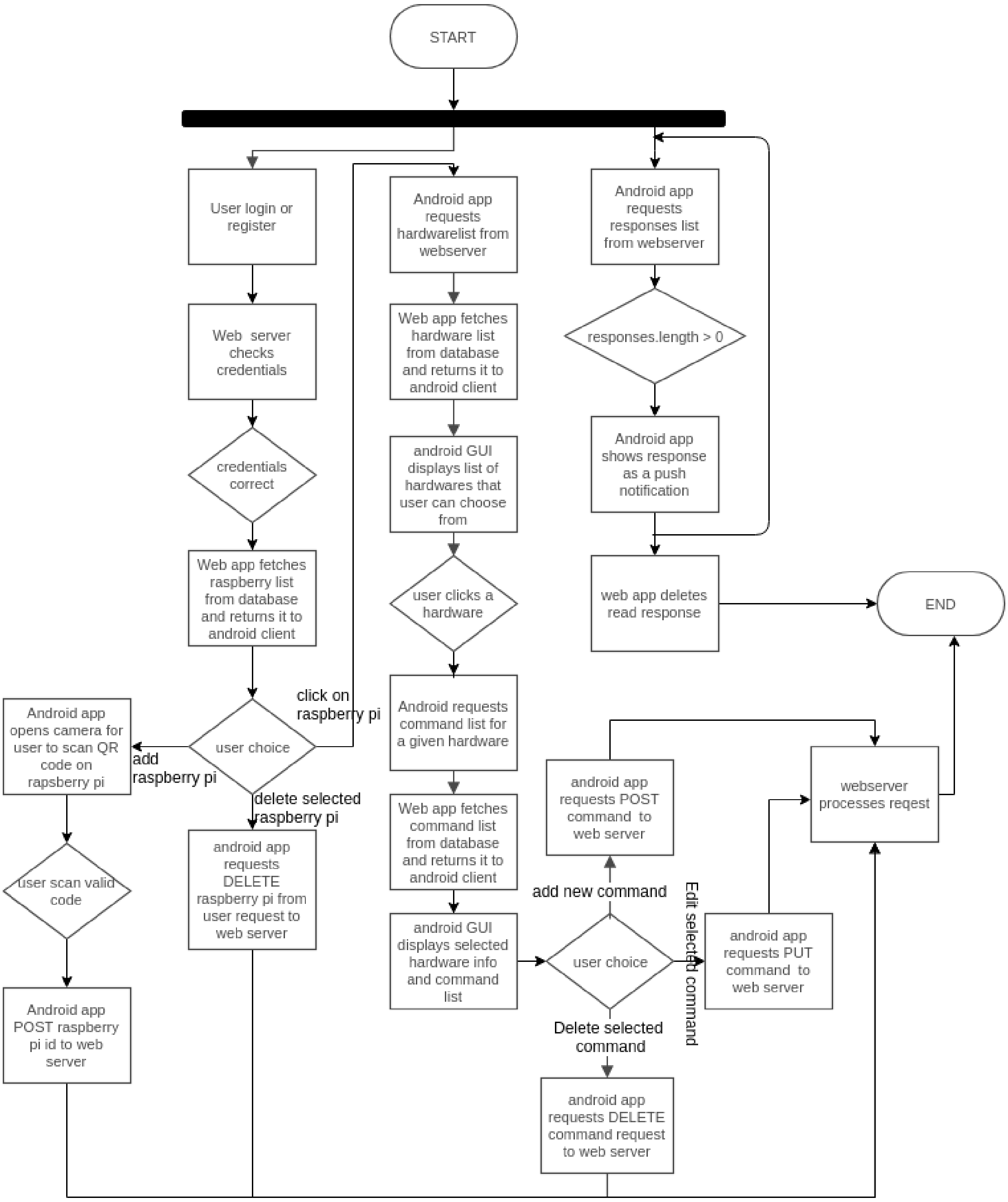


Figure 3.19: Flow chart - Android app

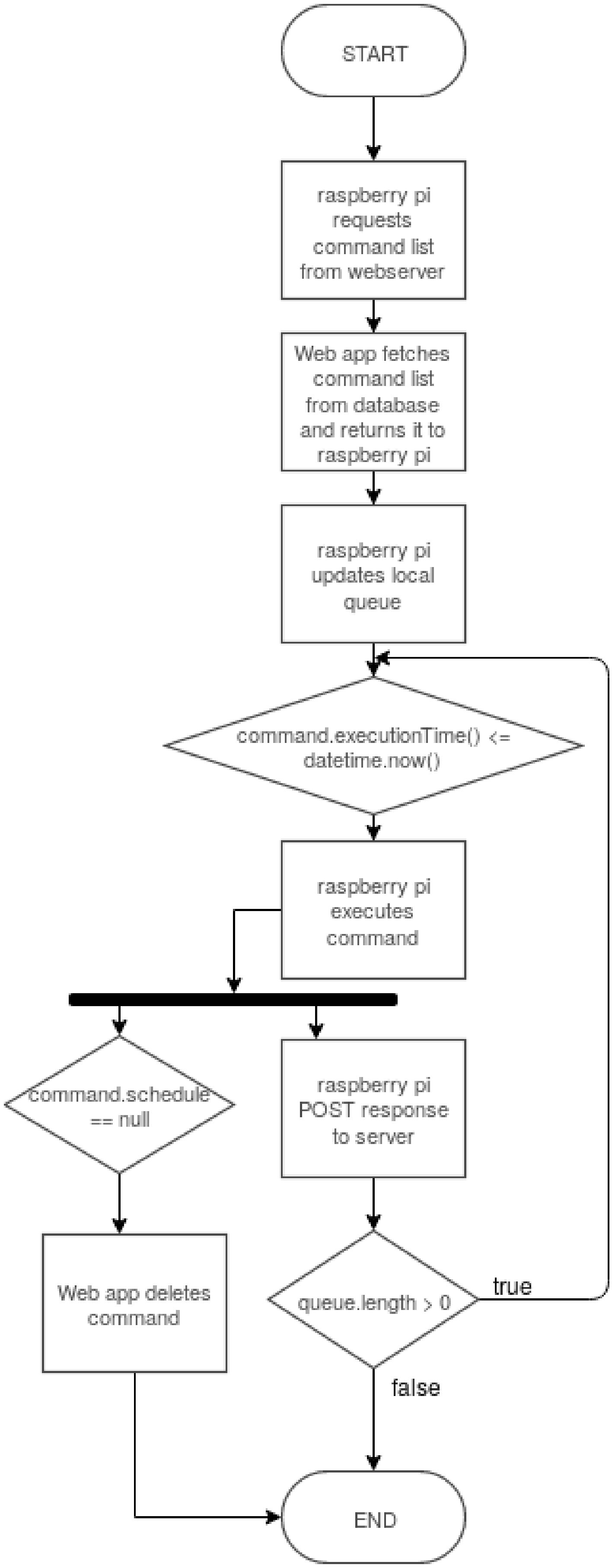


Figure 3.20: Flow chart - Raspberry pi

3.2.5.4 Entity Relationship Diagram

In our system, there are two databases: the *system database* stored in the web server, and *the queue*, a local database for the raspberry pi. To make understanding the databases' easier, we represented it using entity relationship diagram. It is a graphical representation that demonstrates the relationship between concept, people, places, objects or events inside a system. The main components are the entities, relationship and cardinality. Entities are concepts or object that need their data stored. Cardinality defines that relationship in terms of numbers[19].

- **System database**: This database is the main database. It will be stored in the web server and gets accessed by the android client and the raspberry pi. There are 7 main entities:

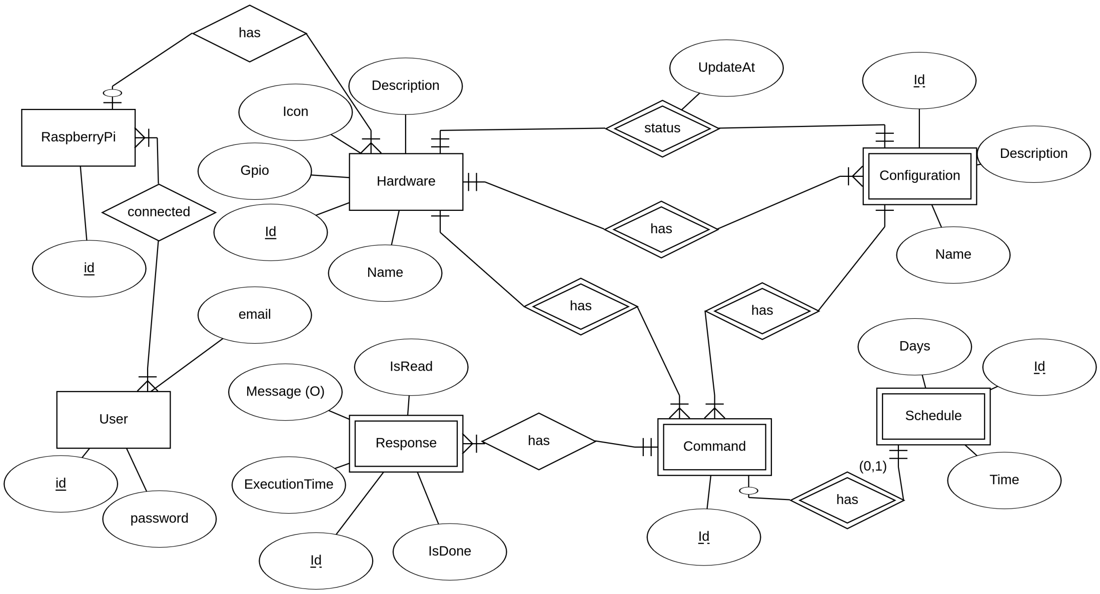
  1. **User:** to manage access, a user entity is essential. IT will store the user credentials.
  2. **RaspberryPi:** Here, the data for each physical raspberry pi is saved. Each user can have many raspberry pis and each raspberry can belong to multiple people.
  3. **Hardware:** it will store information related to the physical components connected to a raspberry pi. Such as LED lights, linear solenoid or an infra-red controller.
  4. **Configuration:** hardwares can have different states. For example, a LED light could be on or off. An RGB LED can be ON on a certain color, of off. Configurations save the possible states a hardware can be in.
  5. **Command:** users can change raspberry pi's hardwares to be in a certain configuration. These commands are stored here. The users requests could either be instant, or scheduled.
  6. **Schedule:** since commands can be scheduled, those data should be saved here. The user can choose the days and time a command fires.

7. **Response:** when raspberry pi finish executing a command, it issues a response.

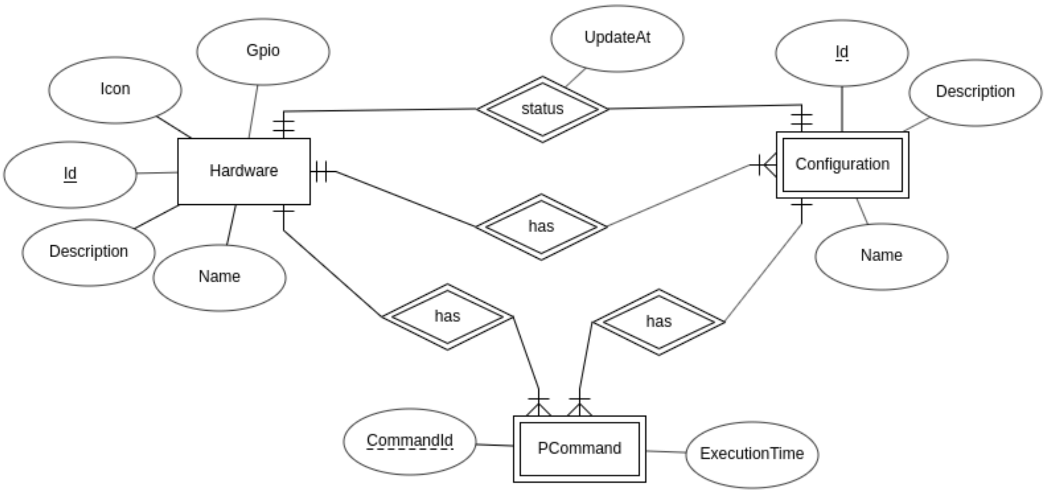
- **Local Queue**: This database is stored locally in raspberry pi. After raspberry gets user commands from server, it process them and order them based on their execution time.

1. **Hardware:** similar to the server's hardware list, this entity store the hardware connected. The system database gets data related to hardware from this entity
2. **Configuration:** this table stores the possible configuration. When the table is edited, the server database is updated.
3. **PCommand:** only the commands to be executed are stored here, once a command is executed, it gets deleted from the queue. This entity contains processed commands. i.e each queue stores the exact date and time a command will be executed instead of the schedule information.

Figure ?? shows the Entity-relationship diagram.



(a) System database



(b) Local queue

Figure 3.21: Entity-relationship diagrams

3.2.5.5 Dataflow Diagram

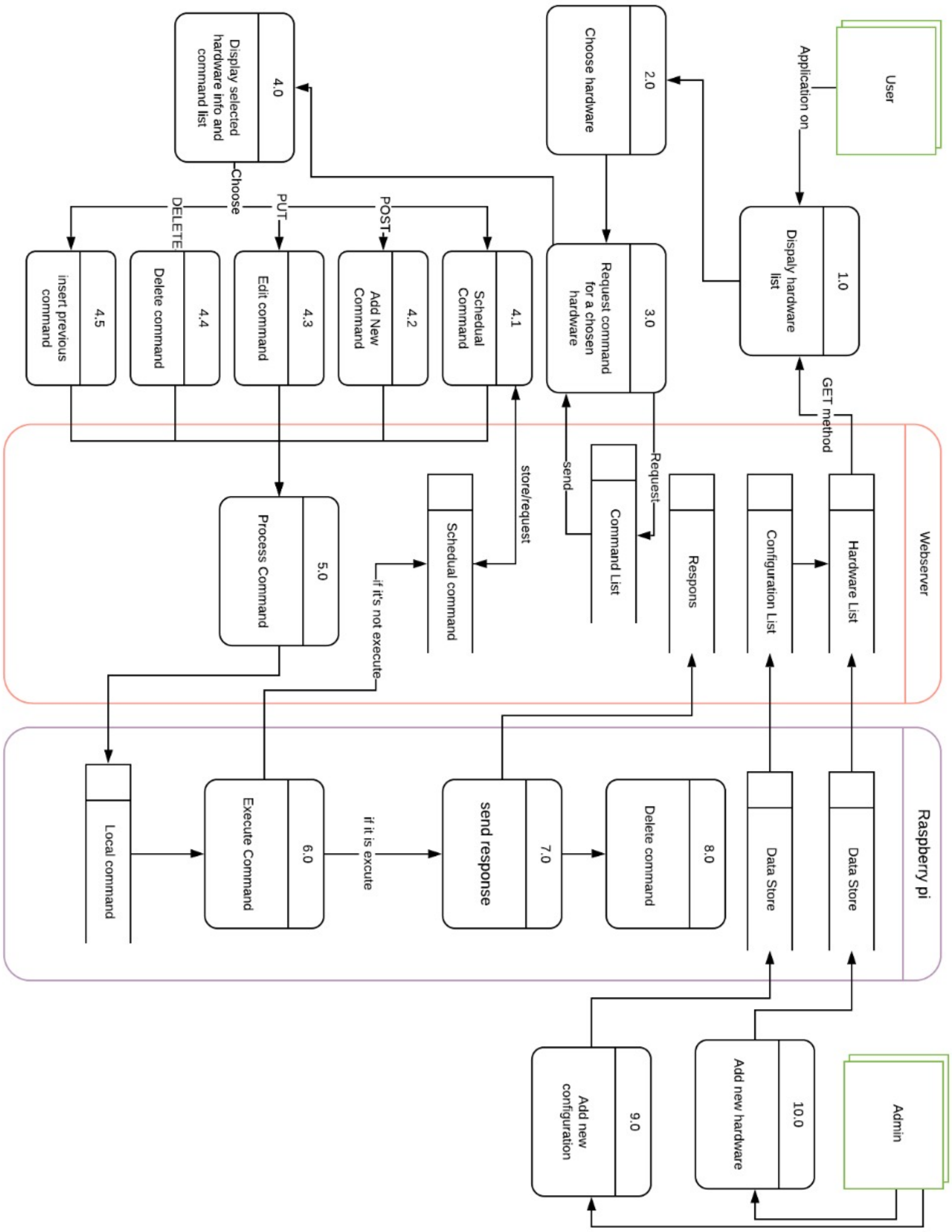


Figure 3.22: Dataflow diagram

3.2.6 Object-Oriented Diagrams

3.2.6.1 Class Diagram

Figure 3.23 shows the class diagram for the web application, and *figure 3.24* shows the class diagram for raspberry pi.The classes are essentially wrappers for the databases tables, except for the class **RaspberryPi**, where the functions for controlling the hardwares are.

The classes which represent database's tables inherit from a **BaseEntity** generic class. This generic class has the entity's id and **updateAt** property, which determines when the entity was created/last updated. This generic class is important as it has the methods needed to communicate with the web server. An example is `hardware.edit(1)`, which corresponds to the REST API **PUT** `/hardware/1`.

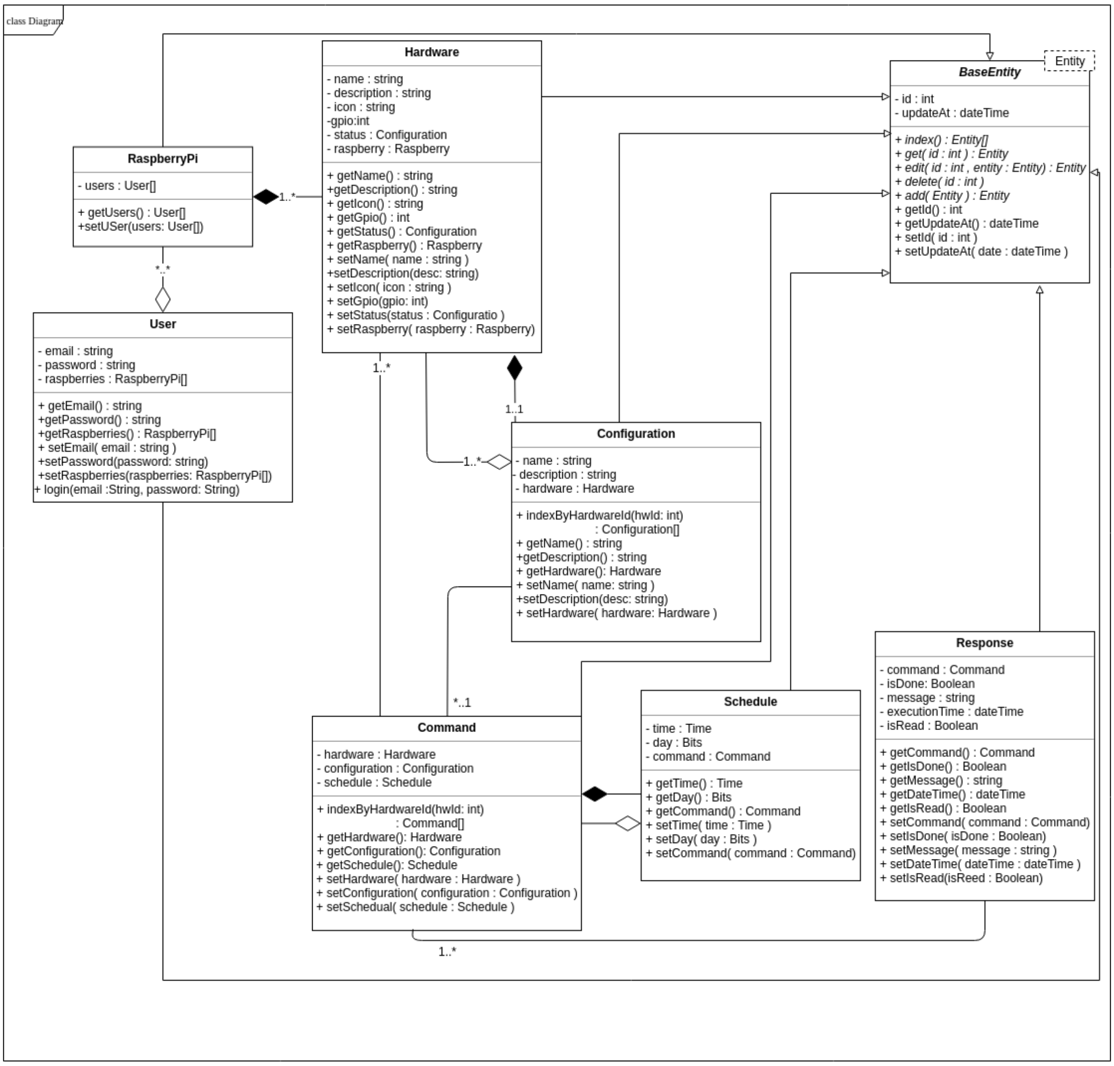


Figure 3.23: Class diagram for the web application

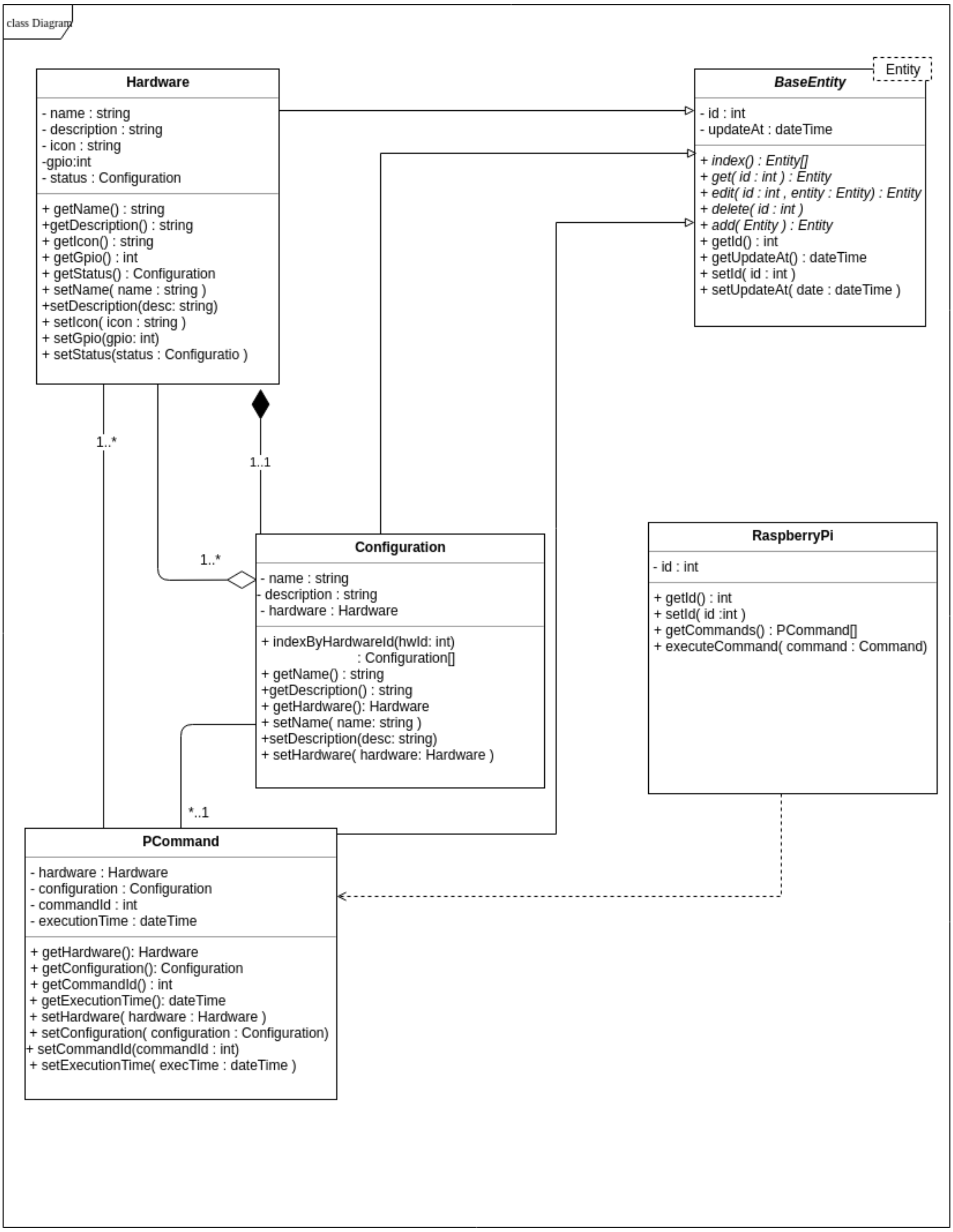


Figure 3.24: Class diagram for the raspberry pi

3.2.6.2 UML representation of the REST API Diagram

Correct calls to the web application via the REST API is the building block of our project. Thus, talking about the structure is essential. The API has 5 entities representing the database's tables. For each entity, client can use 2 methods with the entity url -e.g. `/hardware`-:

- **GET**: this method corresponds to the SQL's `SELECT * from ...;`.
- **POST**: a body of type JSON is required. It holds the object attributes. This method corresponds to the SQL's `INSERT INTO`.

Also, client can use 3 methods with the entity url + id -e.g. `/hardware/6`-:

- **GET**: this method corresponds to the SQL's `SELECT * from ... WHERE id = ...;`.
- **PUT**: a body of type JSON is required. It holds the object attributes. This method corresponds to the SQL's `UPDATE ... WHERE id = ..;`.
- **DELETE**: this method corresponds to the SQL's `DELETE FROM ... WHERE id = ..;`.

Figure 3.25 shows the API's hierarchy and figure 3.26 shows the models. For full documentation see Appendix B

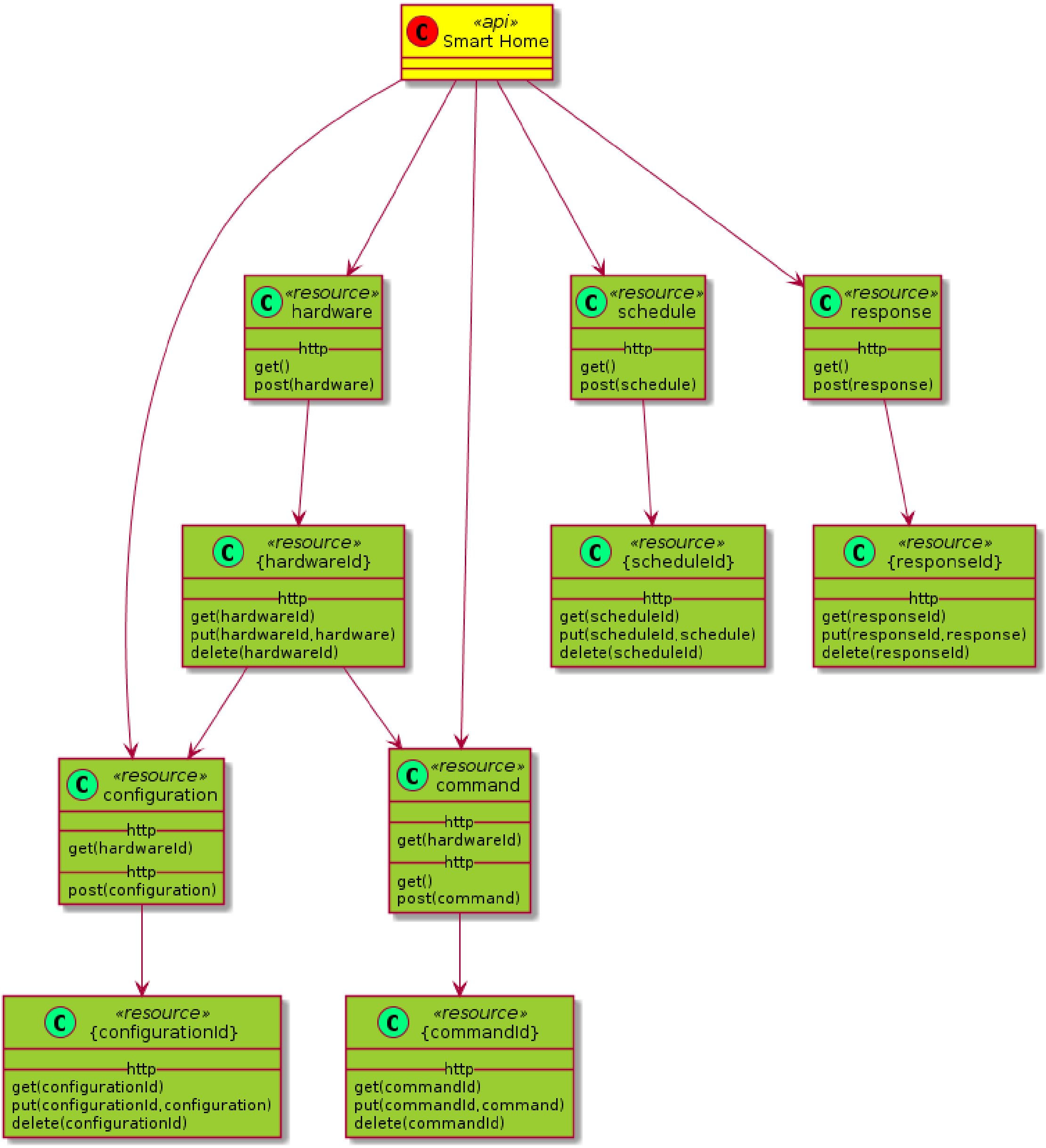


Figure 3.25: REST API

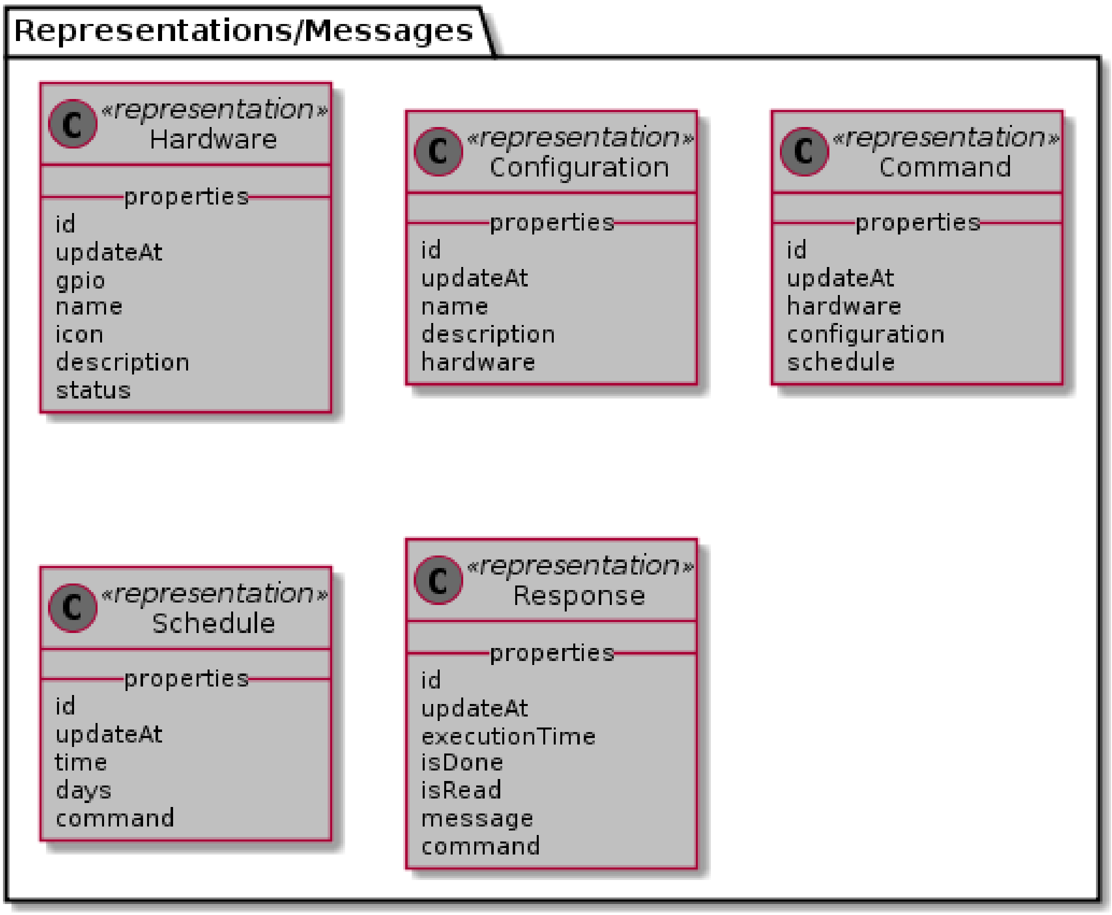


Figure 3.26: REST API models

CHAPTER NO. 4
SYSTEM DESIGN

4 System Design

4.1 System Architecture

The proposed project has 3 sub-systems. The Android App, the web app and finally the raspberry pi script. The android app works as the user interface. The raspberry pi app is the one responsible for controlling the actual hardwares. Finally, the web app works as a medium for managing user requests and raspberry pi's response. *Figure 4.1 and figure 4.2 show the architecture for the system using component and deployment diagrams.*

4.1.1 The android application

Also known as the presentation layer. It enables the user to interact with the system through a simple graphical user interface. Android application can send requests to the web server and receive responses. This app is designed using Java programming language, Android framework, Retrofit library for managing networking and finally Gson for parsing JSON responses.

4.1.2 The raspberry pi script

Also known as the business layer. It executes the commands and send responses to the web application. The script is a simple python program that controls different hardwares using GPIO library.

4.1.3 The web application

Also known as the control layer. It works as an immediate medium between the user and raspberry pi. The advantage for adding this layer is guaranteed execution regardless of user location and distance from raspberry pi. The web app will be deployed on an ubuntu server. It is made using python programming language and flask framework. Also, sqlalchemy, which is an ORM for managing database interaction, will be used to ease the development.

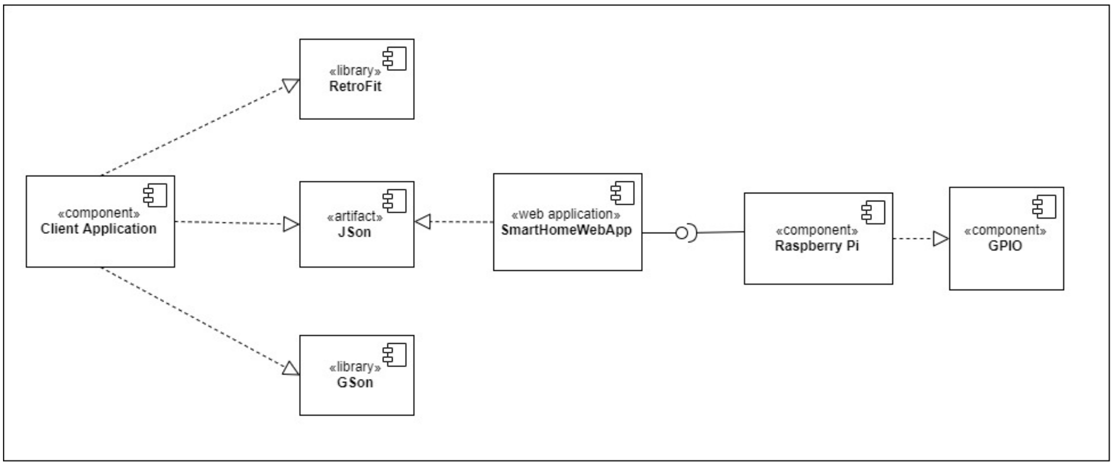


Figure 4.1: Component Diagram

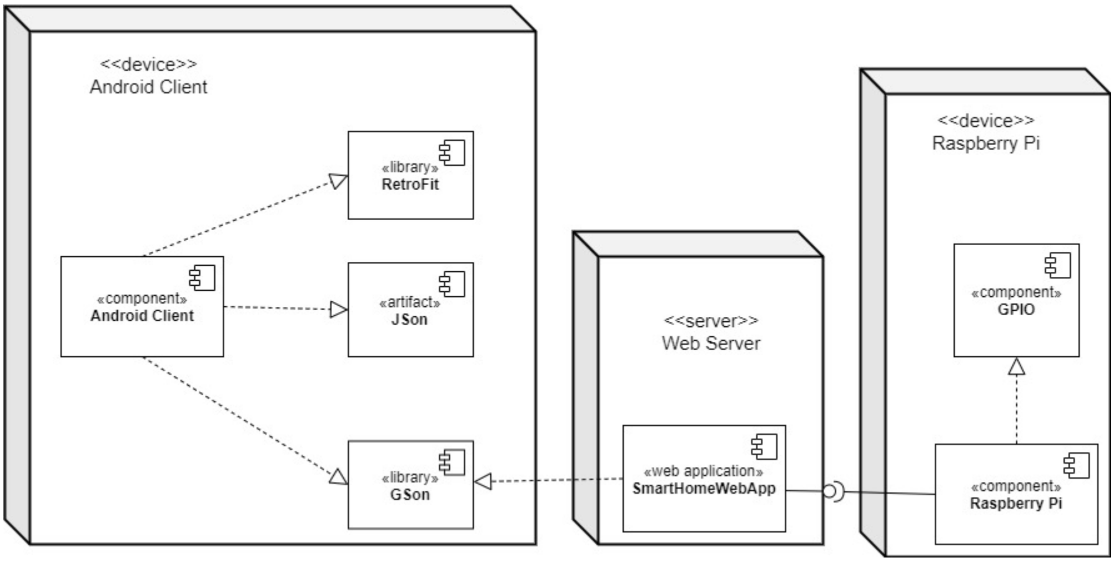


Figure 4.2: Deployment Diagram

4.2 User Interface Design

4.2.1 Home Activity

Figure 4.3 shows the home activity. It is the activity that the user will see when the app is first launched. All the hardwares in the system shall be displayed in a grid.

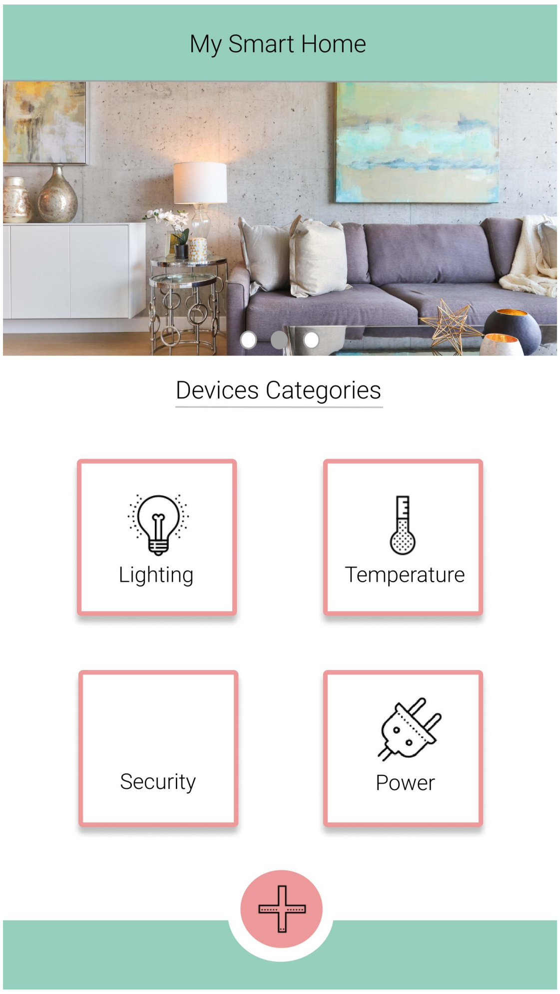


Figure 4.3: UI: Home Activity

4.2.2 Hardware Activity

Figure 4.4 shows the hardware activity. The user first is shown the current status for the hardware. Then, a list of active command is presented with the ability to edit/delete each command. Also, the user can click on the + icon to add a new command.

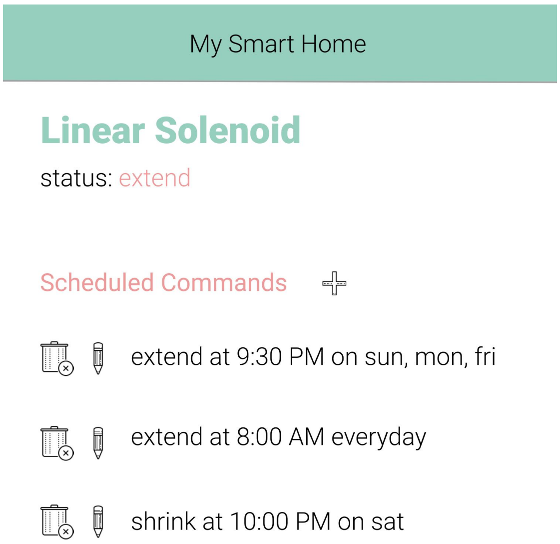


Figure 4.4: UI: Hardware Activity

4.2.3 New Command Activity

Figure 4.5 shows the new command activity, where the user can add a new command to be executed. First, the user choose the configuration wanted. Then either execute the command now, or click on the schedule, which will open a new activity to choose the scheduling information desired.

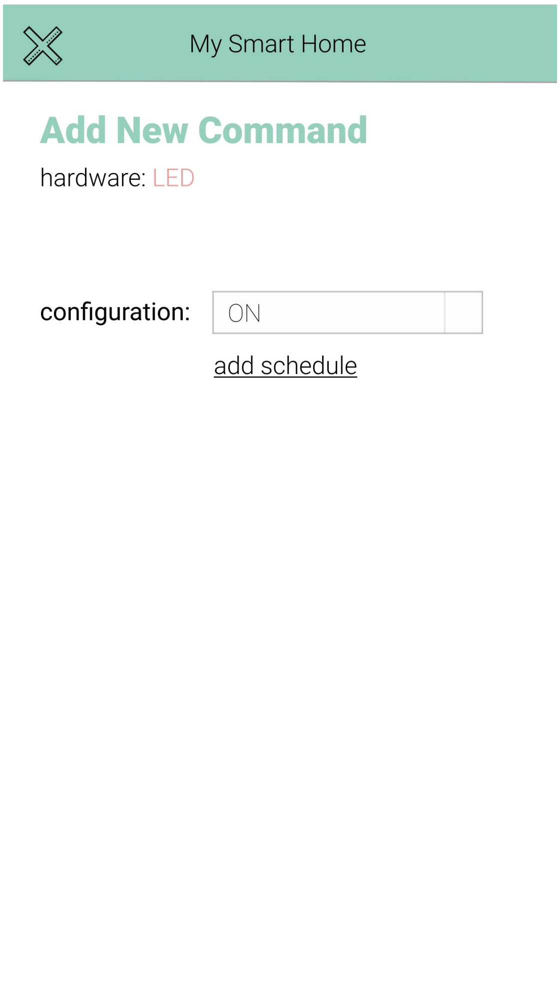


Figure 4.5: UI: New Command Activity

4.2.4 Schedule Activity

Figure 4.6 shows the schedule activity. This activity appears once the user clicks on add schedule in the previous activity. The user can choose the time of execution and the days to execute the action here.

Figure 4.6: UI: Schedule Activity

4.2.5 Progress Fragment

Figure 4.7 shows the waiting progress fragment that will appear until the web server respond with the code 201, which denotes a successful creation of a command.

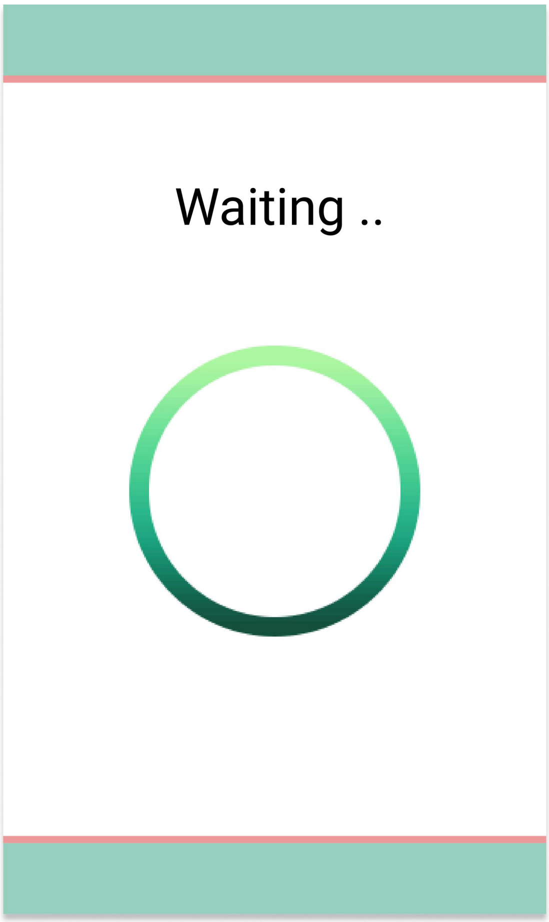


Figure 4.7: UI: Progress Fragment

4.2.6 Message Fragment

Once the server respond, the message embedded with the response shall be displayed to the user. Either **Done!** with 201 or the error message when the code is different. *Figure 4.8* shows the message when the command is successfully created.

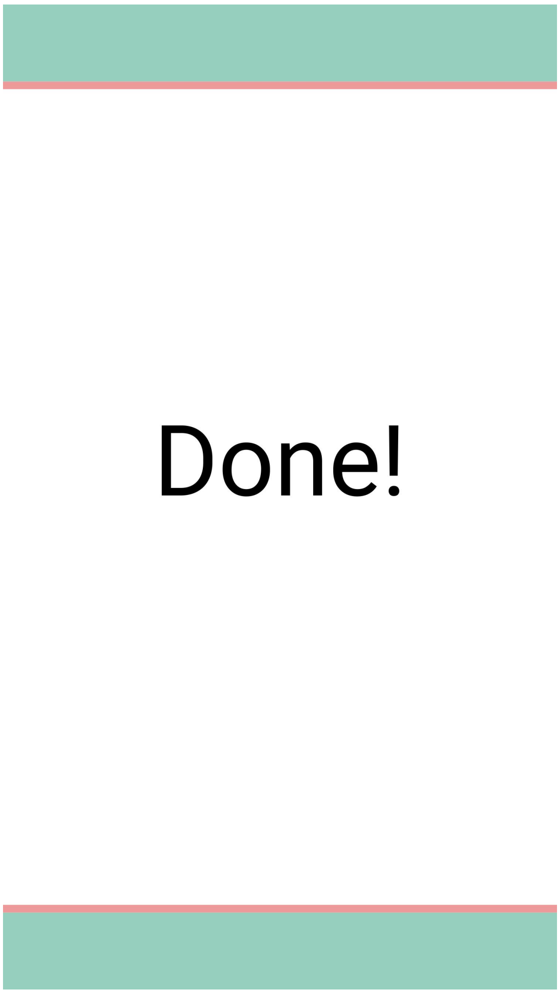


Figure 4.8: UI: Message Fragment

References

[1] "What's raspberry pi?." https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/.

[2] "Linear solenoid actuator." https://www.electronics-tutorials.ws/io/io_6.html.

[3] M. G. Samaila, M. Neto, D. A. Fernandes, M. M. Freire, and P. R. Inácio, "Challenges of securing internet of things devices: A survey," *Security and Privacy*, vol. 1, no. 2, p. e20, 2018.

[4] F. Samie, L. Bauer, and J. Henkel, "Iot technologies for embedded computing: A survey," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, p. 8, ACM, 2016.

[5] M. R. Abdmeziem, D. Tandjaoui, and I. Romdhani, "Architecting the internet of things: state of the art," in *Robots and Sensor Clouds*, pp. 55–75, Springer, 2016.

[6] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: the internet of things architecture, possible applications and key challenges," in *2012 10th international conference on frontiers of information technology*, pp. 257–260, IEEE, 2012.

[7] A. Khalid, "Smart applications for smart live," *International Journal of Computer Science and Mobile Computing*, vol. 5, pp. 97–103, 2016.

[8] I.-I. Pătru, M. Carabaş, M. Bărbulescu, and L. Gheorghe, "Smart home iot system," in *2016 15th RoEduNet Conference: Networking in Education and Research*, pp. 1–6, IEEE, 2016.

[9] M. S. Hossain and G. Muhammad, "Cloud-assisted industrial internet of things (iiot)–enabled framework for health monitoring," *Computer Networks*, vol. 101, pp. 192–202, 2016.

[10] "Droplets on digitalocean - more than just virtual machines." https://www.digitalocean.com/products/droplets/.

[11] "How light emitting diodes work — howstuffworks." https://electronics.howstuffworks.com/led.htm.

[12] "Gpio - raspberry pi documentation." https://www.raspberrypi.org/documentation/usage/gpio/.

[13] "Retrofit." https://square.github.io/retrofit/.

[14] "Postgresql: The world's most advanced open source database." https://www.postgresql.org/.

[15] "Incremental model in sdlc: Use, advantage & disadvantage." https://www.guru99.com/what-is-incremental-model-in-sdlc-advantages-disadvantages.html.

[16] "Insteon." https://www.insteon.com/.

[17] "Wink — about us." https://www.wink.com/about/.

[18] "Smartthings. add a little smartness to your things.." https://www.smartthings.com/.

[19] T. M. Connolly, I. Jacobson, and C. E. Beg, *Database Systems: A Practical Approach to Design, Implementation, and Management*. Pearson, 2005.

CHAPTER NO. 5
APPENDICES

5 Appendices

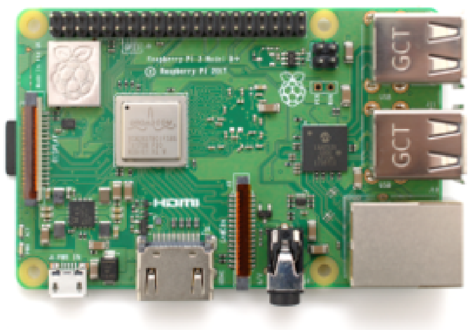
A Figures

A.1 Similar systems hub design

(a) Insteon

(b) Wink

(c) Samsung



(d) Raspberry Pi

Figure A.1: Similar systems: hub design

B REST API Documentation

Smart Home

Overview

This is the documentation for smart home web rest api by reem alghamdi

Version information

Version : 1.0

Contact information

Contact Email : reem.brain@gmail.com

License information

License : MIT
License URL : https://opensource.org/licenses/MIT
Terms of service : null

URI scheme

Host : gp.reem-codes.com
BasePath : /api
Schemes : HTTPS, HTTP

Tags

- Command : What user wants rapberry to do. It is a mapping between the time, the hardware and the configuration
- Configuration : All the possible configuration and status for a given hardware. eg: on, off, red, extended ...
- Hardware : Everything about the sensors and actuators connected to raspberry pi
- Response : What the raspberry wants the user to know. It is a mapping between the time of execution, the command triggered and whether the action was done or not
- Schedule : scheduling information for command if any. Specifys the days and time of day the user would like a command to be triggered

Paths

Add a new command

```
POST /command
```

Parameters

Type	Name	Description	Schema
Body	**command** *optional*	The command to create.	command

command

Name	Description	Schema
configurationId *required*	the id of the configuration to apply to the hardware **Example** : `3`	integer
hardwareId *required*	the id of the hardware this command targets **Example** : `1`	integer
scheduleId *optional*	the id of the schedule if any **Example** : `453`	integer

Responses

HTTP Code	Description	Schema
201	Created	Command
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Command

2

get all commands

```
GET /command
```

Responses

HTTP Code	Description	Schema
200	return an array of command objects	< Command > array

Produces

- `application/json`

Tags

- Command

get a command by id

```
GET /command/{commandId}
```

Parameters

Type	Name	Schema
Path	**commandId** *required*	integer

Responses

HTTP Code	Description	Schema
200	get the command	Command
404	Not found	No Content

Produces

- `application/json`

Tags

- Command

edit an existing command given its id

```
PUT /command/{commandId}
```

Parameters

Type	Name	Description	Schema
Path	**commandId** *required*		integer
Body	**command** *optional*	The command to edit.	command

command

Name	Description	Schema
configurationId *required*	the id of the configuration to apply to the hardware **Example** : 3	integer
hardwareId *required*	the id of the hardware this command targets **Example** : 1	integer
scheduleId *optional*	the id of the schedule if any **Example** : 453	integer

Responses

HTTP Code	Description	Schema
200	Edited	Command
404	Not found	No Content
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Command

delete a command by id

```
DELETE /command/{commandId}
```

Parameters

Type	Name	Schema
Path	**commandId** *required*	integer

Responses

HTTP Code	Description	Schema
204	deleted	No Content
404	Not found	No Content

Tags

- Command

Add a new configuration to a hardware

```
POST /configuration
```

Parameters

Type	Name	Description	Schema
Body	**configuration** *optional*	The configuration to create.	configuration

configuration

Name	Description	Schema
description *optional*	information about the configuration **Example** : `"EXTEND means that it will become longer by 3cm"`	string
hardwareId *required*	the hardware this configuration belongs to **Example** : `"ON belongs to the hardwareID 1, which is RED LED"`	integer
name *required*	the configuration name **Example** : `"ON"`	string

Responses

HTTP Code	Description	Schema
201	Created	Configuration
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Configuration

get a configuration by id

```
GET /configuration/{configurationId}
```

Parameters

Type	Name	Schema
Path	**configurationId** *required*	integer

Responses

HTTP Code	Description	Schema
200	get the configuration	Configuration
404	Not found	No Content

Produces

- `application/json`

Tags

- Configuration

edit an existing configuration given its id

```
PUT /configuration/{configurationId}
```

Parameters

Type	Name	Description	Schema
Path	**configurationId** *required*		integer
Body	**configuration** *optional*	The configuration to edit.	configuration

configuration

Name	Description	Schema
description *optional*	information about the configuration **Example** : `"EXTEND means that it will become longer by 3cm"`	string

Name	Description	Schema
hardwareId *optional*	the hardware this configuration belongs to **Example** : `"ON belongs to the hardwareID 1, which is RED LED"`	integer
name *optional*	the configuration name **Example** : `"ON"`	string

Responses

HTTP Code	Description	Schema
200	Edited	Configuration
404	Not found	No Content
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Configuration

delete a configuration by id

```
DELETE /configuration/{configurationId}
```

Parameters

Type	Name	Schema
Path	**configurationId** *required*	integer

Responses

HTTP Code	Description	Schema
204	deleted	No Content
404	Not found	No Content

Tags

- Configuration

Add a new Hardware to the system

```
POST /hardware
```

Parameters

Type	Name	Description	Schema
Body	**hardware** *optional*	The hardware to create.	hardware

hardware

Name	Description	Schema
description *optional*	additional info regarding the hardware **Example** : `"LED is a small electrical component that can emit light"`	string
gpio *required*	the gpio pin the hardware is installed at **Example** : `11`	integer
icon *optional*	URL image to desired icon in client **Example** : `"https://image.flaticon.com/icons/png/512/32/32750.png"`	string
name *required*	the hardware name **Example** : `"RGB LED"`	string

Responses

HTTP Code	Description	Schema
201	Created	Hardware
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Hardware

get all hardwares connected to raspberry pi

```
GET /hardware
```

Responses

HTTP Code	Description	Schema
200	return an array of hardware objects	< Hardware > array

Produces

- `application/json`

Tags

- Hardware

get a hardware by id

```
GET /hardware/{hardwareId}
```

Parameters

Type	Name	Schema
Path	**hardwareId** *required*	integer

Responses

HTTP Code	Description	Schema
200	get the hardware	Hardware
404	Not found	No Content

Produces

- `application/json`

Tags

- Hardware

edit an existing hardware given its id

```
PUT /hardware/{hardwareId}
```

Parameters

Type	Name	Description	Schema
Path	**hardwareId** *required*		integer
Body	**hardware** *optional*	The hardware to edit.	hardware

hardware

Name	Description	Schema
description *optional*	additional info regarding the hardware **Example** : `"LED is a small electrical component that can emit light"`	string

Name	Description	Schema
gpio *required*	the gpio pin the hardware is installed at **Example** : `11`	integer
icon *optional*	URL image to desired icon in client **Example** : `"https://image.flaticon.com/icons/png/512/32/32750.png"`	string
name *required*	the hardware name **Example** : `"RGB LED"`	string

Responses

HTTP Code	Description	Schema
200	Edited	Hardware
404	Not found	No Content
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Hardware

delete a hardware by id

```
DELETE /hardware/{hardwareId}
```

Parameters

Type	Name	Schema
Path	**hardwareId** *required*	integer

12

Responses

HTTP Code	Description	Schema
204	deleted	No Content
404	Not found	No Content

Tags

- Hardware

get all commands given a hardwareID

```
GET /hardware/{hardwareId}/command
```

Parameters

Type	Name	Schema
Path	**hardwareId** *required*	integer

Responses

HTTP Code	Description	Schema
200	return an array of command objects	< Command > array

Produces

- `application/json`

Tags

- Command
- Hardware

get all configuration for a given hardware

```
GET /hardware/{hardwareId}/configuration
```

Parameters

Type	Name	Schema
Path	**hardwareId** *required*	integer

Responses

HTTP Code	Description	Schema
200	return an array of configuration objects	< Configuration > array

Produces

- `application/json`

Tags

- Configuration
- Hardware

Add a new response

```
POST /response
```

Parameters

Type	Name	Description	Schema
Body	**response** *optional*	The response to create.	response

response

Name	Description	Schema
commandId *required*	the command id the response is for **Example** : `23`	integer
executionTime *optional*	the actual time raspberry pi executed the command	string (date-time)

Name	Description	Schema
isDone *required*	whether the command has been successfully done or not **Example** : `true`	boolean
isRead *optional*	whether the response has been read by the android client **Example** : `false`	boolean
message *optional*	optional message regarding the action **Example** : `"the command 243 was successfully executed!"`	string

Responses

HTTP Code	Description	Schema
201	Created	Response
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Response

get all responses

```
GET /response
```

Responses

HTTP Code	Description	Schema
200	return an array of response objects	< Response > array

Produces

- `application/json`

Tags

- Response

get a response by id

```
GET /response/{responseId}
```

Parameters

Type	Name	Schema
Path	**responseId** *required*	integer

Responses

HTTP Code	Description	Schema
200	get the response	Response
404	Not found	No Content

Produces

- `application/json`

Tags

- Response

edit an existing response given its id

```
PUT /response/{responseId}
```

Parameters

Type	Name	Description	Schema
Path	**responseId** *required*		integer
Body	**response** *optional*	The response to edit.	response

response

Name	Description	Schema
commandId *optional*	the command id the response is for **Example** : `23`	integer
executionTime *optional*	the actual time raspberry pi executed the command	string (date-time)
isDone *optional*	whether the command has been successfully done or not **Example** : `false`	boolean
isRead *optional*	whether the response has been read by the android client **Example** : `true`	boolean
message *optional*	optional message regarding the action **Example** : `"the command 243 was successfully executed!"`	string

Responses

HTTP Code	Description	Schema
200	Edited	Response
404	Not found	No Content
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Response

delete a response by id

```
DELETE /response/{responseId}
```

Parameters

Type	Name	Schema
Path	**responseId** *required*	integer

Responses

HTTP Code	Description	Schema
204	deleted	No Content
404	Not found	No Content

Tags

- Response

Add a new schedule

```
POST /schedule
```

Parameters

Type	Name	Description	Schema
Body	**schedule** *optional*	The schedule to create.	schedule

schedule

Name	Description	Schema
commandId *required*	the command id this schedule belongs to **Example** : `231`	integer
days *optional*	value between 0 and 127, representing 7 bits each bit correspond to a day in the order: sun mon tues web thurs fri sat **Minimum value** : `0` **Maximum value** : `127` **Example** : `120`	integer (int64)
time *optional*	the time of day the command shall be executed **Example** : `"13:50"`	string (time)

Responses

HTTP Code	Description	Schema
201	Created	Schedule
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Schedule

get all schedules

```
GET /schedule
```

Responses

HTTP Code	Description	Schema
200	return an array of schedule objects	< Schedule > array

Produces

- `application/json`

Tags

- Schedule

get a schedule by id

```
GET /schedule/{scheduleId}
```

Parameters

Type	Name	Schema
Path	**scheduleId** *required*	integer

Responses

HTTP Code	Description	Schema
200	get the schedule	Schedule
404	Not found	No Content

Produces

- `application/json`

Tags

- Schedule

edit an existing schedule given its id

```
PUT /schedule/{scheduleId}
```

Parameters

Type	Name	Description	Schema
Path	**scheduleId** *required*		integer
Body	**schedule** *optional*	The schedule to edit.	schedule

schedule

Name	Description	Schema
commandId *required*	the command id this schedule belongs to **Example** : `231`	integer
days *optional*	value between 0 and 127, representing 7 bits each bit correspond to a day in the order: sun mon tues web thurs fri sat **Minimum value** : `0` **Maximum value** : `127` **Example** : `120`	integer (int64)
time *optional*	the time of day the command shall be executed **Example** : `"13:50"`	string (time)

Responses

HTTP Code	Description	Schema
200	Edited	Schedule
404	Not found	No Content
405	Invalid input	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- Schedule

delete a schedule by id

```
DELETE /schedule/{scheduleId}
```

Parameters

Type	Name	Schema
Path	**scheduleId** *required*	integer

Responses

HTTP Code	Description	Schema
204	deleted	No Content
404	Not found	No Content

Tags

- Schedule

Definitions

Command

Name	Description	Schema
configuration *required*	the configuration desired for the hardware	Configuration
hardware *required*	the hardware this command is issued for	Hardware
id *required*	the identifier for the command in the database **Example** : `456`	integer (int64)
schedule *optional*	the schedule for this command. If none this command is immediate	Schedule

Name	Description	Schema
updateAt *optional*	time of creation/last updating	string (date-time)

Configuration

Name	Description	Schema
description *optional*	information about the configuration **Example** : `"EXTEND means that it will become longer by 3cm"`	string
hardware *required*	the hardware this configuration belongs to	Hardware
id *required*	the identifier for the configuration in the database **Example** : `3`	integer (int64)
name *required*	name of the configuration **Example** : `"ON"`	string
updateAt *optional*	time of creation/last updating	string (date-time)

Hardware

Name	Description	Schema
description *optional*	additional info regarding the hardware **Example** : `"LED is a small electrical component that can emit light"`	string
gpio *required*	the gpio pin the hardware is installed at **Example** : `11`	integer
icon *optional*	URL image to desired icon in client **Example** : `"https://image.flaticon.com/icons/png/512/32/32750.png"`	string
id *required*	the identifier for the configuration in the database **Example** : `1`	integer (int64)
name *required*	the hardware name **Example** : `"RGB LED"`	string

Name	Description	Schema
status *optional*	the current configuration for the hardware	Configuration
updateAt *optional*	time of creation/last updating	string (date-time)

Response

Name	Description	Schema
command *required*	the command executed resulting in this response	Command
executionTime *optional*	the actual time raspberry pi executed the command	string (date-time)
id *required*	the identifier for the response in the database **Example** : `865`	integer (int64)
isDone *required*	whether the command has been successfully done or not **Example** : `false`	boolean
isRead *optional*	whether the response has been read by the android client **Example** : `true`	boolean
message *optional*	optional message regarding the action **Example** : `"electrical error: hardware is not connected to the circuit"`	string
updateAt *optional*	time of creation/last updating	string (date-time)

Schedule

Name	Description	Schema
command *required*	the command where this scheduling info is for	Command

Name	Description	Schema
days *optional*	value between 0 and 127, representing 7 bits each bit correspond to a day in the order: sun mon tues web thurs fri sat **Minimum value** : `0` **Maximum value** : `127` **Example** : `120`	integer (int64)
id *required*	the identifier for the schedule in the database **Example** : `75`	integer (int64)
time *optional*	the time of day the command shall be executed **Example** : `"13:50"`	string (time)
updateAt *optional*	time of creation/last updating	string (date-time)