



**YILDIZ TECHNICAL UNIVERSITY
FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING
COMPUTER ENGINEERING DEPARTMENT**

SENIOR PROJECT

CLOUD BASED SMART HOME SYSTEM

Project Supervisor: Prof. Dr. Nizamettin AYDIN

Project Group:

12011009	Hüseyin Can ERCAN
12011092	Vedat Can KEKLİK

İstanbul, 2016

CONTENTS

SYMBOL LIST	v
ABBREVIATION LIST	vi
FIGURE LIST.....	vii
TABLE LIST	ix
PREFACE.....	x
ABSTRACT.....	xi
ÖZET	xii
1. INTRODUCTION	1
2. LITERATURE REVIEW	2
3. FEASIBILITY STUDIES.....	4
3.1. Technical Feasibility.....	4
3.1.1. Cloud.....	4
3.1.2. Main Box	5
3.1.3. Sensor Boards and Control Devices	6
3.2. Schedule Feasibility	6
3.3. Financial Feasibility.....	6
3.3.1. Cloud.....	6
3.3.2. Main box	6
3.3.3. Sensors	7
4. SYSTEM ANALYSIS	8
4.1. System Components	8
4.1.1. Main Box	8
4.1.2. Cloud.....	9
4.1.3. Sensors	10
4.1.4. Controllable Devices.....	10
4.2. System Specifications	10
4.2.1. Sensor Communication, Data Collection.....	10
4.2.2. Device Control	10
4.2.3. Alarm Conditions.....	10
4.2.4. Predefined scenarios	11
5. SYSTEM ARCHITECTURE DESIGN.....	12

5.1. Software Design.....	12
5.1.1. Web API & MQTT	12
5.1.2. Main Box Software	17
5.1.3. Device Node	29
5.2. Database Design	41
6. APPLICATION ANALYSIS	43
6.1. User	43
6.2. Home & Notifications.....	44
6.3. Devices.....	45
6.4. Task System.....	50
7. EXPERIMENTAL RESULTS	56
7.1. Failure Tests.....	56
7.1.1. Web API Failures.....	56
7.1.2. Main Box Failures	56
7.1.3. Device Failures	57
7.1.4. MQTT Broker Failure.....	57
7.2. Complete Test	57
8. PERFORMANCE ANALYSIS	58
8.1. Main Box – Device Performance.....	58
8.2. Web API – MQTT Broker Performance.....	60
9. CONCLUSION.....	61
REFERENCES	62
APPENDIX.....	64

SYMBOL LIST

ABBREVIATION LIST

API	Application Program Interface
EEPROM	Electrically Erasable Programmable Read-Only Memory
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
IR	Infrared
JSON	JavaScript Object Notation
LED	Light Emitting Diode
MCU	Microcontroller Unit
MQTT	MQ Telemetry Transport
NoSQL	Non Structured Query Language
PWM	Pulse Width Modulation
SQL	Structured Query Language
SRAM	Static Random-Access Memory
UART	Universal Asynchronous Receiver/Transmitter
Wi-Fi	Wireless Fidelity

FIGURE LIST

Figure 4.1 Use case diagram.....	8
Figure 5.1 System organization	12
Figure 5.2 UML class diagram of API	13
Figure 5.3 UML sequence diagram of MQTT connection	15
Figure 5.4 UML sequence diagram of MQTT publish	15
Figure 5.5 UML sequence diagram of MQTT subscribe.....	16
Figure 5.6 UML class diagram of MQTT Broker	17
Figure 5.7 UML class diagram of Main Box.....	18
Figure 5.8 Organization of Main Box software	19
Figure 5.9 Device command exchange	20
Figure 5.10 State transitions of device property	21
Figure 5.11 Device function call diagram	21
Figure 5.12 Task state diagram.....	26
Figure 5.13 Task listening loop diagram	28
Figure 5.14 UML class diagram of device node.....	30
Figure 5.15 Initialization process.....	35
Figure 5.16 Regular working cycle.....	36
Figure 5.17 E-R diagram	42
Figure 6.1 Login page.....	43
Figure 6.2 Signup page	43
Figure 6.3 Change password page	44
Figure 6.4 Home page.....	44
Figure 6.5 Notifications	45
Figure 6.6 Devices main page.....	45
Figure 6.7 Device detail page	46
Figure 6.8 Sensors of device.....	46
Figure 6.9 Chart Page of Sensor (Continuous Data)	47
Figure 6.10 Chart Page of Sensor (Discrete Data).....	48
Figure 6.11 Alert message, after clicking “LED: SET off”	49
Figure 6.12 “Pending...” label of “LED: off (SET)” disappeared	49
Figure 6.13 Alert message, after clicking “alarm (RUN)”	50

Figure 6.14 Tasks.....	50
Figure 6.15 New task creation	51
Figure 6.16 New task creation step #2.....	51
Figure 6.17 New task creation step #2.....	52
Figure 6.18 New task creation step #3.....	53
Figure 6.19 New task creation step #3.....	53
Figure 6.20 New task creation final step	54
Figure 6.21 Alarm was created by “Untitled Task”.....	54
Figure 6.22 Setting alarm’s silence time	54
Figure 6.23 After time setting operation.....	55
Figure 6.24 Mail alert that comes from alarm	55

TABLE LIST

Table 5.1 Classic Relational Storage	22
Table 5.2 Descriptions of fields	23
Table 8.1 Model table of Arduino boards	59
Table 8.2 Class number limits	59
Table 8.3 Sensor simulation results	60

PREFACE

We thank our families for their support and interest for our project. We thank our lecturer Prof. Dr. Nizamettin AYDIN for his support. We thank everyone for their interest in our ideas.

ABSTRACT

The main theme of the project is a cloud linked smart home system. Modularity, platform independency, improvability are the differences of the system from other home systems. At the lowest layer of the architecture self-introducing sensor-control nodes are in operation, at upper layers no predefined device configuration data or predefined control protocol are used.

All of the sensor and control equipment were grouped by nodes. Every single equipment class has programmed to introduce itself to the system by sending a JSON. The project has proved such dynamic architecture is possible with even microcontroller units. Simplistic object oriented design and several resource optimizations have resulted in suitable sensor-control nodes with high cost efficiency.

Also dynamic scenario design service has provided to user. Every sensor data and every command have included to scenario editor, by this way user has gained great freedom, creation of any desired scenario has become possible. The project has proved that self-introducing sensor-control systems are implementable with low cost, less sourced hardware.

ÖZET

Proje teması cloud bağlantısına sahip akıllı ev sistemidir. Modülerlik, platform bağımsızlığı, geliştirilebilirlik diğer ev sistemlerinden farklarıdır. En alt katmanda kendi kendini tanıtan sensör-kontrol düğümleri kullanılmıştır, üs katmanlarda önceden tanımlanmış cihaz konfigürasyon verisi veya önceden tanımlı kontrol protokolü kullanılmamıştır.

Tüm sensör ve kontrol ekipmanı düğümlerce gruplanmıştır. Her bir cihaz kendi kendini JSON göndererek tanıtmak üzere programlanmıştır. Proje bu ölçüde dinamik mimarinin mikro denetleyicilerle dahi mümkün olduğunu kanıtlamıştır. Basit nesneye yönelik dizayn ve birtakım kaynak optimizasyonları maliyet verimliliği göstererek, uygun sensor-kontrol düğümleri ile sonuçlanmıştır.

Ayrıca dinamik senaryo dizaynı kullanıcıya sunulmuştur. Her sensör ve komut senaryo editörüne dahil edilmiştir, böylece kullanıcı büyük ölçüde özgürlük kazanmıştır, istenilen her senaryonun oluşturulması mümkün hale gelmiştir. Proje kanıtlamıştır ki kendi kendini tanıtabilen sensör-kontrol sistemleri düşük maliyet ve az kaynaklı donanım ile mümkündür.

1. INTRODUCTION

A smart home system provides control and monitoring services to user. User can reach home's physical conditions' information and if the system is capable user can control the house. The market contains several smart home systems and specific components. Price of these systems are various.

By the recent developments wireless connection became cheap and easy to use technology. Also working with big data becomes easier day by day. Main purpose of this project is to create a cheap, easy to use, secure, modular home system.

Predefined scenarios, sensor devices, control devices, specific condition alarms, advanced monitoring are in our scope. Our aim is to make life easier.

2. LITERATURE REVIEW

Today's smart home systems have large variety. Cheap sensors and computational power changing the approach. Easy and cheap internet connection makes home systems accessible. Water sensing, gas detection, light sensing, sound detection, vibration detection, movement sensing are possible with various sensors and devices. These are very common and standard expectations. But systems at the market have very specific functions. Speakers, water meters, thermostats are examples. We are aiming to create a platform.

Single and incompatible units of different manufacturers create problems for complete systems. Also these units do not aim to construct a complete and wide system. They are cheap, actually not really cheap, single function modules for specific needs. But we are aiming a compatible, modular system framework. We are aiming to use security precautions for inner home communication and internet connection.

In our opinion supervisory control node is needed. Otherwise the access and management becomes dangerous. Also multiple sub nodes need a supervisory node. A single board computer and a microprocessor can do different jobs. A single board computer can manage security but a microprocessor, for example Arduino Uno is an easy target for threads. Low processing power weakens the security.

Several small units, which connect directly to cloud, makes management impossible. Connection establishment and reconnection becomes very hard to handle. But for a single unit it is not a problem. Shortly, we are planning to use a supervisory node for the sake of security and scalability.

Sub nodes of the system has specific function like other systems. But on the other hand reducing costs are possible with multi-sensory nodes, they can reach the home network by, the same Wi-Fi connection for example. This is our approach about sensing. Spending a node for every sensor type is pointless for wide networks. But electro mechanic aspects of system are different. We cannot unite them under a specific node such as switches, lamps, plugs and similar control nodes.

We separated nodes into two groups. Control nodes and sensor nodes. But a node can be both. The JSON data format resolves the problem. The supervisory node and cloud can understand what the selectable functions are and what is coming from sensor node. With this feature an educated user can create and add his/her specific nodes to system. For example nobody sells “Geiger counters” for smart home systems but for our system it is just a sensor node.

3. FEASIBILITY STUDIES

In this section, technical feasibility, financial feasibility and time feasibility is discussed.

3.1. Technical Feasibility

In this section, software and hardware requirements of all components of the system is discussed.

3.1.1. Cloud

Cloud Software Feasibility: Cloud is a general name for internet based resource services (web server, database server, storage, etc.). In this project, system capacity can be increased with cloud. System must be always online. Therefore cloud-based services are the right choice. Other advantages of cloud are easy installation and fast recovery. A conventional web server may be more costly in terms of time and money. Also, cloud storage provides flexibility for future applications (video, audio etc.).

A Linux-based operating system will work on cloud. Linux is widely used in server market. And many popular and important software (web servers, database servers etc.) works on Linux. These are the main reason of using Linux on cloud environment. Also Linux is preferred because it is open source and free.

A platform named NodeJS [1] is used on cloud as web server. In fact, NodeJS is not a web server. It is a runtime environment for non-blocking I/O. But some modules that it contains helps using platform as a web server. JavaScript is being used to program NodeJS. Asynchronous programming, event-driven structure is a plus for NodeJS. NodeJS is open source, cross-platform and free.

MQTT [2] is preferred for the connection between main box and cloud. MQTT is a protocol which is invented before “Internet of Things” concept. But, MQTT provides some features (low overhead, offline messaging and more) which fits IoT. Another important feature of MQTT is publisher/subscriber pattern. This feature makes connection and messaging much easier than other protocols. NodeJS environment provides MQTT client and MQTT broker modules.

Another component which will work on cloud is database. A NoSQL database is preferred instead of SQL database. First reason of this is data structure variety. Different sensors send different data. A sensor sends 1 decimal number. But another one sends 3 decimals and 1 string. This is not appropriate for schematic structure of SQL databases. Also NoSQL is easily scalable and distributable. And this is a plus for IoT concept. Preferred NoSQL database is MongoDB for this project. MongoDB is free and open-source.

Cloud Hardware Feasibility: Service provider is responsible for hardware of the cloud platform. At the beginning the following configuration is preferred:

- 1 GB RAM
- 1 CPU Core
- 24 GB Storage
- 2 TB Transfer
- 40 Gbps Network In
- 125 Mbps Network Out

3.1.2. Main Box

Main Box Software Feasibility: An embedded Linux based system works on main box. Main box provides the data transmission between home and cloud. A NodeJS based MQTT client works on main box. It processes incoming commands, and sends data to the cloud. Main box checks sensor data periodically. When an unusual thing happens, main box produces an alarm and sends this alarm signal to user via different ways (web interface, email etc.).

Main Box Hardware Feasibility: Requirements for main box Linux board are:

- Single Core / Multi Core ARM/x86 CPU
- 2GB RAM
- 5GB Storage
- Ethernet
- Wi-Fi (802.11 b/g/n/ac)

These specifications are not minimum. In order to provide stability and robustness these are enough.

3.1.3. Sensor Boards and Control Devices

Sensor Board Software Feasibility: The fundamental component of these devices is Arduino. A C-like programming language is being used to program Arduino.

Sensor Board Hardware Feasibility: Sensor board contains, a microcontroller card (Arduino), a sensor kit (temperature, humidity, movement, etc.) and a Wi-Fi chip (ESP8266).

3.2. Schedule Feasibility

A Gantt diagram (See Appendix A and Appendix B) has been drawn for schedule feasibility. Diagram shows the schedule.

3.3. Financial Feasibility

In this section cost of components is discussed.

3.3.1. Cloud

Service provider is responsible for hardware of the cloud platform. For example a system that provides specifications below can be purchased for 10\$ per month.

- 1 GB RAM
- 1 CPU Core
- 24 GB Storage
- 2 TB Transfer
- 40 Gbps Network In
- 125 Mbps Network Out

This specifications depends on number of main box, connected to the cloud. More main box means more system load. In this case system will need more hardware capacity in order to handle this load generated by main boxes.

3.3.2. Main box

Main box is a Linux-based minicomputer. It is possible to buy these cards in 35-200\$ price range. Batteries and/or UPS can be added to main box system in order to setup a persistent connection with cloud. Also, a 3G Modem is required for standalone internet communication. Price of the 3G service is determined by service provider (GSM).

3.3.3. Sensors

An Arduino and an ESP8266 chip are needed for each sensor device. Arduino is a microcontroller card which has different types. Sensor chip + Arduino + ESP8266 can be purchased for approximately 30\$. These devices may need battery. Battery will increase unit cost. Using residence's electricity instead of battery is also possible.

4. SYSTEM ANALYSIS

In this section, functionalities of the system is discussed.

4.1. System Components

This part explains system components.

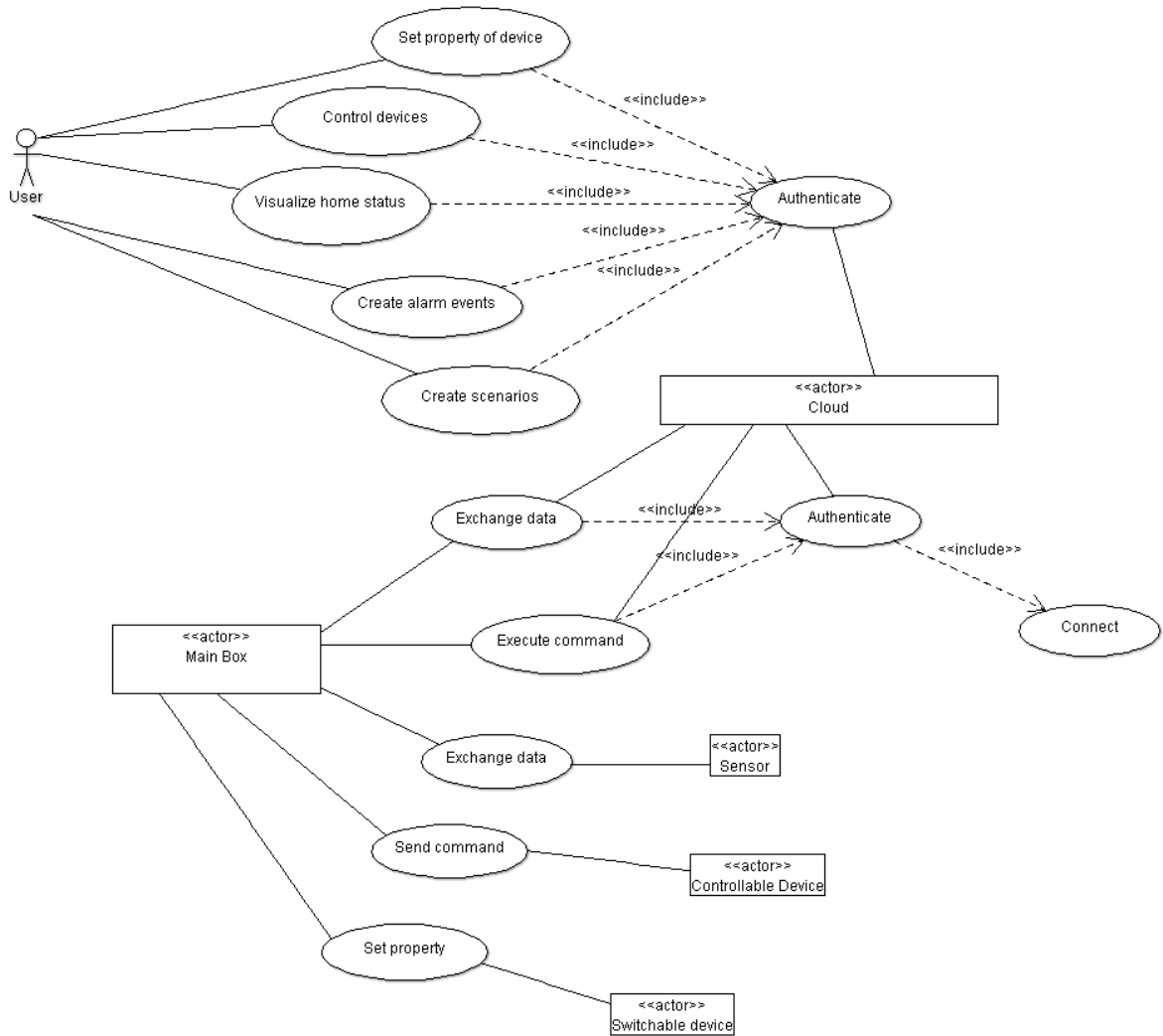


Figure 4.1 Use case diagram

4.1.1. Main Box

Main box is the supervisory components of the home network. The hardware is a single board computer which uses Linux OS.

Responsibilities of Main Box are,

- Communication with other sub nodes
- Provisioning of control services
- Transmission of data to cloud
- Creation of alarms

Use cases of main box examined under processing power, security and management titles.

Processing Power: Nodes of the system have to send results of measurements to cloud. Also they can accept basic commands for management like Wi-Fi password change etc.

A standalone node cannot provide all of them. Limited processing power and unstable connection are the problems of the standalone connection. Processing becomes a problem for a minimal system.

Management via main box is an optional solution. High processing power makes easy to manage.

Security: An unprotected, low level device is a possible target for attackers. Low processing power weakens the security also. Even one node is important, node's data or control commands create great dangers.

Protecting a single location is much easier than protecting several locations.

Management: Management from a single point is much more stable and easy.

4.1.2. Cloud

Cloud stands between users and main boxes. Holds records of incoming data from main boxes. Transmits user commands to main boxes. User can send commands to house by web interface via cloud.

4.1.3. Sensors

Sensor modules has three basic component:

- Arduino
- ESP8266 Wi-Fi module
- Sensor circuit (which connected to Arduino)

Arduino send the measurements to main box. The idea to be independent from Arduino. User can use any device which connects main box.

4.1.4. Controllable Devices

IR controlled devices like TVs, air conditioners, controllable plugs, switches are in our scope.

4.2. System Specifications

This part explains fundamental characteristics of the system.

4.2.1. Sensor Communication, Data Collection

Sensor nodes send collected data to main box. The main box processes the received data and sends it cloud in a formal format. Cloud is responsible for data monitoring, reporting and statistical studies.

4.2.2. Device Control

Nodes of the system has to be controllable. The architecture and real world requires this. For example Wi-Fi password change operation is mandatory for a Wi-Fi connected device. Without this control we have to reprogram it for every Wi-Fi password change.

4.2.3. Alarm Conditions

Different users have different needs. An unexpected movement or lack of movement can create problems for different user.

4.2.4. Predefined scenarios

Predefined scenarios makes the home easy to look after. User can control house better with selected scenarios. For example a holiday scenario can shut down every device in house and guard the house against intrusion.

5. SYSTEM ARCHITECTURE DESIGN

In this section, design details of the system is discussed.

A brief diagram of system organization is provided (see Figure 5.1)

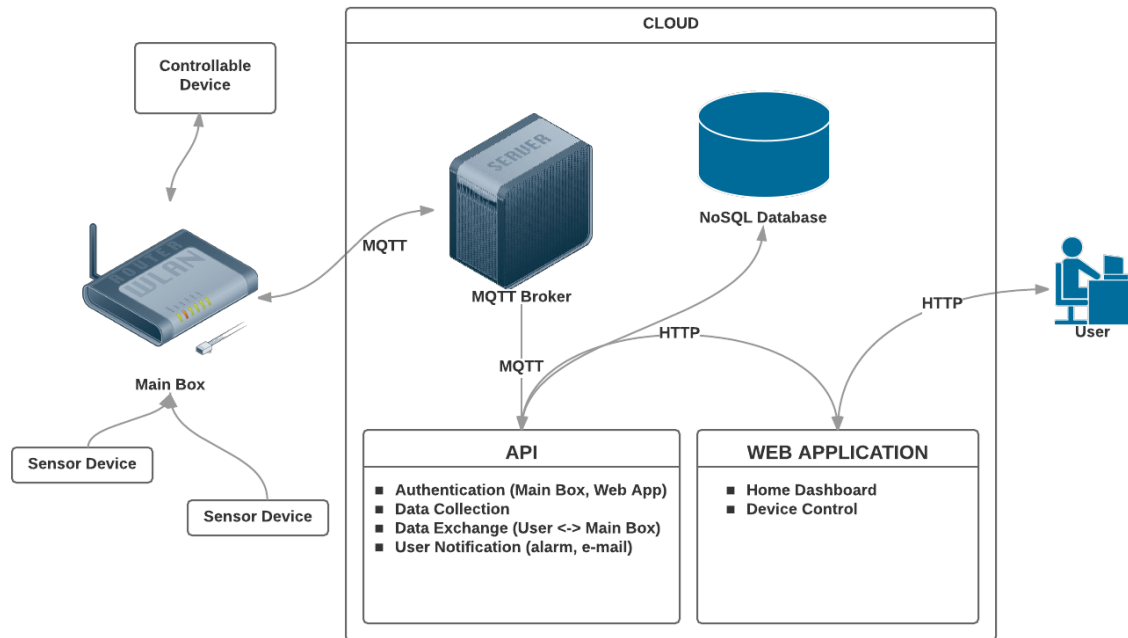


Figure 5.1 System organization

5.1. Software Design

The system has different parts, and each part has its own software. The system roughly has two main parts, **server** and **main box**. Server and Main Box is discussed in following sections.

5.1.1. Web API & MQTT

API: API is the central data transaction component of the system. Events, logs, sensor data and authentication is controlled and provided by API. Modules of API are:

- Devices
- Tasks & Alarms
- Predefined Scenarios
- Notifications
- Authentication & Authorization

Design of API is object oriented as shown **Figure 5.2**.

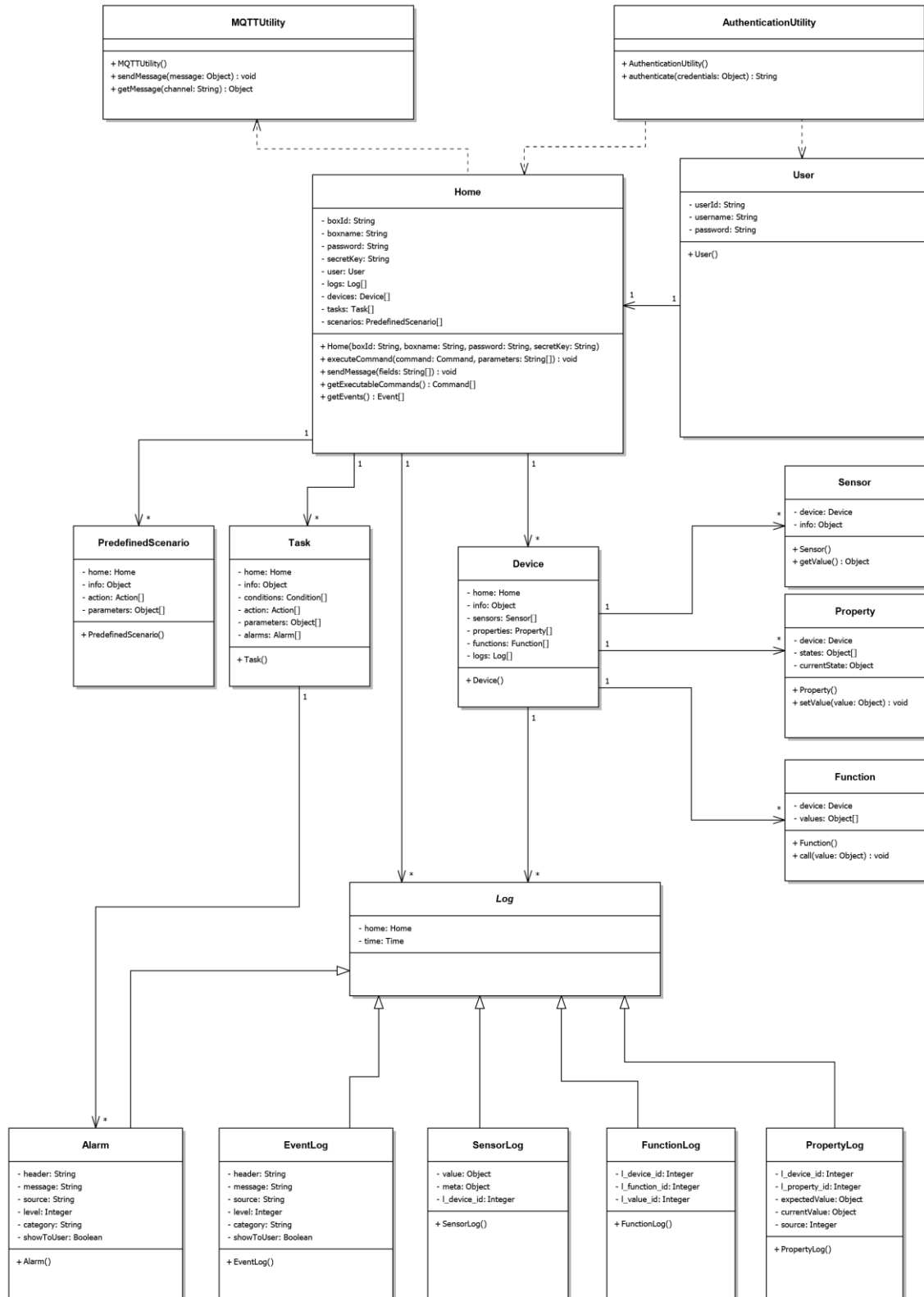


Figure 5.2 UML class diagram of API

MQTT: MQTT is a lightweight communication protocol. In protocol, there are clients and a broker. A client connects to the broker. A client may “subscribe” a channel. And it may put messages into any channel.

Main Box and the Web API communicates via MQTT. They both connect to the “MQTT Broker” as clients. A residence gets messages from “home/{HOME_ID}/in” channel. And puts messages into “home/{HOME_ID}/out” channel. Web API listens “home/+out” channel. “+” is a wildcard which means “listen all home/out channels”.

MQTT Message Recovery: Main box may lose its connection because of network issues. And user may still want to send messages (open lights, turn off fan etc.) to its residence remotely. Hence, system needs to store messages. Otherwise, messages that sent by user when main box is offline would be lost. In order to avoid this message loss, MQTT broker stores messages in a database. And sends those messages to home when it connects again [3].

MQTT Command Structure: MQTT provides binary messaging infrastructure. There is no standard message type. In this system, messages standard is,

```
command: 'COMMAND_NAME',

body: {
    'key': 'value',
    .
    .
    .
}
```

Connection & Messaging: There is an authentication mechanism for MQTT client connection. In order to start communication, all clients and Web API must send their credentials to the MQTT Broker [4] [5].

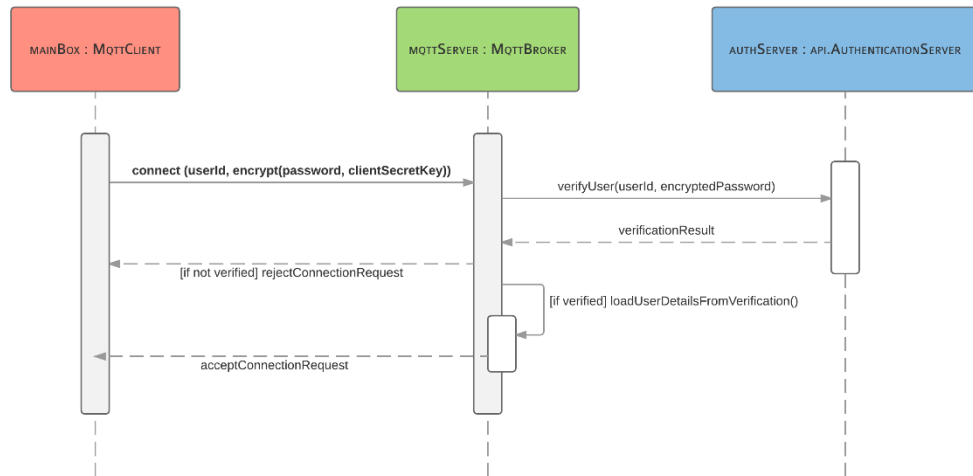


Figure 5.3 UML sequence diagram of MQTT connection

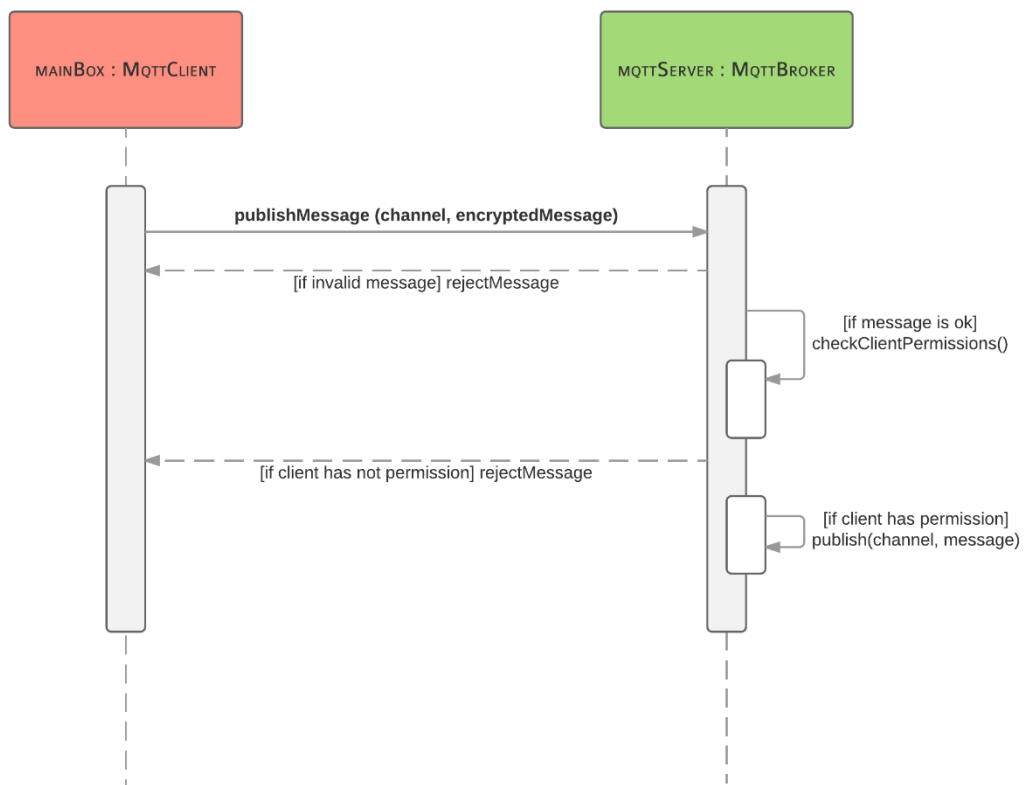


Figure 5.4 UML sequence diagram of MQTT publish

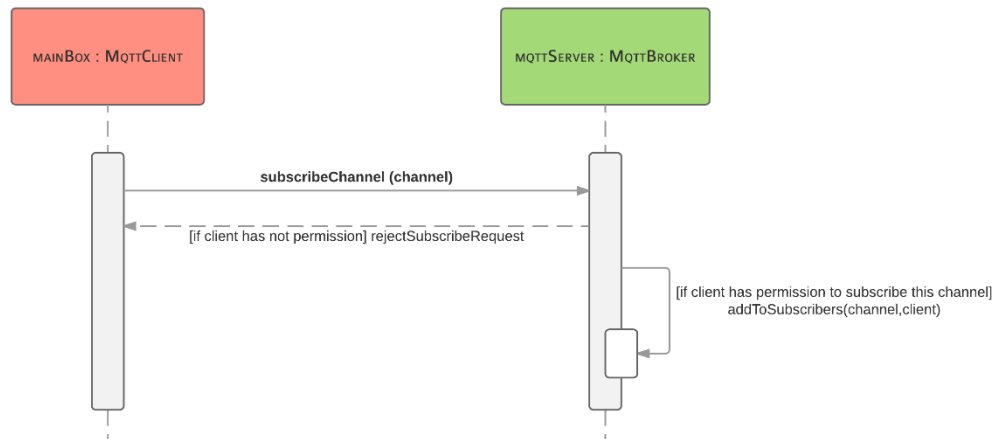


Figure 5.5 UML sequence diagram of MQTT subscribe

A publisher sends message into a specific channel. Before publishing, MQTT broker checks for publisher's identity. If user is allowed, he sends message. Otherwise, he gets rejection message.

A subscriber gets all messages that passed into a specific channel. Subscription isn't necessary in order to pass message. These are separate functionalities.

MQTT broker checks subscription requests, if user is allowed, he gets messages. Otherwise, he gets rejection message.

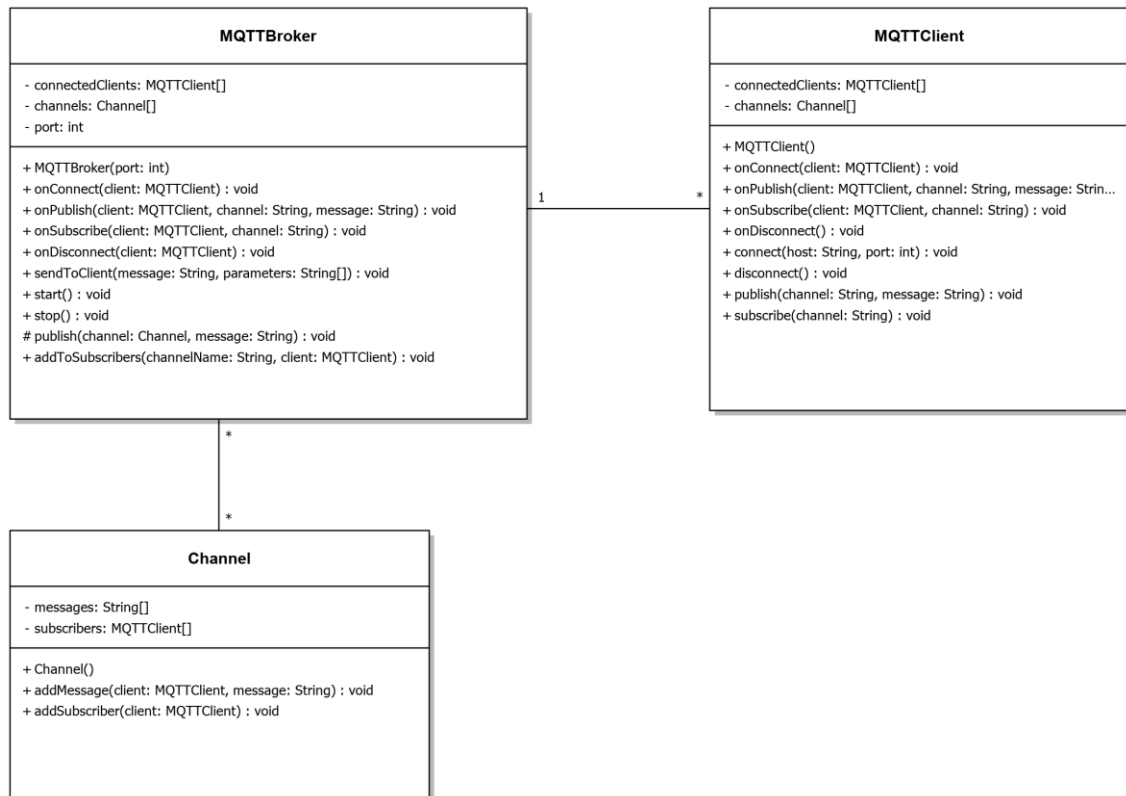


Figure 5.6 UML class diagram of MQTT Broker

5.1.2. Main Box Software

Main Box is the bridge that stands between devices and the API. Fundamental functions of main box are:

- Device interaction (control, collection of data, centralization)
- Periodic listening of environment for user tasks

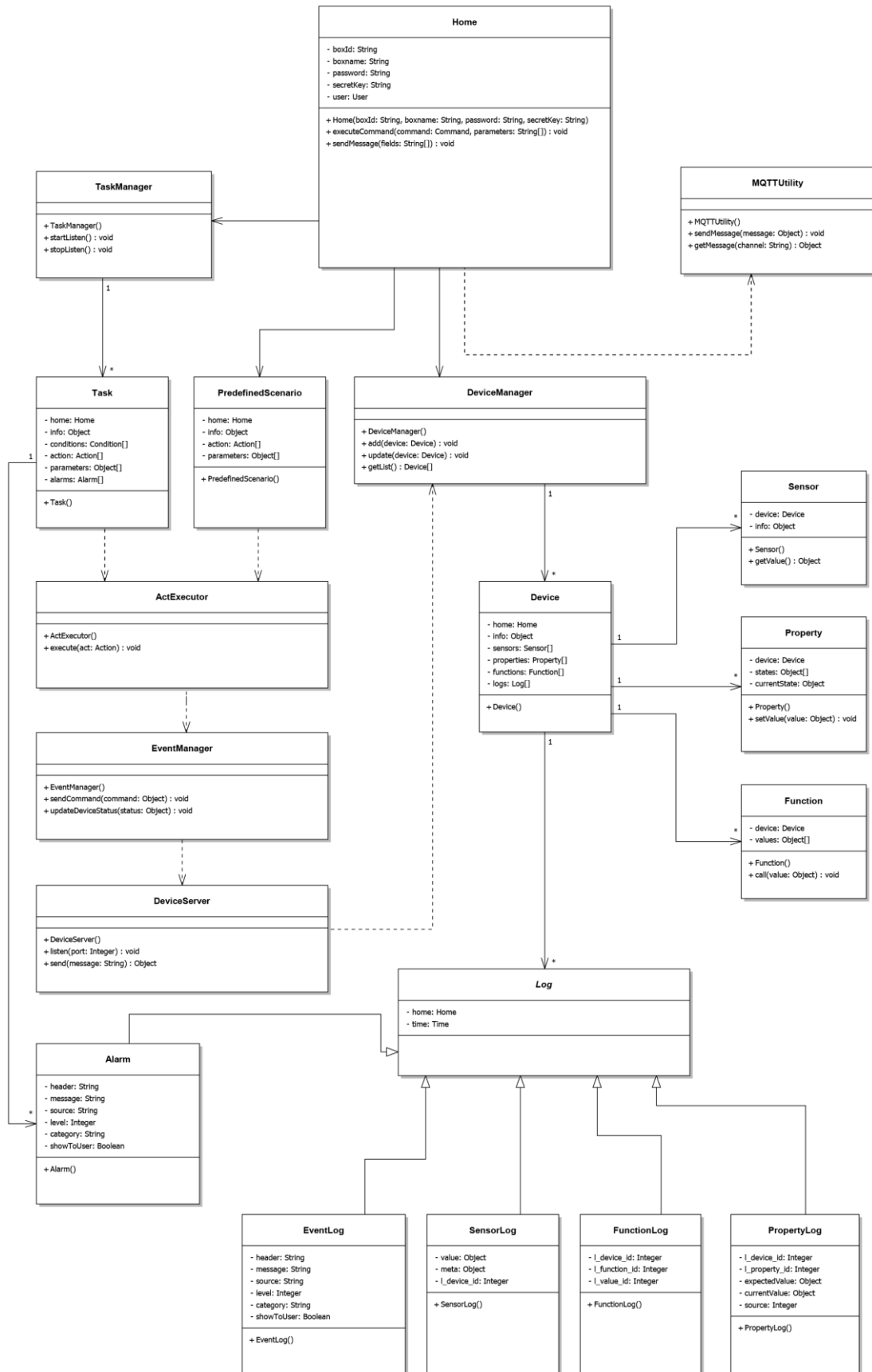


Figure 5.7 UML class diagram of Main Box

Device Interaction: There are multiple devices that connect to main box. Main box registers all connected devices to the process memory at bootstrap stage. Main box starts a HTTP server for device communication. Device periodically updates its status in the main box over this HTTP server. Also API commands (property set, function call) is sent within HTTP response that given by HTTP server. Device HTTP server, sends only one API message in one HTTP response.

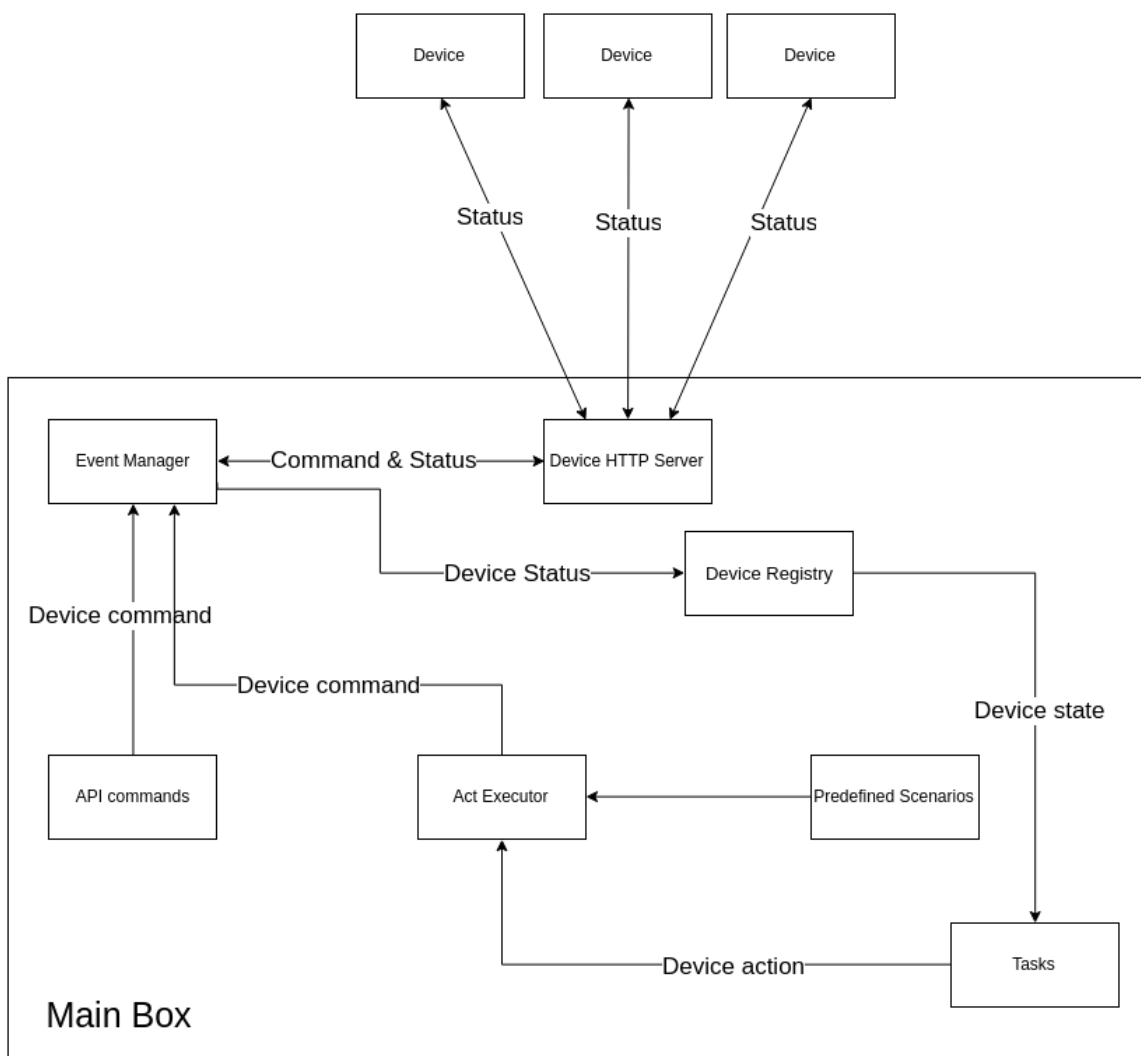


Figure 5.8 Organization of Main Box software

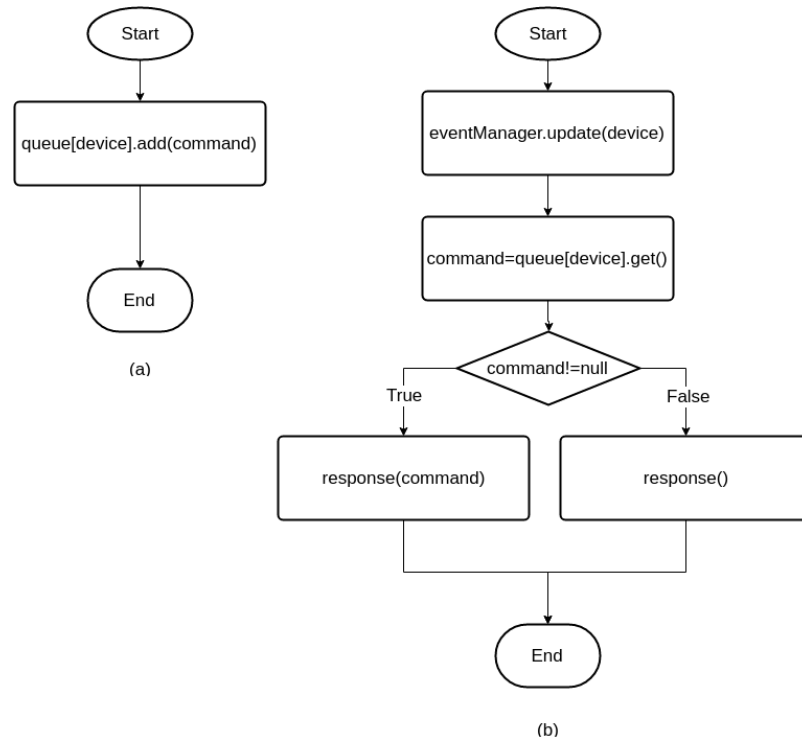


Figure 5.9 Device command exchange

(a): Incoming command is added to device queue

(b): When device starts a new exchange, event manager updates device status with incoming data from device. Then HTTP server gets a new command from queue. If command is not null, then it sends response with command to the device.

Device Properties: Property is an attribute of a device. Property concept consists of two terms: property name and possible states. A device may have multiple properties. Current state of property is stored by Web API, main node and device itself.

Architecture of device property system: Property set operation is 3-phased. In the beginning, API sends set command to main box. Then main box puts this request into the related device's command execution queue. Finally, device sends result of execution to main box, and main box sends it to API.

For example, let's say a LED light is a property of a device. And this light has got three different states: on, off and blink. Current state of this property is off.

Property set operation may fail in some cases (hardware failure, network failure etc.). In those cases, system sends failure information (current state of that property) to the API.

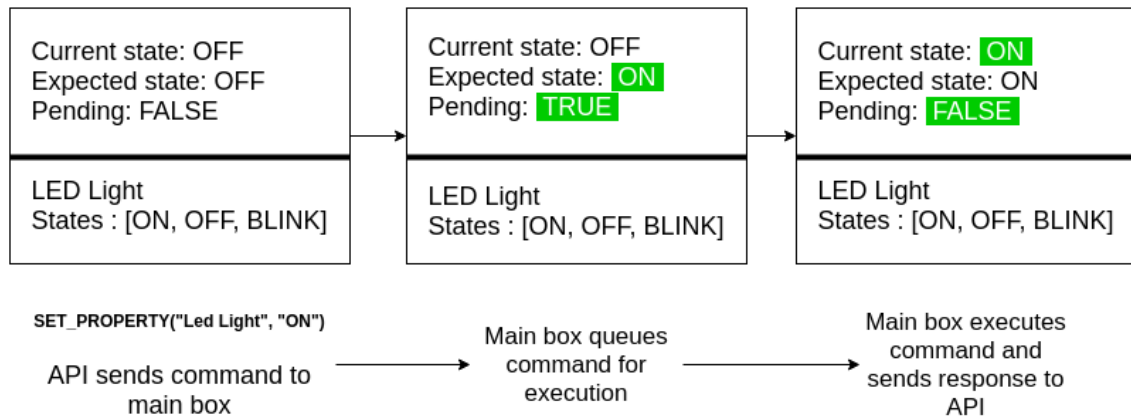


Figure 5.10 State transitions of device property

Device Functions: Function concept consists of two terms: function name and possible values. Device function is similar to device property. But there is a fundamental difference between them. A property has got states, but a function is stateless. User may call a function over API. Main box receives the command, puts into queue. After execution, device sends its response to main box. Return value of function is implicit. Users cannot see any feedback after execution.

Function system is a good option for controllable devices that don't have feedback mechanism. For example an infrared controller only sends signal to controllable device. And controllable device returns nothing. So there is no return value or state in this case.

Example: Let's say IR control is a function of a device. And this function may accept two different values: on, off.

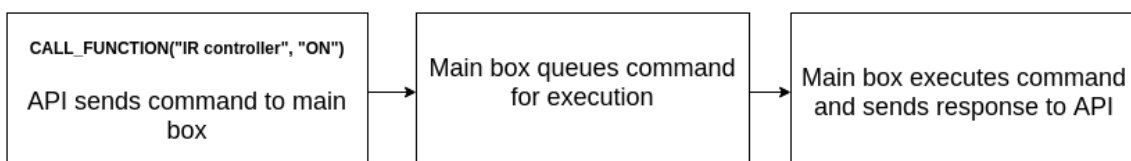


Figure 5.11 Device function call diagram

Sensor data collection: Sensors produce time-based data. This data can be stored as (time, type_of_data, value) tuple in a relational database.

Table 5.1 Classic relational storage

time	type_of_data	value
2015-11-02 14:20:30 397	temperature	12.3
2015-11-02 14:21:40 560	temperature	12.6

This storage approach is also feasible for document-oriented NoSQL databases like MongoDB.

But there are some challenges. Because, there is a big amount of sensor data. Repetition of “type_of_data” column for every single record is a problem in the first place. But this is a requirement for sake of data uniqueness. It must be kept somewhere. Otherwise, the data would be meaningless.

Document oriented databases provides a more elegant solution possibility [6]. The data can be stored in a grouped form instead of relational storage approach. In grouped form, hourly data is stored as a single document. A database document represents single sensor's data.

```
home : ObjectId("56a237866b521bb76b554ba2"),
l_device_id : "536",
l_sensor_id : "28",
timestamp : ISODate("2015-12-25T15:00:00.000+0000"),
meta : {
  total : NumberInt(120),
  sum : 59.831048742728875,
  max : NumberInt(1),
  min : 0.003288469150121426
},
values : [
  0:{
    meta:{
      "total" : 2,
```

```

        "sum" : 1.3356159098294427,
        "max" : 0.8187119948743449,
        "min" : 0.5169039149550977
    }
    sec:[
        0: 0.5169039149550977,
        .
        .
        .
        59:0.6189485649
    ]
}
.
.
.
59: {...}
]

```

Table 5.2 Descriptions of fields

home	Unique home id
l_device_id	The device that sensor works on
l_sensor_id	Unique sensor id
timestamp	Represents the date of sensor data. Resolution is an hour
meta	Meta is total values for one hour
values	Value array stores 60 elements constantly which refers to 60 minutes in an hour

A cell of values array (corresponding to a minute) has two fields: “meta” and “sec”. Meta structure helps to calculation of average values. In the absence of the “meta” structure, all seconds in an hour must be summed for every calculation. In order to avoid that, this concept was adopted. Average value for an hour = meta.sum/meta.total

“meta” is identical with meta that field of parent document in terms of structure. Design goal of “meta” field is fast-access for both contexts. Meta values (total, sum, max, min) are always up-to-date. Because every write operation updates that values.

“sec” is a container for every seconds in a minute. It has 60 elements constantly (for 60 seconds). Seconds are not for data visualization at this moment. Charts are generated with hourly metadata. But second-level value collection can be necessary for future applications.

Sensor data writing procedure:

1. Collected data arrives to API from main box.
2. API sends a query to database for the request with identity tuple (home, l_device_id, l_sensor_id, timestamp).
 - a. If the document exists, then API updates the second of minute
 - b. If the document doesn't exist, then API creates a new document, and writes fields.
3. The update query also recalculates “meta” values for given (minute, second) tuple, and metadata of hour.
4. Update operation is completed.

System does only 1 document insertion for one hour, and does 3600 update (maximum) operation for seconds in an hour. This reduces database resource allocation overhead.

Sensor data reading procedure: At first database retrieves appropriate data from collection. And then groups it for given resolution. The system provides 4 level of resolution. Year, month, day and hour.

1. Query arrives to API over user interface.

```
home : ObjectId("56a237866b521bb76b554ba2"),
  l_device_id : "536",
  l_sensor_id : "28",
  timestamp : {
    $gte : ISODate("2015-12-01T00:00:00.000+0000"),
    $lt : ISODate("2015-12-21T00:00:00.000+0000")
  }
```

2. Database group's result set with selected resolution.
3. Database calculates average for each selected element with their metadata.

Handling Discrete Data: In the system, there are two types of sensor data: continuous and discrete. Continuous data was discussed at previous part. Examples of continuous data are temperature, humidity, etc. Examples of discrete data are Existence of gas, Sound level, Motion level, etc.

Fundamental difference of these two types is continuous data can be expressed as its average, but discrete data cannot be. For example, existence of gas can be only false or true.

There isn't no third possibility. There are 2 discrete levels. Due to this reason, it is impossible to calculate average of these values. Consequently, storage of discrete sensor data must be different. In order to solve this problem, structure of "values" field was changed as below. In this storage approach there is no time-based preallocation.

```
"values" : [
  {
    "date" : ISODate("2016-04-29T00:02:02.942+0000") ,
    "value" : false
  },
  {
    "date" : ISODate("2016-04-30T00:05:01.576+0000") ,
    "value" : true
  },
  .
  .
  .
]
```

Tasks: Task system is a listener system for physical events. In this system, user may create task from Web Interface. At main box boot, API sends all tasks to main box. Listening operation is handled by main box. If all conditions are met, system creates alarm and executes related actions.

$3 \times 60 = 180$ seconds

Period = 2 seconds

Minimum number of occurrences = $180/2 = 90$

System counts condition occurrences. If number of occurrences is great or equal to 90 then action execution will be started. This can be called as “task match”.

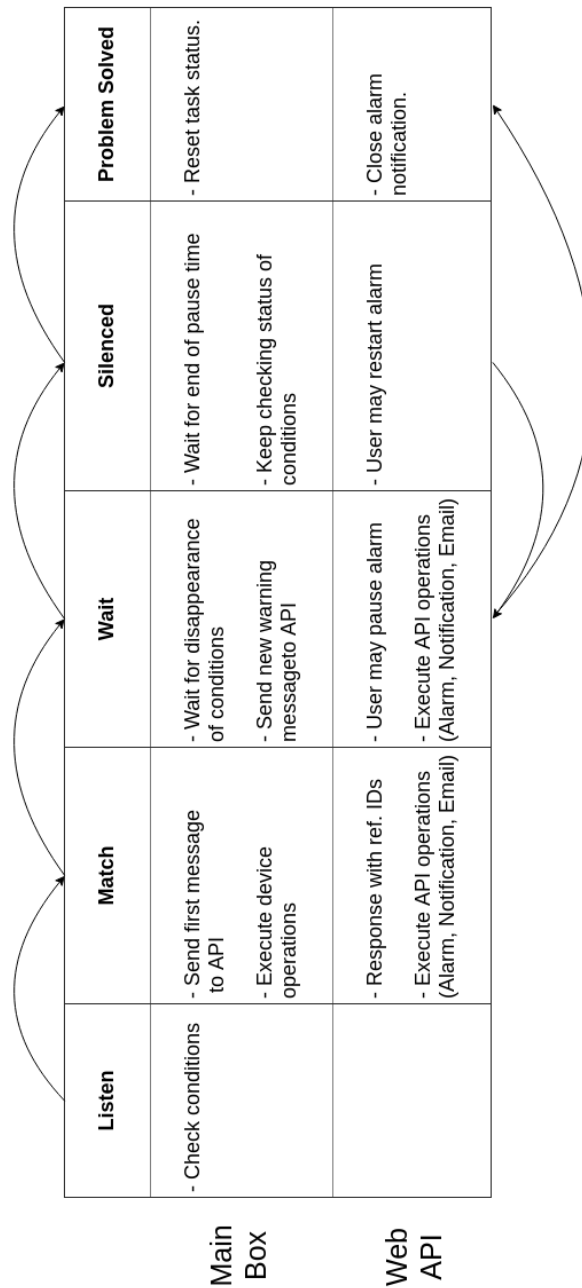


Figure 5.12 Task state diagram

A task consists of two parts: filters, actions.

Time Filters: There are two types of time filters: day and time. Seven days of a week can be selected as “active day”. Also user may define time range for task. For example, a user may arrange work times as:

“Listen this task between 12:00 and 23:59 on (Monday or Tuesday or Saturday or Sunday)”

If time is appropriate for this filter, then task will be listened by main box.

Device actions: Sensor and property values can be used as task filters. For example, user may select temperature sensor value and light state as filter.

(necessary) Temperature \geq 20

(necessary) Lights are on

This expression means “if temperature is great or equal to 20 and lights are on, then execute action”.

Wait time of a filter is also configurable. Wait time is basically time threshold for filter. For example, “If this conditions happens along 4 hours at least, then do something.”

Send message action: System sends a text message to email addresses defined by user. Set a property or call a function: User may select any number of executable action. For example, “if all conditions are met, then open light and close IR controllable device”.

Task Alarms: Besides execution of action set, the system also creates an alarm in order to warn user. This alarm message can be seen under “Alarms” section. User may set a silence interval for an alarm. Because alarm will produce messages periodically until task conditions disappear.

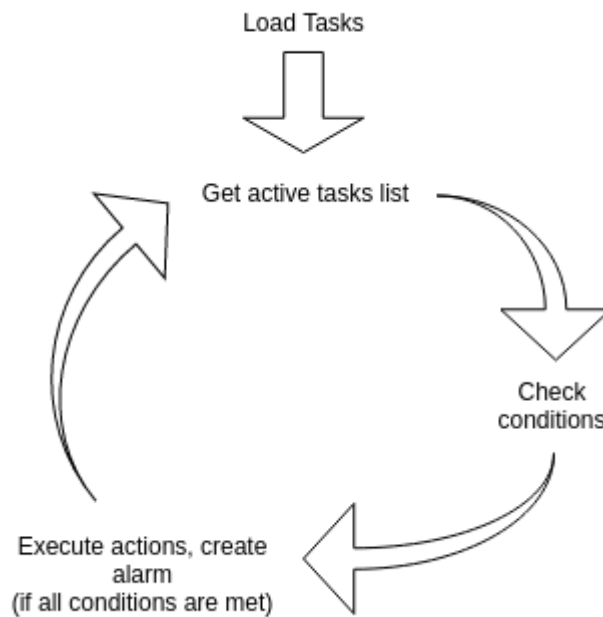


Figure 5.13 Task listening loop diagram

Predefined Scenarios: Predefined scenarios are basically sets of commands. A scenario may contain any number of property set and any number of function calls.

User may create a scenario by using Web Interface. All scenarios are sent by API to main box on device boot. A scenario can be executed by using Web Interface.

Notification System: Notification Center lists important events that happens in residence. Types of events are:

- Main box connect/disconnect events
- Task system alarms

Basically, notifications are produced by main box event system. If main box event message has “showToUser=1” parameter, then that event will be appeared on the Notification Center. Property set, function call, sensor data update events are not shown directly in Notification Center. Notifications are tagged with importance level tags. For

example “connection of main box” is just an “info” event. But “disconnection of main box” is a critical event.

5.1.3. Device Node

Device Node is the most basic component of the system, it is responsible from providing the sensory data and execution of the Main Box commands. Device Node has to have a proper communication channel, it can have sensor components (Sub Sensors) or control components (Sub Controllers). A temperature sensor, a PWM driven motor or a high voltage control relay are the examples of environmental devices. Current design and implementation has an HTTP communication channel which established via ESP8266 [7]Wi-Fi module and based on Arduino Uno Rev3 [8].

Sensor Nodes interferes only with the Main Box. A group of Device Node and single Main Box creates a home’s control and feedback system. Device Nodes contain the identification information, at the booting process every Device Node sends its identification information. By this way any node specific data, control specification or a specific sensor measurement process becomes hidden from the Main Box.

Device Node’s generic introduction process widens the possibility of sensors and control equipment. Self-introducing sensor interface and self-introducing control interface provide freedom to user. Every specific aspect of the sensor or control equipment is embedded in its own class which runs on the Device Node hardware. Every Sub Sensor and Sub Controller class have to implement an interface for this mechanism. Specific implementations for different sensors and control equipment become possible with this way. Any programmer can develop his/her own Sub Sensor or Sub Controller class for the system on one condition, class has to implement Sub Sensor or Sub Controller interface.

Device Nodes have to introduce themselves after the booting process. Current implementation and design presumes the Device Node’s hardware resources are very limited and communication channel is very unstable. Because the system approach was aiming to create an inexpensive and modular system at the very beginning.

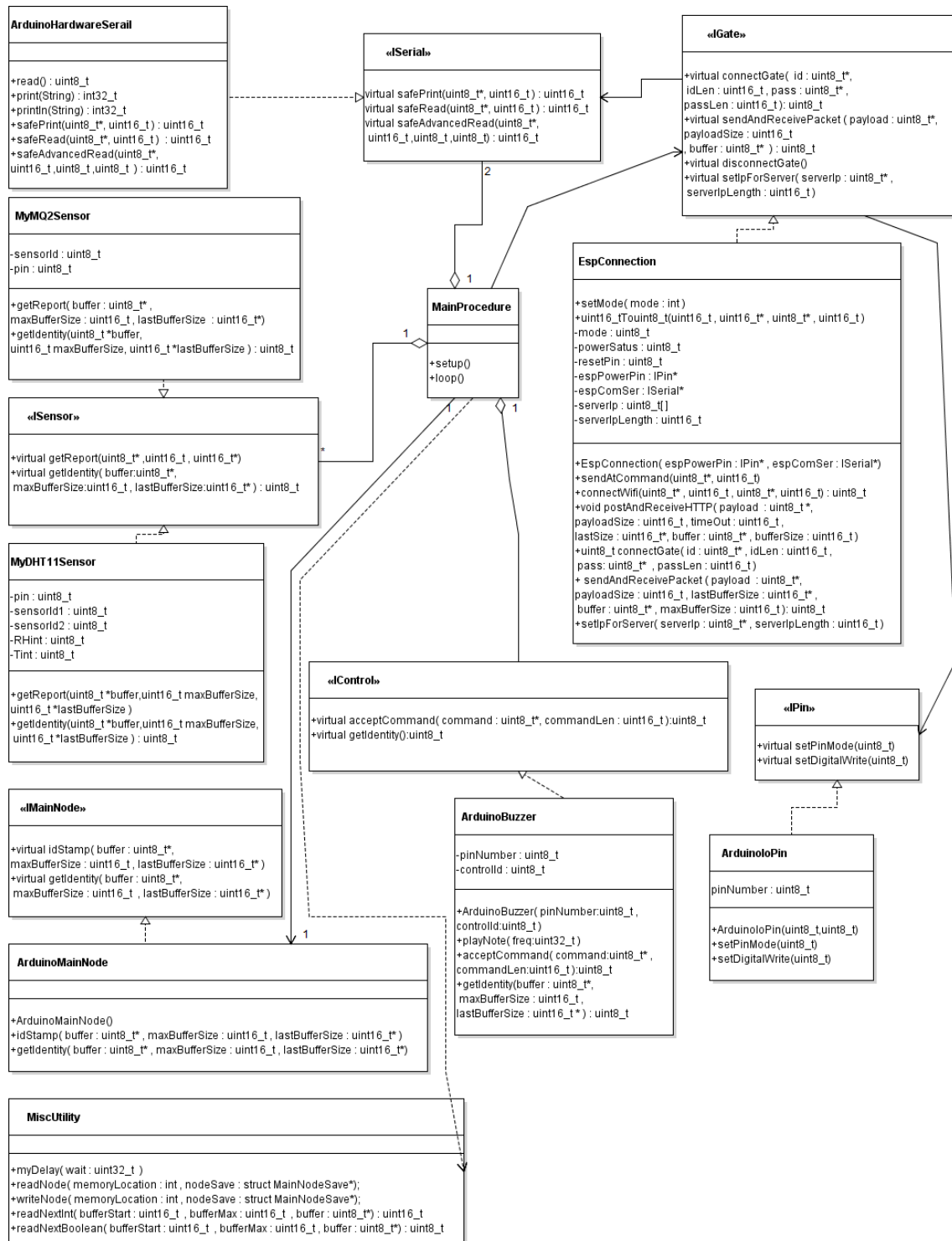


Figure 5.14 UML class diagram of device node

Every communication cycle device Node sends sensor and state data and takes the box command. Sensor data and state data are in the JSON [9] format, box commands have a specific format.

Further part explains of the Device Node and related units.

Software Design of Device Node: Device Node implements a platform surface for Sub Controllers and Sub Sensors. Software design depends on two most important interface [10]. They are IControl and ISensor. Control classes and sensor classes have their own specifications. For example DHT11 [11] has a serial interface, MQ2 [12] gas sensor has analog output, relay controller has a digital input, and some buzzers can take analog input. One single class cannot afford all of these specifications. A design like this is impossible to maintain and develop. Firstly all of these sensors and controllers separated from main code. This separation provided via interfaces. By this way all of killer details buried in the invisible classes. A Device Node can run without any of these specifications, it interests only with the implemented interface methods.

IController:

```
virtual uint8_t getStateReport(uint8_t *buffer,uint16_t
maxBufferSize, uint16_t*lastBufferSize);
virtual uint8_t acceptCommand(uint8_t *command , uint16_t
commandLen);
virtual uint8_t getIdentity(uint8_t *buffer,uint16_t
maxBufferSize, uint16_t *lastBufferSize
```

ISensor:

```
virtual void getReport(uint8_t* ,uint16_t , uint16_t*);
virtual uint8_t getIdentity(uint8_t *buffer,uint16_t
maxBufferSize, uint16_t *lastBufferSize );
```

IController identifies the Sub Controller, ISensor identifies the Sub Sensor interface. The interface requirements are kept very limited for the sake of modularity and simplicity also sake of the resources.

IController:getStatereport: Every controller can have several states. For example; on, off or high, medium, low. These states are defined at the identity JSON which is provided via “getIdentity” method. This method is responsible for providing the current state of the Sub Controller to Device Node in JSON format thus Device Node can send the collective status information to Main Box. Some of Sub Controllers do not have any state, they return empty JSONs.

IController:acceptCommand: Device Node does not have the responsibility of the execution of the Sub Controller commands. Sub Controllers have this responsibility. Again every controller can have several executable commands. For example a buzzer can have beep() and alarm() commands or a light can flash with different patterns. These are actions, they can’t be described as states. Current implementation defines state changing commands’ with zero command id. Other commands come after this id. Every command is defined in identity JSON like states of the controller.

Every Sub Controller can execute its commands with command identifier. State changing commands also need state ids.

Sub Controller’s command accepting and executing responsibility decreases the burden of Device Node. Device Node only has to direct the command ids and state ids to correct Sub Controller. Device Node does not know and does not care the meaning of the command, only routes it.

IController:getIdentity: Sub Controllers have commands and states. In the other hand Device Node does not care about any of this information, Device Node is responsible for transportation of this information to Main Box, only. Every Sub Controller identity is carried to the Main Box with a wrapper message which has a stamp of the Device Node.

```

controllers: [{deviceType: String,
               localId: Number,
               name: String,
               states:[{n:String,i:Number}] ,
               commands:[{id:Number,name: String}]]}]

```

JSON template of the Sub Controller represented above.

Every Sub Controller identity JSON hardcoded in its class. At the current implementation with every “getIdentity” call it fills the pointed area with its identity JSON.

ISensor: getReport: Sensors have different specifications, they can have analog outputs or digital outputs, they can have very different measurement units. Device Node can’t afford this complexity. Every sensor separated from Device Node. When the Device Node requests a measurement “ISensor” interface gives a measurement JSON to Device Node. Device Node delivers this measurement JSON to Main Box. This is the only interaction of the Device Node with sensors while runtime. Main Box evaluates the coming sensor JSON with sensor’s pre arrived identity information.

ISensor: getIdentity: Sensors’ features, measurement units and variable types defined in its identity JSON. Device Node delivers this information to Main Box. By this way upper layers can understand the incoming data from sensor.

```

sensors: [{deviceType: String,
            localId: Number,
            name: String,
            varType: String,
            unit:String }]

```

JSON template of the Sub Sensor, represented above.

With this simple and low cost interface architecture, system gained great modularity and flexibility. All of the details extracted from Device Node, also developer can create his/her own sub devices without interfering the Device Node infrastructure.

Sub Controllers and Sub Sensors have a common feature at current implementation, they are coded in C++ and they are running on Arduino Uno. Current code has been limited by the memory limit of the Arduino Uno's microcontroller unit. Arduino has three types of memory, they are Flash, SRAM and EEPROM. Flash occupied by the program, SRAM used by the decelerated variables and fields, EEPROM used for long term records.

Current design aims minimum SRAM usage, because SRAM is the most critical one. Because of this all of the static strings and arrays, for example all of the static sub device JSON declarations embedded in the SRAM with PSTR, Arduino's program space macro. This capability makes possible to use large static fields with such a limited SRAM. Only one field which can carry one large field at a time can maintain several fields of like this. By this way current implementation can contain several Sub Controllers and Sub Sensors at the same time. Again Flash has a limit, and exaggerated usage of memory depletes even the Flash, also extreme usage of memory makes microcontroller unstable, memory leaks becomes possible. Arduino has a weak resource management system, it does not have an operating system, and especially near limit memory usage creates memory leaks. Current implementation sensitive to this kind of resource problems, but with a fully operational platform like a Linux board or a complete kernel this kind resource problems becomes avoidable.

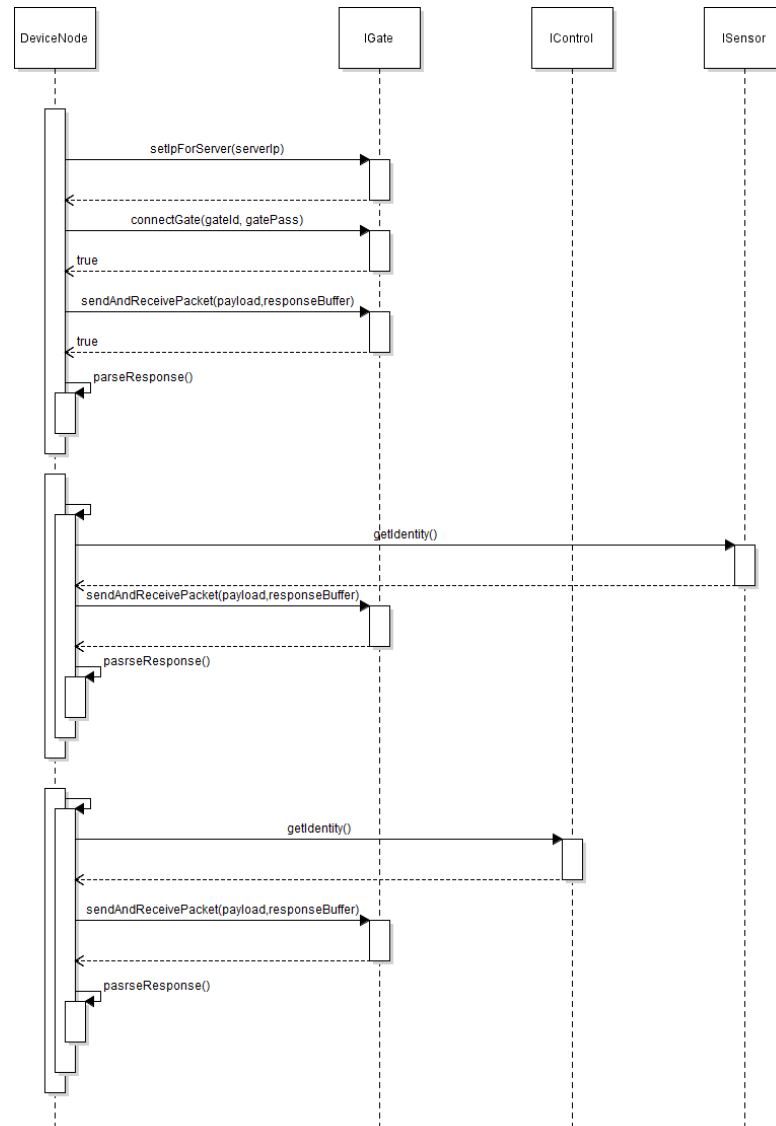


Figure 5.15 Initialization process

Initialization process of the Device Node affected from limited resources. Booting process divided, at every initialization message Device Node sends a sub device's information. The message can contain a sensor's JSON or a controller's JSON. Main Box responsible from the reassembly of the complete Device Node structure. After this initialization process the Main Box can understand the sensor feedback, controller states and can inform the API. Without proper initialization Device Node have no use in the system.

Initialization process is critical, Device Node has to introduce itself perfectly. One missing initialization packet leads to an unacceptable state. To provide this mechanism every reached Device Node message responded by the Main Box. Acknowledgement response guarantees the message receiving. With this positive feedback Device Node can continue to register its sub devices. After the registration Device Node can switch the regular functioning cycle. It can send sensory data and states and it can accept the controller commands. Device Node sends initialization data after every boot, this is a detail of the current implementation, by this way Device Node can function with a simple algorithm and simple mechanism.

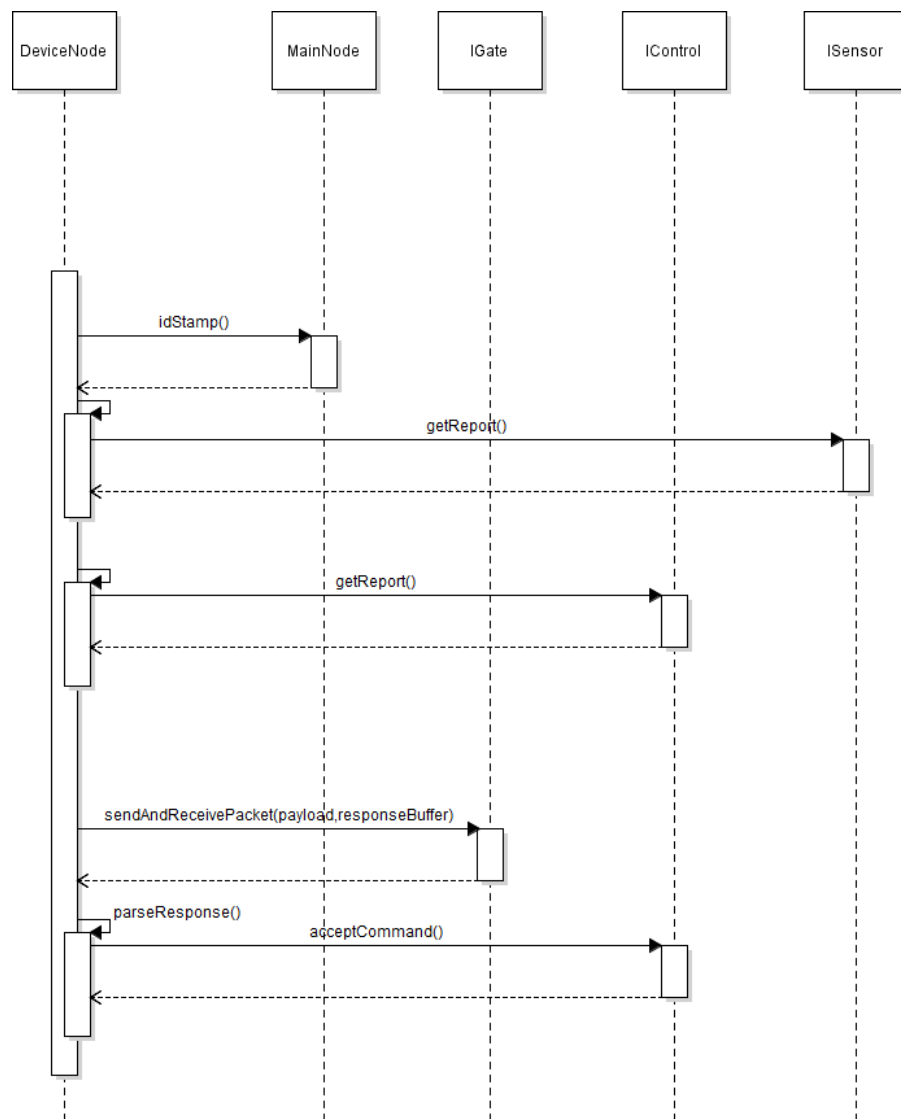


Figure 5.16 Regular working cycle

Communication: Device Node's current implementation uses http connection via ESP8266 version one. ESP8266 is a low cost Wi-Fi chip. It uses basic AT commands. By this commands the chip can connect Wi-Fi and send http messages to given IP and port. Also it can run on access point mode. Some better versions of the ESP8266 can run Lua scripts. Running a basic web server is possible with these models. Current hardware implementation uses the earliest model of the ESP8266. So, advanced client modes are not supported by used model of the ESP8266. Because of this Arduino Uno implements a client for ESP8266. This client class is responsible from stable http messaging. But, again Device Node does not care and know the details of the ESP8266 class. It sees ESP8266 as a communication channel.

ESP8266 connection class implements the IGate interface. Every detail of the ESP8266 AT command set is hidden under its own class. Device Node only send packets and receives responses from this interface provider. Also Device Node has to provide necessary connection information to ESP8266 like Wi-Fi name and password, server IP. The object oriented approach and interface usage creates a very modular design. But real word have implementation difficulties.

Arduino Uno has only one real UART channel. Programmer can use this channel for debugging. Also Arduino IDE uses the same channel for program loading. This channel is the only real UART channel which has a dedicated hardware, Arduino can provide software serial from other pins but they do not have a corresponding hardware on the Arduino Uno board. Different models of the Arduino platform have different resources like Arduino Mega, it has three UART channel. ESP8266 uses the same UART interface for communication. At this stage a serious problem occurs. Same channel used for debugging, program loading and ESP8266 communication. This is not possible, UART is a point-to-point connection. To resolve this problem ESP8266 channel shifted to a software serial channel.

Hypothesis was “A separated ESP8266 communication channel can solve the problem”. But while testing a major problem occurred again. Software serial couldn't read long response strings from ESP8266, for example available Wi-Fi list of the ESP8266. So

debug channel shifted to serial channel. But program loading remained as a problem. ESP8266 disconnected from Arduino at every program loading, after loading reconnected. Because, ESP8266 interferes the communication media while program loading, generates some noise strings. These noise disturbs the program loading process. But also eavesdropping the ESP8266 channel with a USB-TTL is possible with the exact same technique, this technique provides traffic monitoring facility for debug. Programmer can monitor the flowing messages between ESP8266 and Arduino.

Communication problem is created by the current hardware and resource limitations. Also ESP8266 client class on the Arduino consumes great resources. Specially providing stable communication and timing for the ESP8266, also embedded static http headers and strings consume resources. For further development, better communication hardware is recommended.

Device Node uses IGate interface for Main Box communication.

IGate:

```
virtual uint8_t connectGate(uint8_t* id, uint16_t idLen,
uint8_t* pass, uint16_t passLen);
virtual uint8_t sendAndReceivePacket (uint8_t *payload ,
uint16_t payloadSize, uint16_t *lastBufferSize, uint8_t
*buffer, uint16_t maxBufferSize);
virtual void disconnectGate();
virtual void setIpForServer(uint8_t *serverIp , uint16_t
serverIpLength);
```

IGate:connectGate: Device Node does not know the details of the communication channel. For example current implementation uses http POST messages, this is a hidden detail. Node Device provides necessary connection information to communication class like Wi-Fi name and password. Connect gate is used for network access, simply Wi-Fi connection.

IGate:setIpForServer: This interface method added after some tests. Current implementation uses local Wi-Fi networks. And dynamic server IP becomes a need under

this circumstances. System should adapt every network. Also current implementation stores the server IP in EEPROM, EEPROM access provided via interface layers even the long term data storing has no footprint on Node Device implementation. By this way Device Node can connect the Main Box after every reset.

IGate:sendAndReceivePacket: Every communication sequence, Device Node sends a collected sensor and state packet in JSON format and receives a response. This response has a specific format at current implementation.

Example sensor and state feedback JSON :

```
{  "oId":0, "oRs":0,
    "mainId":"1021",
    "mssgTyp":"reg",
    "Data":[{"s":[{"localId":1, "val":44}]}],
          {"c":[{"conId":1, "st":1},
               {"conId":2, "st":null}]}
    ]
}
```

Every state and sensor measurement sent with its unique id. The Main Box can map the incoming feedback with this ids. Sending the unit types and other details at every iteration is meaningless, also creates overhead over communication. For example {"localId":1,"val":44} does not have any meaning alone but Main Box can understand this JSON part, Main Box has the complete identity of the sensor. Same mechanism applied on the Sub Controllers' states. Main Box can understand the current situation of the controller.

Example Main Box response string :

```
"#RECEIVED{1,1021,1,2}$"
```

Every response has to contain a specific mark, otherwise ESP8266 class thinks it lost the server connection. RECEIVED part of the string provides this facility. Also Arduino and ESP8266 communicates over a very simple and unreliable hardware channel. '#' and '\$' symbols are the start and stop marks of the server response. With this marks Arduino Uno can detect the beginning and the ending the server response. {1,1021,1,2} part is the command part of the response. Current implementation uses one sided communication

start. Device Node can reach the Main Box, Main Box can't reach the ESP8266. Because running the ESP8266 at Access Point mode is not secure. Also this usage type of ESP8266 requires great deal of processing power. Also maintaining such a server on a MCU with the other Device Node responsibilities is impossible.

Current implementation and design uses a synchronous communication scenario. Device Node connects the Main box and transmits the collected data and takes the response. Response can contain a command. At the next iteration Device Node sends the last command id. This id provides a message tracking facility to the Main Box. Also message contains an execution result. Every Sub Controller returns a positive response to an executable command. By this way we can understand the command execution succeeded at the very ground level. The response returns the very executor of the command.

"oId":0,"oRs":0 part is the response and command id part of the Device Node feedback. "oId" means old command id, "oRs" means old response. JSON field names are very short because of the memory limitations.

{1,1021,1,2} example string identifies a command execution request. First parameter defines the command id, second part defines the Device Node id. Device Node checks this part to ensure response is sent to itself. Third part defines the Sub Controller id. Device Node uses this parameter to pass remaining part of the command to Sub Controller. Sub Controller executes the command via given id. If the requested command id is available id interface returns a positive response to Device Node.

Parameter Passing Problem: This approach assumes every command and state called by its unique id. Still executing a complex command on Device Node is possible. Command string is not limited by the design. A command execution string can continue after the command id and it can contain parameters. But this usage violates one basic rule of the Sub Controller identity. Fundamental idea assumes all of the Sub Controllers have to contain all of the executable commands as JSON. If we chose to use parameter containing executions this creates a massive problem. The Main Box can't define the complete command string. This type of commands needs external mechanism for

parameter definition. Actually this is a simple problem, someone has to tell complete versions of the commands to Main Box, or dynamic creation of the complete commands is an option. But still this approach violates the nature of the system characteristic.

Existing object oriented design contains some support classes. Long term data storing procedure, I/O operations, serial communication and similar platform dependent operations separated from the Device Node infrastructure. Simply Device Node does not use any function of the Arduino related libraries'. For example a standard operating system provides a proper file system, but Arduino is a microcontroller. So EEPROM usage should not concern the Device Node, IO operations have a similar situation. Because of this reason support classes added the design.

Hardware Design of Device Node: All hardware architecture based on Arduino and Arduino compatible environment modules. Current implementation has screw sockets, pluggable cables and power providing boards. Aim of the current hardware is to make all system mobile and representable. Design of the current implementation has no further purpose.

Some of the available sensor and control equipment has high current need. Arduino Uno can't provide such high current needs. For example ESP8266 can need up to 320 mA, Arduino Uno can afford only 200 mA. Because of this power problem external breadboard power supply used for every Node Device hardware set.

5.2. Database Design

The system uses a NoSQL database. Consequently, a relational diagram doesn't fit perfectly. But an E-R Diagram is provided (as shown **Figure 5.17**) in order to explain database entities.

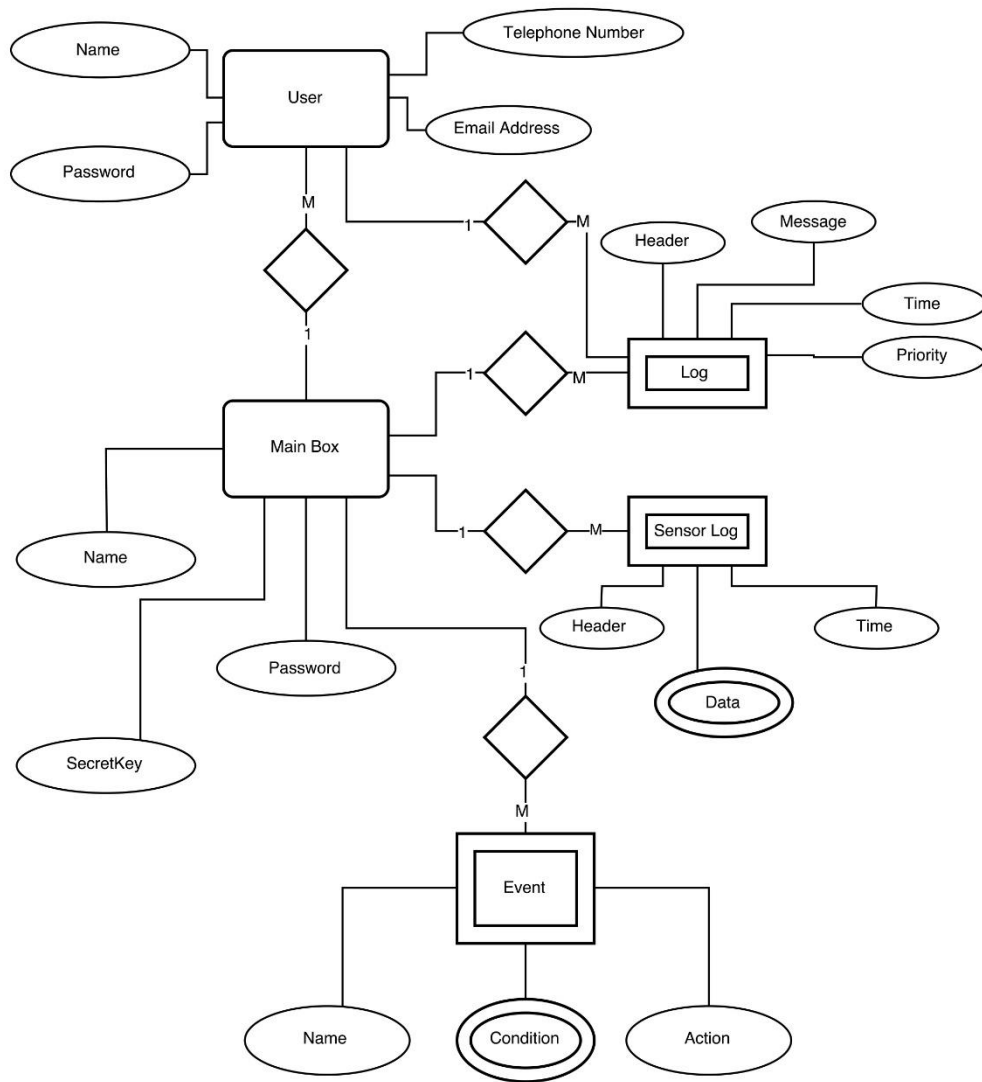


Figure 5.17 E-R diagram

6. APPLICATION ANALYSIS

In this section web interface pages are shown.

6.1. User

User may login (see Figure 6.1), sign up (see Figure 6.2), and change his/her password (see Figure 6.3) via user pages.

The login page features a dark blue header with the word "Login" in white. Below the header, there are two white input fields: one for "Email" and one for "Password". At the bottom of the form area, there are two buttons: "Login" and "Register". The background of the page is a scenic image of snow-capped mountains under a clear sky.

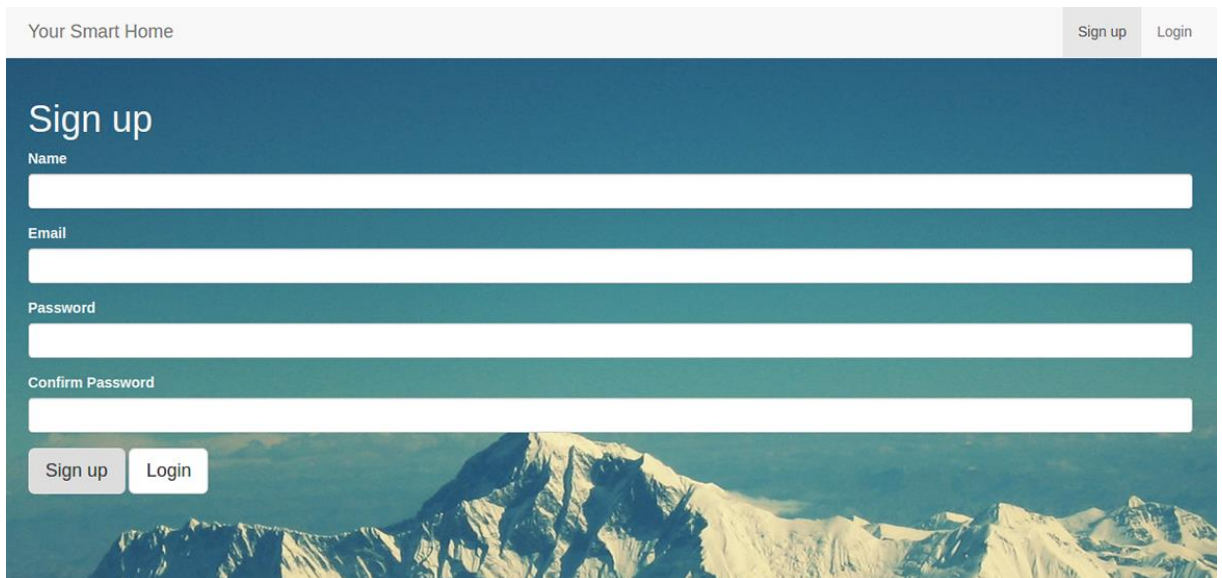
Login

Email

Password

Login Register

Figure 6.1 Login page

The signup page has a light gray header with the text "Your Smart Home" on the left and "Sign up" and "Login" buttons on the right. The main content area has a dark blue background with the word "Sign up" in white. Below this, there are four white input fields for "Name", "Email", "Password", and "Confirm Password". At the bottom, there are two buttons: "Sign up" and "Login". The background image is the same mountain scene as in Figure 6.1.

Your Smart Home

Sign up Login

Sign up

Name

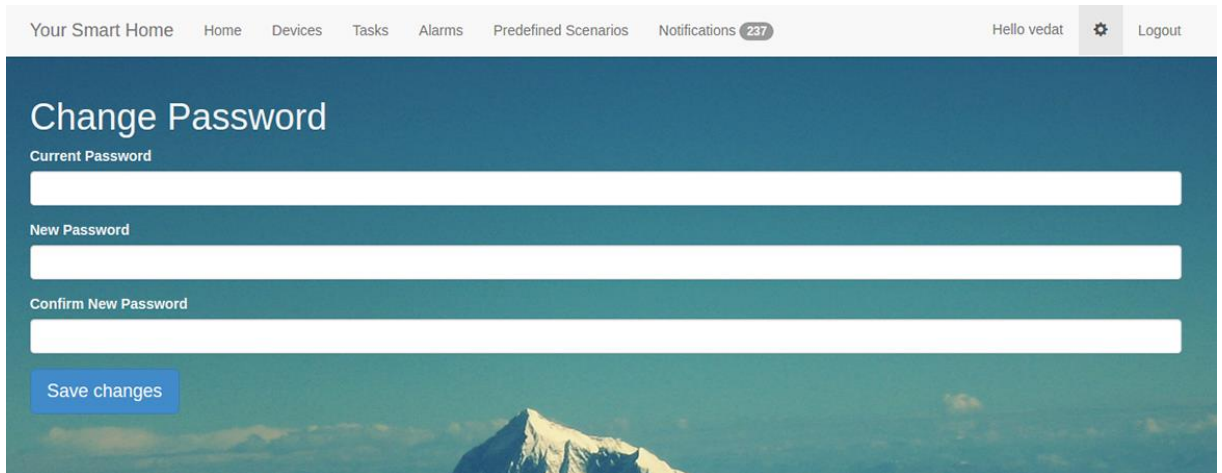
Email

Password

Confirm Password

Sign up Login

Figure 6.2 Signup page



The screenshot shows the 'Change Password' page of a smart home dashboard. The page has a dark blue header with a navigation menu: 'Your Smart Home', 'Home', 'Devices', 'Tasks', 'Alarms', 'Predefined Scenarios', 'Notifications' (with a red badge showing '237'), 'Hello vedat', a settings gear icon, and 'Logout'. The main content area has a background image of a snowy mountain peak. It contains three white input fields labeled 'Current Password', 'New Password', and 'Confirm New Password'. Below the fields is a blue button labeled 'Save changes'.

Figure 6.3 Change password page

6.2. Home & Notifications

Sensors of all devices that registered to home are shown in home page (see Figure 6.4). Device connectivity and alarm notifications are shown in Notifications page (see Figure 6.5).

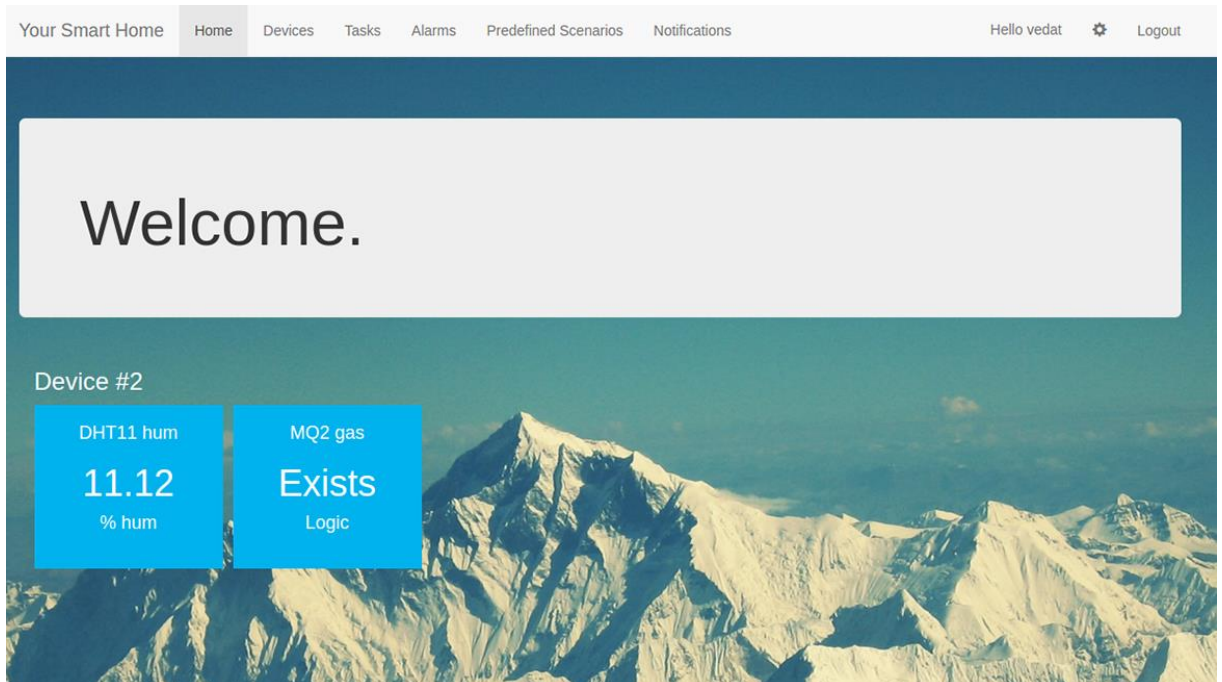


Figure 6.4 Home page

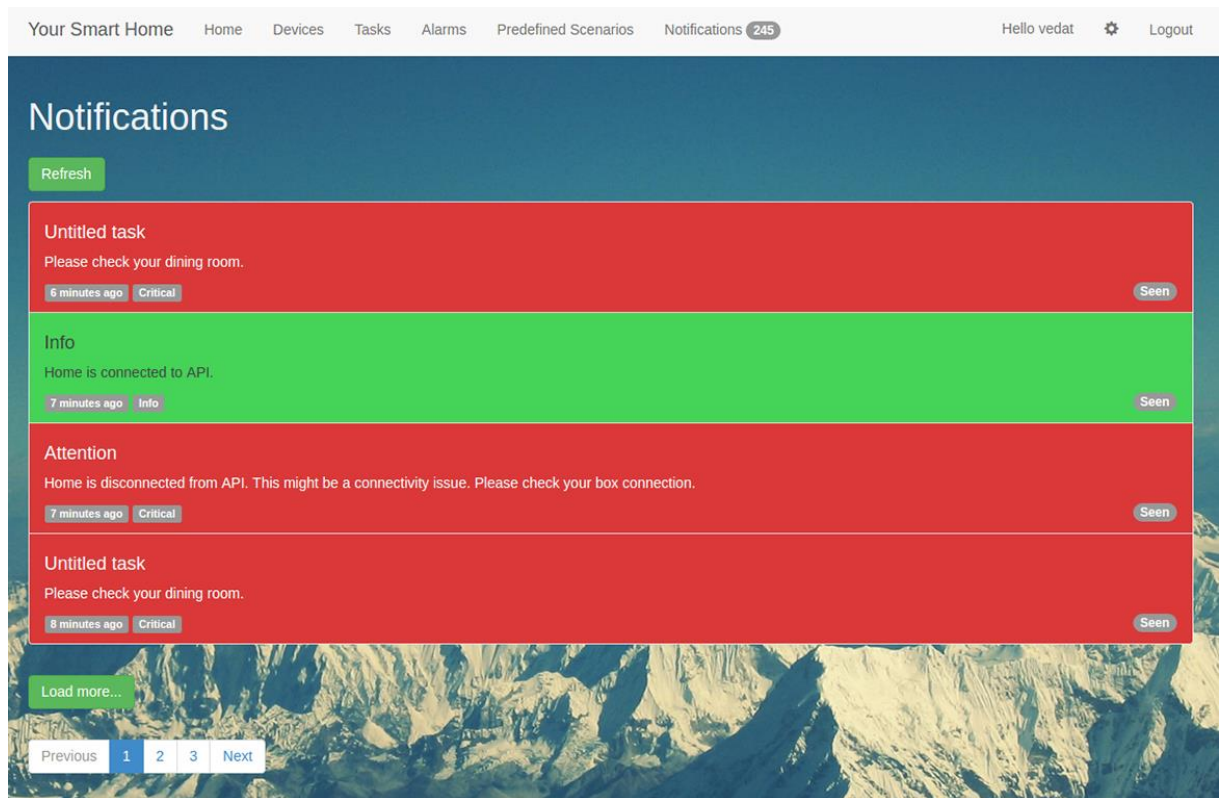


Figure 6.5 Notifications

6.3. Devices

Users may manage their registered devices from “Devices” pages.

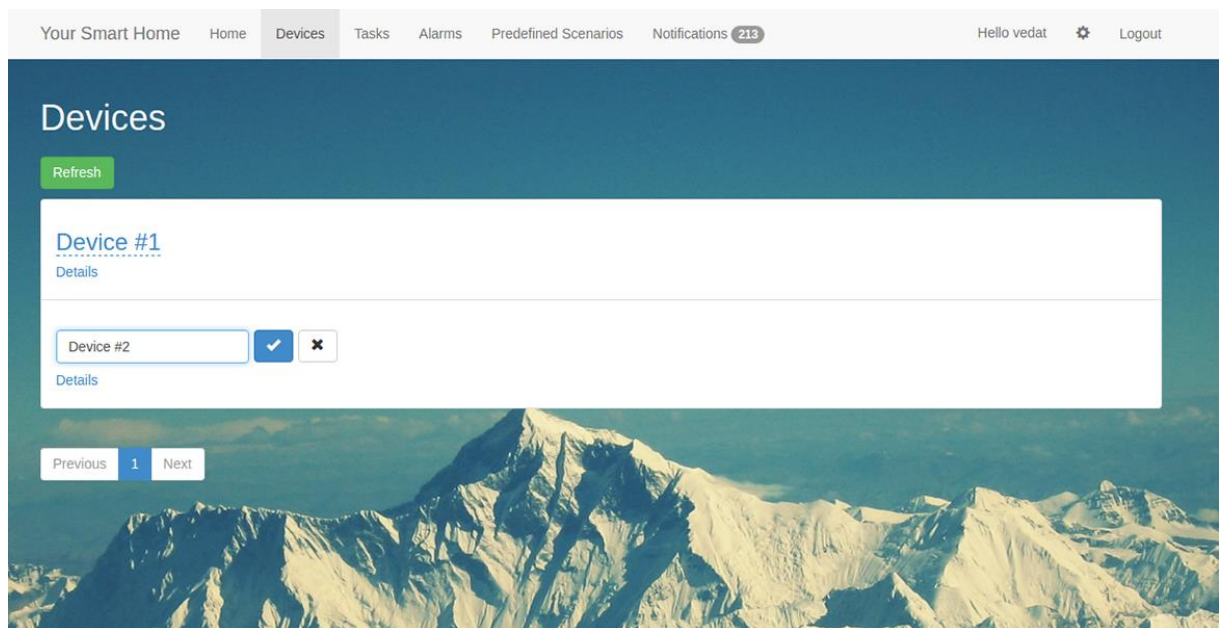


Figure 6.6 Devices main page

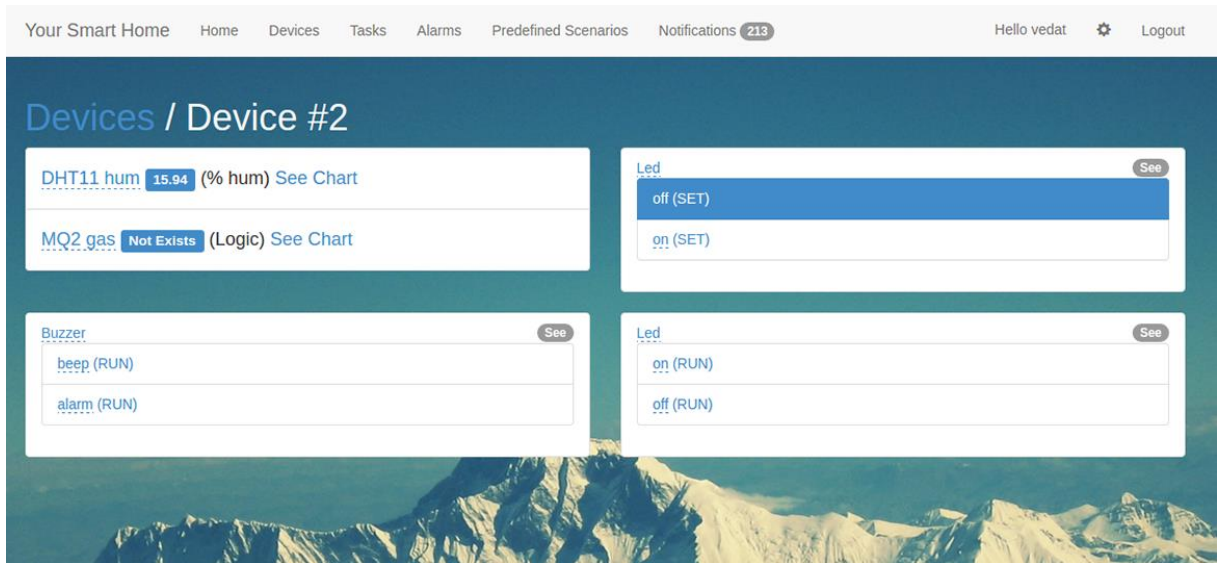


Figure 6.7 Device detail page

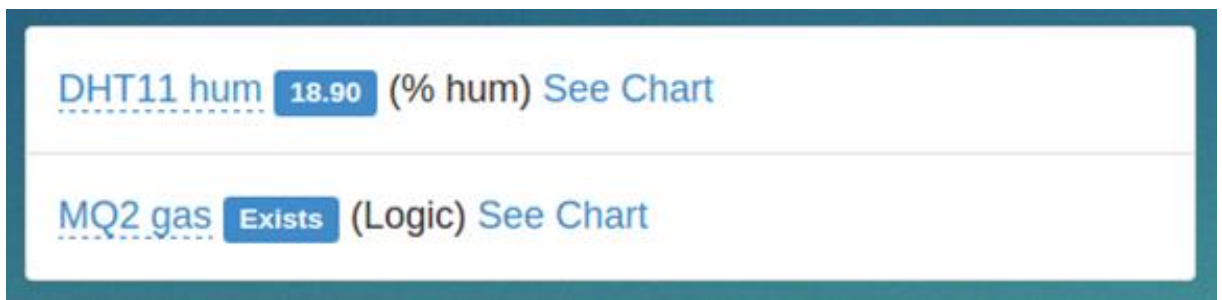


Figure 6.8 Sensors of device

Device list is provided by “Devices” page (see Figure 6.6). Every device has a detail page also (see Figure 6.7). Device detail page sections may be different for different devices. Because, some devices contain sensors, and some devices only provide control interface. For example, in a detail page of a device that contains sensors, sensor status is shown in list form. Last value that read from sensor is written in blue label. Unit of the sensor comes after that label. If sensor provides a chart, user may see it by clicking on “See chart” link (see Figure 6.8).

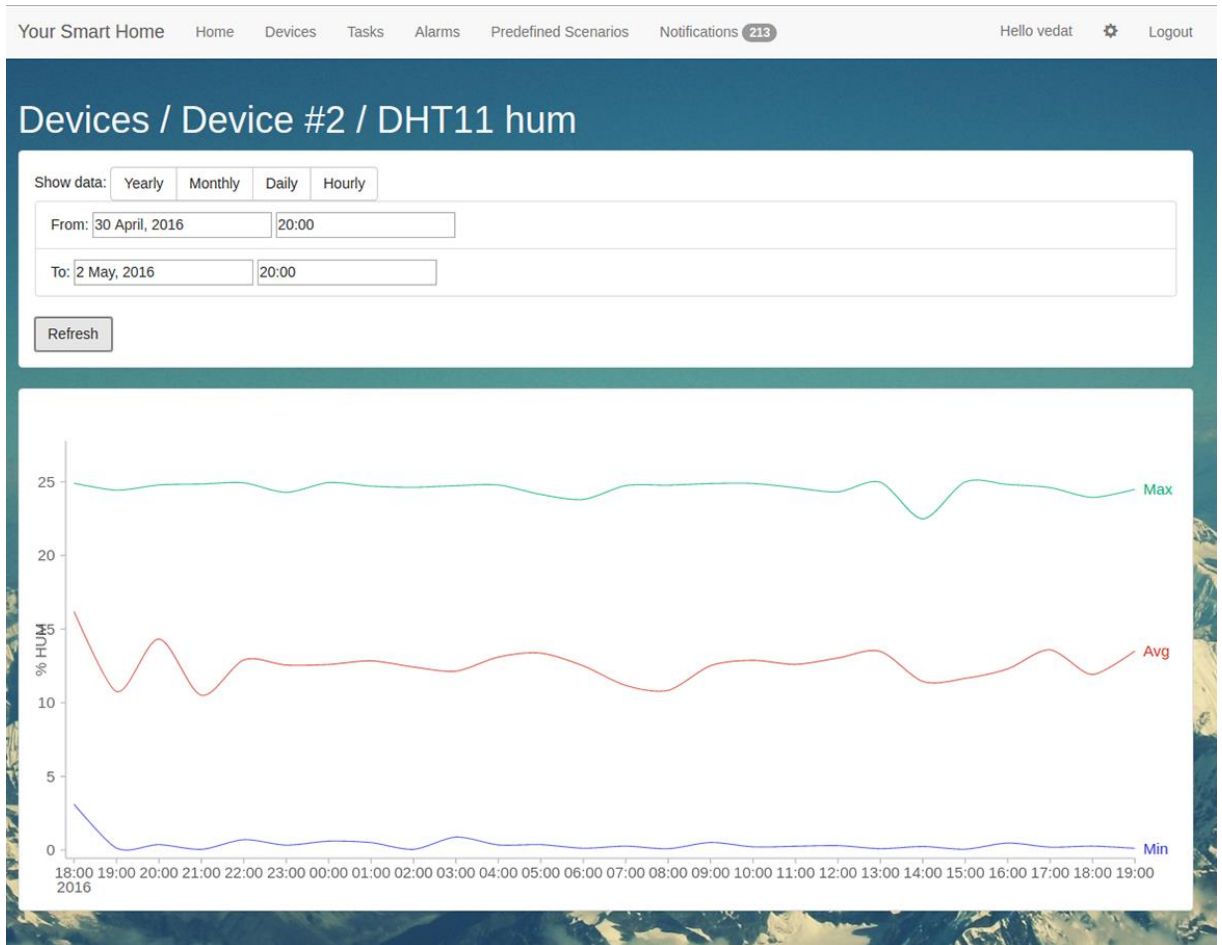


Figure 6.9 Chart page of sensor (continuous data)

There are two types of sensor charts: continuous chart and discrete chart. Continuous chart is used for visualize values that comes from sensors like temperature, humidity (see Figure 6.9) etc. But this chart is not appropriate for sensors that generates discrete values. Because of this reason, system visualizes discrete data with discrete chart (see Figure 6.10). Main difference of discrete chart is regions of value levels are very clear.

The other list type in device detail page is “properties”. For example there is “Led” list that contains “off (SET)” and “on (SET)” buttons in Figure 6.11. Functionality of a button is changing property/state of sub-device. In short, user may set the current state of a sub-device by clicking the state.

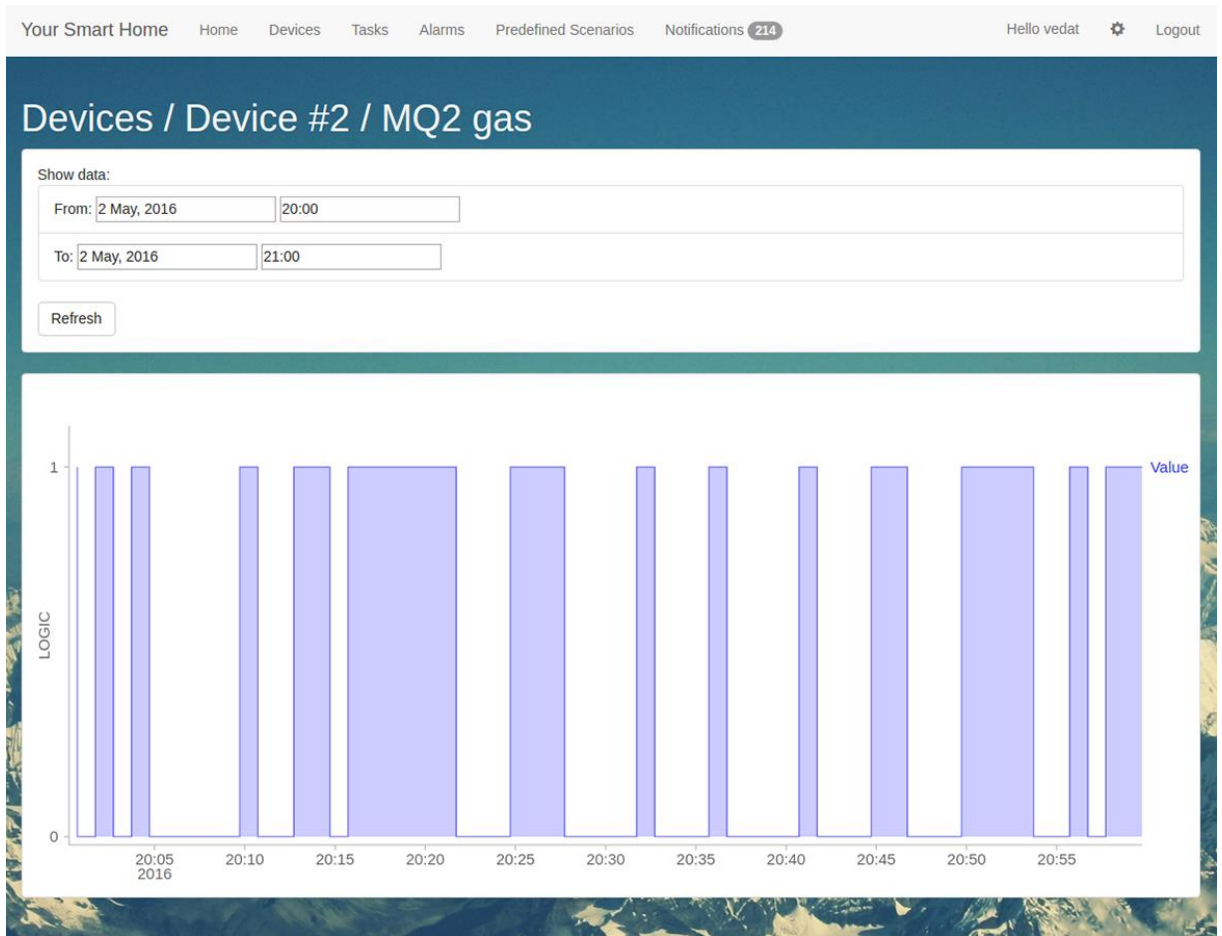


Figure 6.10 Chart page of sensor (discrete data)

Last list type in device detail page is “functions”. For example there is “Buzzer” list that contains “beep (RUN)” and “alarm (RUN)” buttons in Figure 6.11. Functionality of a button is calling a function of sub-device.

Order-response sequence is shown in Figure 6.11, Figure 6.12 and Figure 6.13. “Pending” label means “API sent your message to the sub-device, and waiting for the response of it.” status. The blue button in property list, shows the current state of the sub-device. In contrast, there is no “current” or “pending” state for function calls.

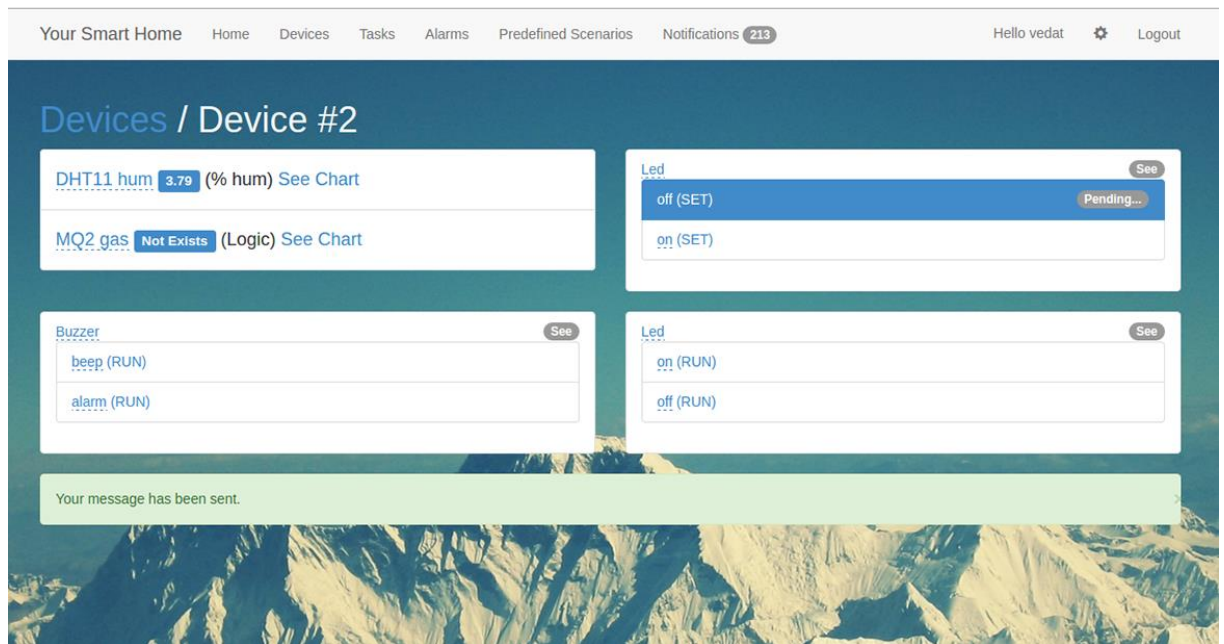


Figure 6.11 Alert message, after clicking “LED: SET off”

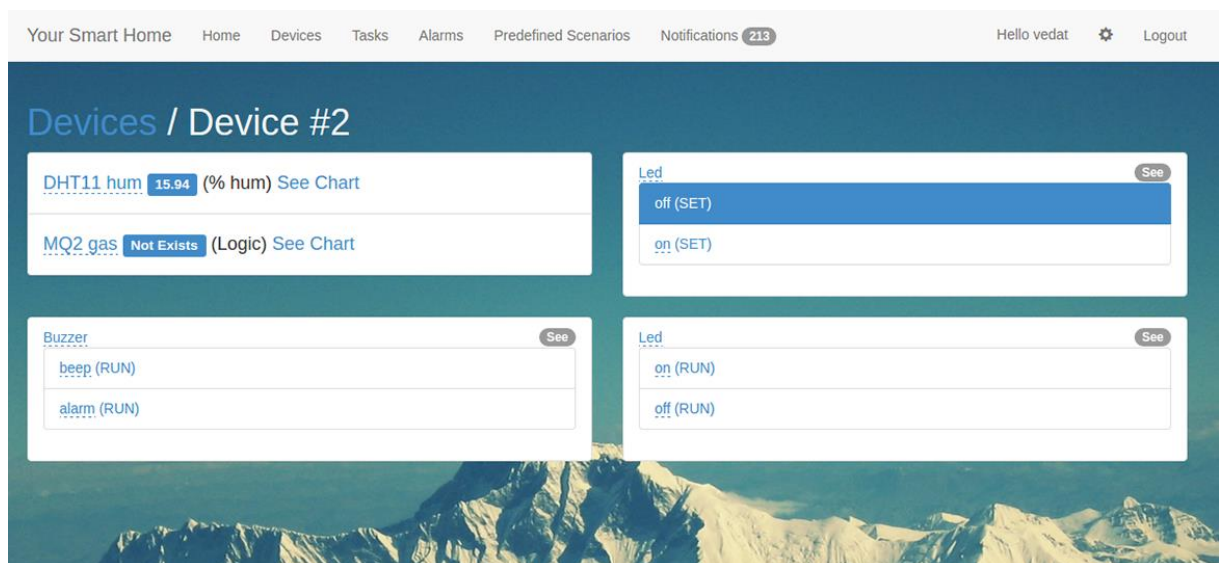


Figure 6.12 “Pending...” label of “LED: off (SET)” disappeared

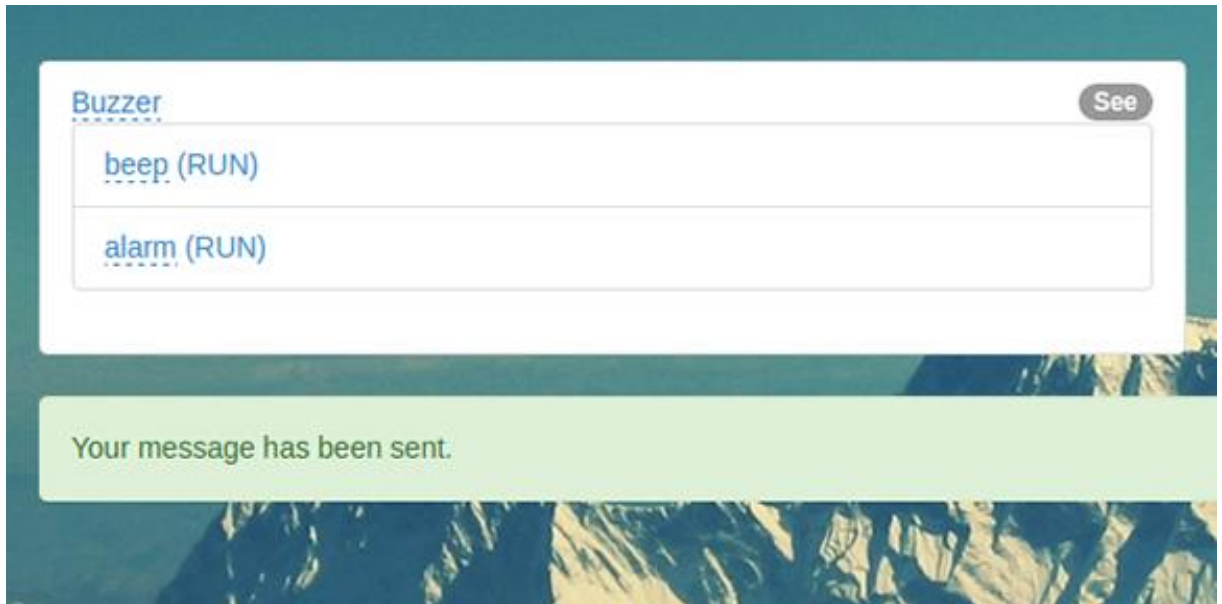


Figure 6.13 Alert message, after clicking “alarm (RUN)”

6.4. Task System

Task system pages and functionalities are shown in this section.

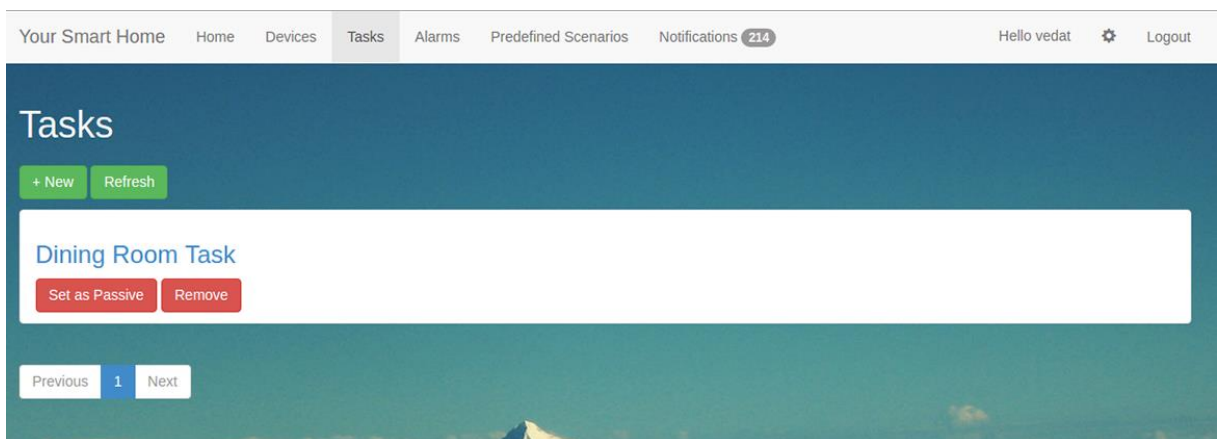


Figure 6.14 Tasks

Your Smart Home Home Devices Tasks Alarms Predefined Scenarios Notifications **214** Hello vedat Logout

New Task

1. Info & Timing 2. Filters 3. Actions 4. Finish!

Name

Profile name helps you to remember profiles without checking their details.

Time range

☒ Monday
 ☒ Tuesday
 ☒ Wednesday
 ☒ Thursday
 ☒ Friday
 ☒ Saturday
 ☒ Sunday

Start: :

End: :

(e.g. If this conditions happens between 00:00 and 23:59 in selected days, then do something.)

Is active? ☒ Yes ☐ No

If you choose "Yes", then this condition will be checked. Otherwise, it will be stay disabled until you set as "Yes".

[Next >](#)

Figure 6.15 New task creation

Your Smart Home Home Devices Tasks Alarms Predefined Scenarios Notifications **214** Hello vedat Logout

New Task

1. Info & Timing 2. Filters 3. Actions 4. Finish!

Wait time

This option allows you to set wait time for filters. (0 = disabled)
(e.g. If this conditions happens along 4 hours at least, then do something.)

Find Filter

Selected filters

- Device #1
- Relay
- Device #2
- DHT11 hum**
- MQ2 gas
- Led

Figure 6.16 New task creation step #2

Figure 6.17 New task creation step #2

User may create a task by using task creation interface. In first page (see Figure 6.15) time constraints are arranged by user. In second page wait time and filters are selected (see Figure 6.16 and Figure 6.17). In third page user may select the actions for task. Email address(es), level of alarm and list of actions that allowed by system are parts of this page (see Figure 6.18 and Figure 6.19). In fourth page, system shows the configuration of the task (see Figure 6.20).

When a task produce an alarm, this alarm is shown in Alarms page. User may set alarm as silenced (see Figure 6.21, Figure 6.22 and Figure 6.23).

Your Smart Home Home Devices Tasks Alarms Predefined Scenarios Notifications **214** Hello vedat Logout

New Task

1. Info & Timing 2. Filters 3. Actions 4. Finish!

Alarm Level

☒ Notification

☐ Warning

☐ Danger

Send Message

To

Email address(es)

Message

Send message every

minute(s)

Find Action

Selected actions

- Device #1
- Relay
- Relay
- IRControl
- Device #2
- Led
- Buzzer

Figure 6.18 New task creation step #3

every

minute(s)

Find Action

Selected actions

Set Device #2:Buzzer as

Set Device #1:Relay as

[< Back](#) [Next >](#)

Figure 6.19 New task creation step #3

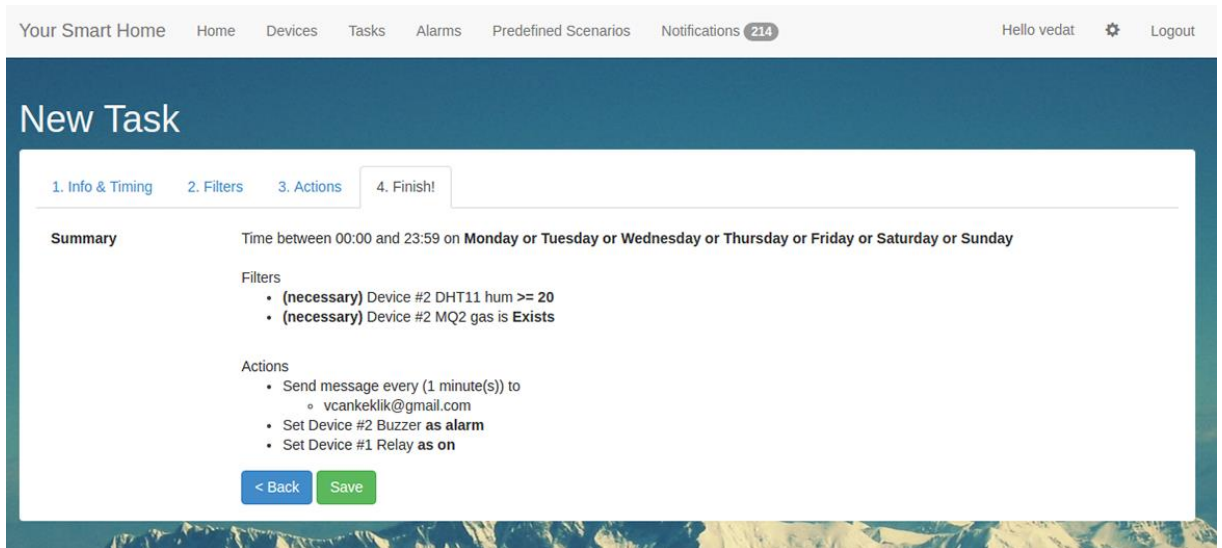


Figure 6.20 New task creation final step

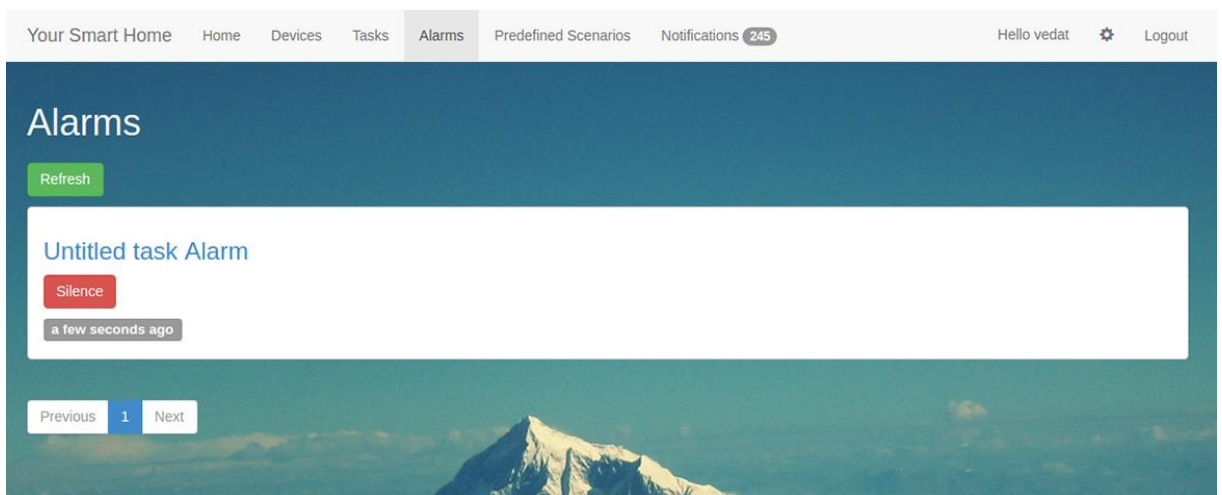


Figure 6.21 Alarm was created by “Untitled Task”

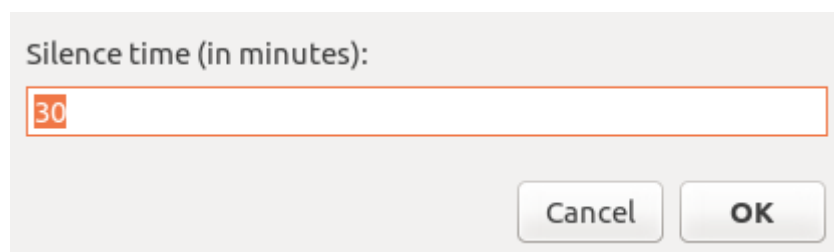


Figure 6.22 Setting alarm's silence time

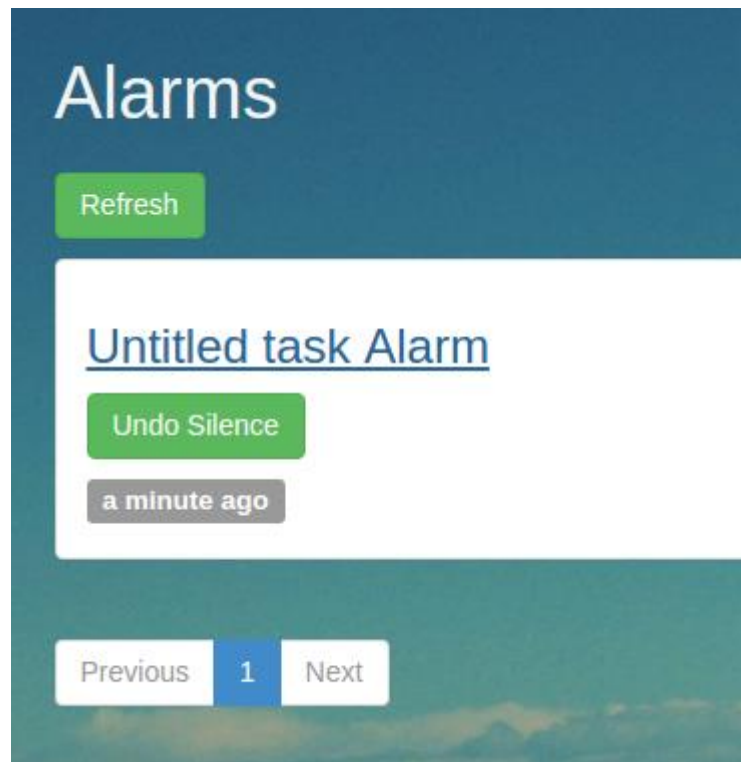


Figure 6.23 After time setting operation

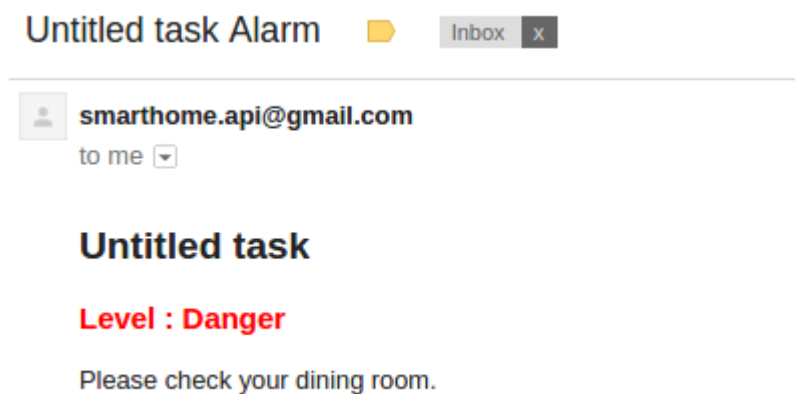


Figure 6.24 Mail alert that comes from alarm

7. EXPERIMENTAL RESULTS

The system structure is multi-layered. And these layers need to be tested separately. In this section, fault tolerance level of system and behavior of system on edge cases are discussed.

7.1. Failure Tests

Because of the MQTT architecture, every client must be connected to the broker anytime. Otherwise, there will be message loses. Therefore, every disconnection, or other network issues should be handled carefully.

7.1.1. Web API Failures

Web API is a client from MQTT Broker's side. So it needs to be connected all the time like other MQTT clients. Web API only listens home/+out channels. There is no tight connection between Web API and main box. And this makes the main box more flexible. When API is down or disconnected, main box keeps running independently. It checks tasks, and keeps open its MQTT connection. This is a plus for fault tolerance. In this case, sensor data won't be saved to Web API database. In real-life Web API doesn't disconnect frequently. If it does, this is a bigger issue than saving sensor data. Because, this means all control interface is collapsed or unreachable.

In any case of failure, system save a log for the error and keep running.

7.1.2. Main Box Failures

Main Box is the local connection component of the system. Failure possibility of Main Box is higher than Web API. Because network may be less reliable in user context than Web API side. Also misuse of device by users is also a danger.

In any case of failure, main box keeps running with logging internally. So far so good, but what happens if a hardware failure happens? There is two important tasks: inform the user, and store user commands while main box is unreachable. These tasks are responsibility of MQTT Broker software. If a main box disconnects, then MQTT Broker informs the Web API. And user may see this event as a notification. In short, system informs the user when Main Box is down. The second task (storage of user commands) is also done by MQTT Broker. If user send a command while Main Box is disconnected,

MQTT Broker saves the message to its database. When Main Box connected to the broker, gets those messages.

7.1.3. Device Failures

Devices are lightweight. In case of software failure or malfunction, they restart itself. Hence, they are fault-tolerant for software level errors. But hardware-level issues (power loss) need user intervention.

7.1.4. MQTT Broker Failure

MQTT Broker is the main communication component in the system. It must be active all the time. In MQTT architecture, clients attempt to connect periodically if they are disconnected. This means the broker will restart and clients connect again.

7.2. Complete Test

Complete test of the system took 40 minutes. Startup took about 5 minutes, excluded. System device manager, has sent commands to devices with an acceptable delay (avg. 500ms).

System task manager have worked successfully. Result of task manager test is time quantum of task manager shouldn't be too short or too long. In fact it should be set according to the quality of device system. If sensor devices are slow then making task manager fast will be meaningless.

Sensor system has worked without failure generally. Main reason of those minor errors are sensor hardware malfunction.

8. PERFORMANCE ANALYSIS

System performance is discussed in this section.

8.1. Main Box – Device Performance

The system has a layered architecture. Lower layers were implemented with C++ and main box software was implemented with JavaScript. The main goal of the system is to create a modular architecture. For example the sensor node's code should be able to run on a single board computer, with proper base library implementations.

Changeable parts and interface classes resulted in a very adaptive system. In this case system is dependent on the resources of the base materials. Microcontroller's memory size, communication devices' limits and latency and Wi-Fi network condition have effect on the system. Even a sensor module has effect on the system, time cost may change the period of sensor feedback cycle.

Implemented system uses ESP8266 Wi-Fi module, user sends the AT commands and HTTP packet via UART. Current communication hardware can support 0.75 HTTP packet per second, but packet size limited by the microcontroller's memory. Maximum size is 200 bytes. But microcontroller's sensor and controller configuration affects this size, with free memory developer can expand the communication packet limit.

Digital input and output number, analog input and output number, SRAM size, Flash size and UART number are the most critical resources of the lower layer (see Table 8.1). Sensor and control communication established via these I/O pins. Consequently capacity of a single node depends on the selected hardware. User may use an Arduino board or a Raspberry. Every base hardware has a capacity, higher pin number and memory size means more sensor and control equipment. Memory cost of 3 sensors and 1 controller class is 23268 bytes of FLASH, 1367 bytes of SRAM. These values depended on the inner structure of class. Average memory cost of a class is 5817 bytes of FLASH and 341 bytes of SRAM. With accepted memory cost values supportable class number becomes calculable. Limit values can guide to most optimal base hardware selection (see Table 8.2).

Table 8.1 Model table of Arduino boards

Name	CPU speed [MHz]	Analog In/Out	Digital IO/PWM	EEPROM [kB]	SRAM [kB]	Flash [kB]	UART
101	32	6/0	14/4	-	24	196	-
Due	84	12/2	54/12	-	96	512	4
LilyPad	8	6/0	14/6	0.512	1	16	-
Mega 2560	16	16/0	54/15	4	8	256	4
Micro	16	12/0	20/7	1	2.5	32	1
Nano	16	8/0	14/6	1	1-2	16-32	1
Pro	16-8	6/0	14/6	1	1-2	16-32	1
Uno	16	6/0	14/6	1	2	32	1
Yun	16-400	12/0	20/7	1	2.5 16MB	32 64MB	1
Zero	48	6/1	14/10	-	32	256	2

Table 8.2 Class number limits

Name	SRAM [kB]	Flash [kB]	SRAM class limit	FLASH class limit
101	24	196	72	34
Due	96	512	288	90
LilyPad	1	16	3	2
Mega 2560	8	256	24	45
Micro	2.5	32	7	4
Nano	1-2	16-32	3-6	2-4
Pro	1-2	16-32	3-6	2-4
Uno	2	32	6	4
Yun	2.5 16MB	32 64MB	7 7*1024	4 8*1024
Zero	32	256	96	32

8.2. Web API – MQTT Broker Performance

MQTT Broker provides high availability. It is linearly scalable because of its architecture. There is no direct connection between main box and APIs. They only listen each other's in/out channels. So if there is multiple MQTT Brokers, they will work separately without knowing any implementation or session details.

Web API has many more tasks to do. Sensor data storage is the most costly operation in the Web API. Because there are many sensors. Hundreds of sensors couldn't be tested physically. This would be expensive. Consequently, sensors were simulated with software.

In test, MQTT Broker and Web API server have worked together on a personal computer. And sensor data generator have generated data. Period is the time between sensor messages. For example in first row 3 sensors have sent their data in every 8 milliseconds (see Table 8.3). Reason of data duplication is document oriented database architecture. There are no strict transaction mechanism in MongoDB. Therefore, some records becomes duplicated. This test were ran with 8 GB memory. This is not enough for a web server and MQTT Broker. Also all parts of test have worked on same machine. This conditions should be considered for evaluation. It is obvious, in real environment, (real Web and MQTT servers, and real network) system may handle much more than this load.

Table 8.3 Sensor simulation results

Period [ms]	# of sensors	Data duplication	System status
8	3	Yes	Responsive
15	3	Yes	Responsive
50	3	No	Responsive
100	3	No	Responsive
100	100	Yes	Slow responses
200	100	Yes	Slow responses
200	1000	Yes	Unresponsive

9. CONCLUSION

Aim of the project is to create a smart home system with cloud link. Modularity and independency were the main goals of the architecture. Abstraction of the details of the lower levels, isolation from this complexity made this architecture possible.

At current implementation self-introducing nodes are implemented on Arduino Uno, with 2kB SRAM and 32 kB FLASH. Resource consumption primarily depends on the class design. In this case developer has to take some precautions. Large memory allocations, recursion, large storage memory usage and similar actions are undesirable. Working with an operating system is much easier and promising, threaded structure, resource management, proper file system, reliable network connection, multiple protocol selection are the benefits of an operating system. Raspberry Pi or another single-board computer can provide far greater resources than Arduino Uno. But project implemented with a cost concern and we have proved self-introducing nodes are possible with even a microcontroller. The project have reached its goals. But there are still some parts that need to be improved.

For further study we highly recommend more reliable communication modules for device implementation. ESP8266 has some advantages like low cost and generic control interface. But it is not very reliable for bigger implementations.

Using a main box for device management has many advantages. On the other hand, it makes the architecture more complex. Therefore, direct API access may be considered as an option for devices.

Task system was implemented as planned. For further study, its device check algorithm should be developed. For example, an interrupt based, asynchronous system may be more efficient instead of periodical scanning.

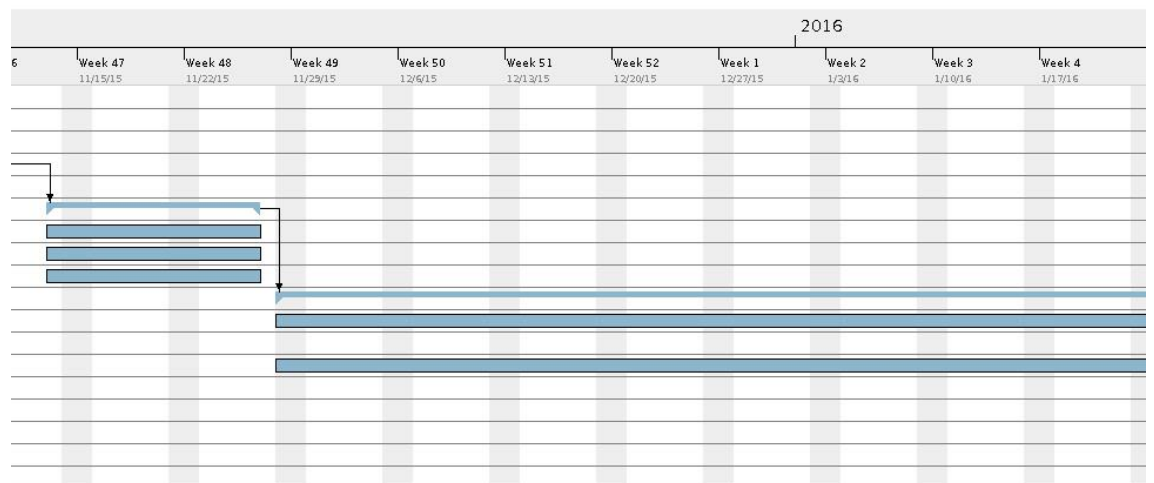
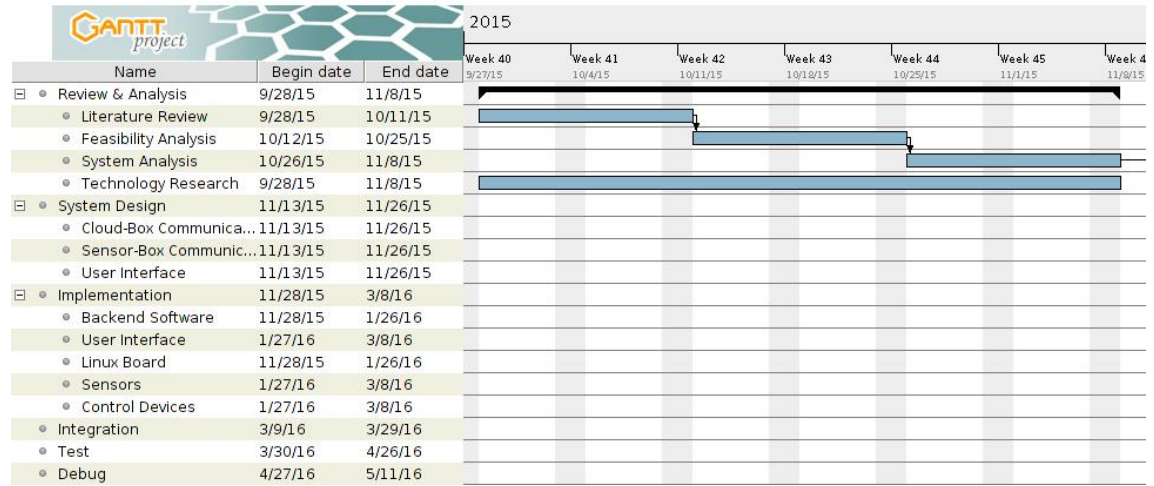
REFERENCES

- [1] Node.js Foundation, "About Node.js®," 2016. [Online]. Available: <https://nodejs.org/en/about/>. [Accessed 15 01 2016].
- [2] OASIS, "MQTT Version 3.1.1," 29 10 2014. [Online]. Available: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf>. [Accessed 15 01 2016].
- [3] HiveMQ, "MQTT Essentials Part 7: Persistent Session and Queuing Messages," [Online]. Available: <http://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages>. [Accessed 15 01 2016].
- [4] HiveMQ, "MQTT Security Fundamentals: Authentication with Username and Password," [Online]. Available: <http://www.hivemq.com/blog/mqtt-security-fundamentals-authentication-username-password>. [Accessed 15 01 2016].
- [5] M. Collina, "Mosca : Authentication & Authorization," GitHub, [Online]. Available: <https://github.com/mcollina/mosca/wiki/Authentication-&Authorization>. [Accessed 15 01 2016].
- [6] S. Parikh and K. Stirman, "Schema Design for Time Series Data in MongoDB," MongoDB, 30 10 2014. [Online]. Available: <http://blog.mongodb.org/post/65517193370/schema-design-for-time-series-data-in-mongodb>. [Accessed 15 01 2016].
- [7] Espressif, "ESP8266EX Datasheet Version 4.3," 06 2015. [Online]. Available: https://cdn-shop.adafruit.com/product-files/2471/0A-ESP8266__Datasheet__EN_v4.3.pdf. [Accessed 15 01 2016].
- [8] Arduino, "Arduino UNO & Genuino UNO," Arduino, [Online]. Available: <https://www.arduino.cc/en/Main/ArduinoBoardUno>. [Accessed 15 01 2016].
- [9] M. Droettboom, "Understanding JSON Schema," Space Telescope Science Institute, 12 10 2015. [Online]. Available: <http://spacetelescope.github.io/understanding-json-schema/UnderstandingJSONSchema.pdf>. [Accessed 15 01 2016].

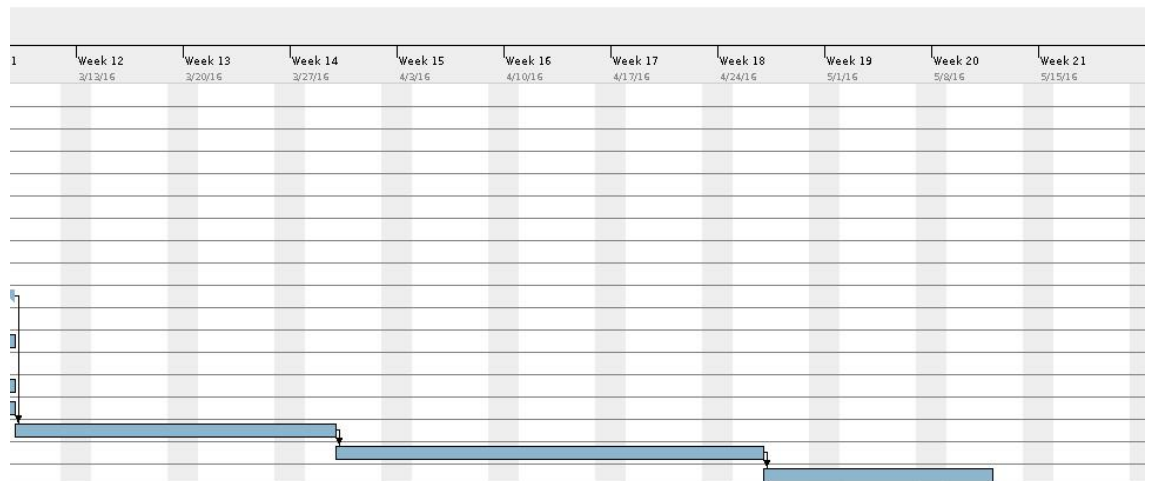
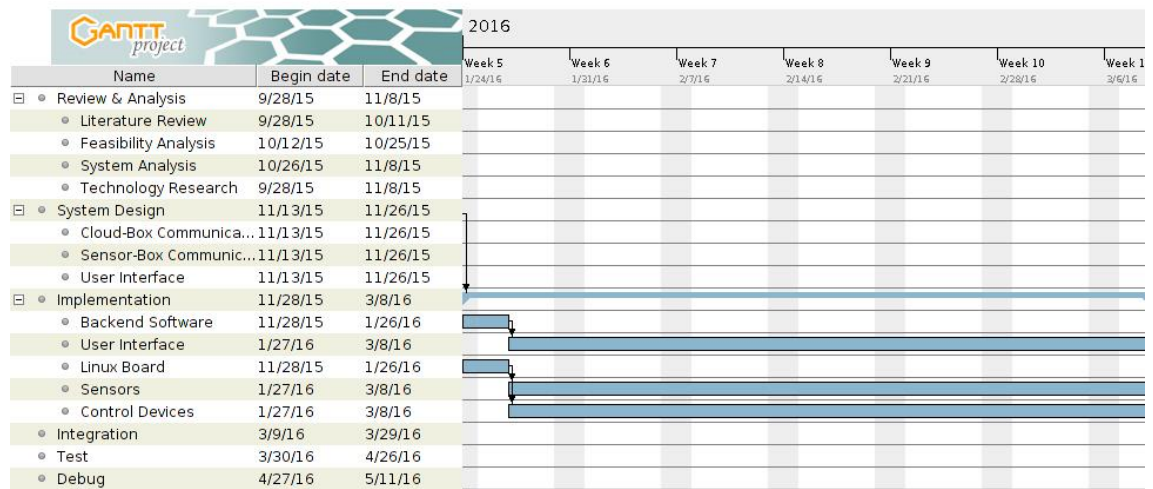
- [10] J. R. Knowles, "C++ Polymorphism vs. Arduino," 22 10 2012. [Online]. Available: <http://jamesreubenknowles.com/cpp-polymorphism-vs-arduino-1621>. [Accessed 15 01 2016].
- [11] D-Robotics UK, "DHT11 Humidity & Temperature Sensor," 30 7 2010. [Online]. Available: <http://www.micropik.com/PDF/dht11.pdf>. [Accessed 15 01 2016].
- [12] HANWEI ELETRONICS CO.,LTD, "MQ2 Gas Sensor Datasheet," [Online]. Available: <https://www.seeedstudio.com/depot/datasheet/MQ-2.pdf>. [Accessed 15 01 2016].

APPENDIX

APPENDIX-A



APPENDIX-B



CURRICULUM VITAE

Name Surname : Hüseyin Can ERCAN
Birthday : 10.02.1994
Place of birth : Çanakkale
High school : Çanakkale Fen Lisesi
Practical training : Computer Engineering
Work experience : None
Achievements : None
Memberships : None

Name Surname : Vedat Can KEKLİK
Birthday : 09.08.1994
Place of birth : Ankara
High school : Atilla Uras Lisesi
Practical training : Computer Engineering
Work experience : None
Achievements : None
Memberships : None