King Abdulaziz University

Faculty Of Engineering

Electrical and Computer Engineering (Computer)

EE495-Internet Of Things

Date : June 16

Section : GA

Semester : Spring 2022

Final Report

Digitization of Buildings to Allow Monitoring and Door Acces Control

Name : Hesham T. Banafa ID : 1742275

Name : Hussain Alhibshi ID : 1937555

Name : Osamah Badughaish ID : 1855395

Name : Asaad Waleed Daadouch ID : 1766585

Instructor
Dr. Saud Wasly

Content

1 Introduction

The need for smart systems in modern society is of great importance. Since microcontrollers, microprocessors, and single board computers are becoming ever more affordable, it is prime time for us engineers to implement robust systems that improve quality of life. In many forms of these systems, data can be used to perform analysis, and further optimizations for business or individuals. Moreover, through data analysis, security is inherently improved due to logging of events and actions, and other data-inferred information.

In this course, the fundamentals of the concepts of "Internet of Things" or IoT, are emphasized and summarized in four, well known, fields of engineering. In essence, many of the engineering requirements for systems that collect data from sensing, take action, are connected to a network, and are controllable, are at the core of the Electrical and Computer Engineering program. Thus, embarking on this project and challenge as senior students of said program, becomes a method for exposing us to cutting edge technology in terms of microcontroller architecture and capabilities, software libraries, protocols. The skills required to implement a full system for IoT devices are sharpened by using them in real practical application. Furthermore, as part of this project, we are exposed to external material and skills, such as Mobile Application Frameworks, Data security, and Novel Sensor designs.

In the following sections and chapters, we present our developed project in detail, which is split into 4 different subsystems. The first subsystem is the Backend or Backbone of the whole system, enabling a central point of communication for devices and user applications. Second, the mobile application made for users and administrators to control, use, and view the system is discussed, in terms of User interface (UI) design choices and communication methods with Devices and the Backend services. Third, the IoT Door system is detailed, going over the implementation choices in terms of hardware and software, safety and security mechanisms, and failover modes. Finally, the Iot Monitor subsystem is detailed, in the same vein of the IoT door. Moreover, a novel method for tracking the number of people present within the scope of a single room.

1.2 Objective

The Objective of this project is to design and develop a full system to control room authorization, collect sensory information and metrics, and showcase presence of people in a case of emergency, to aid in the evacuation efforts, in a target building. In this project, we are expected to employ engineering design techniques for solution evaluation and generation, electrical design and interconnection, and software programming.

1.3 Internet of Things

As mentioned above, the concept of Internet of things is the application of four pillars [4] to satisfy the criteria.

- Device/Controller (Thing)
- Connectivity
- Data Collection and Processing
- User Interface

The work in [3] also defines these in terms of layers. The first layer is the sensing layer, where data is collected by the end IoT device via any method and measure. The second layer is the network layer, where communication and data transmission is enabled by networking standards, or a novel method of achieving the same objective. Li et al [3], also denotes a service layer, where business logic, database, and other service oriented IoT systems are residing. Finally, an interface layer is defined where customer, user, and administrator interfaces are provided via an Application Programming Interface (API), Native Personal Computer Graphical programs, or mobile Applications. In this project's subsystems, there is overlap between the layers, however, all the corners of a full IoT system are met and integrated together, to achieve the end goals of the project.

Internet of things are designed with many architectures, depending on the target application and objective. As of recent, IoT systems have become at the core of home automation, spawning an emerging market of hardware and software vendors, offering the end service of home automation, control, and monitoring. This kind of architecture is common and is mostly centralized on one or multiple gateways that communicate with the vendor cloud services. Moreover, another, less common, architecture relies on the communication between the devices themselves, in absence of a gateway or central point, often referred to as Machine-to-Machine (M2M). The former requires every device to communicate with the gateway directly, thus, reducing the complexity, at the cost of limited scalability. We note however, an IoT system during the design phase is open-ended, in terms of network topology and architecture, as long as the four pillars are met. In this work, we architect the system to be more centralized on a robust backend that enables secure communication, session tracking, and stateful control for IoT devices and end-user applications. And IoT devices are designed to be a building block, that are able to be standalone, not limited to sensing, or actuation.

Ray in [1] defines IoT as “system that performs various types of functions, such as services involved in device modelling, device control, data publishing, data analysis and device detection”. Hence, it summarizes the four pillars mentioned above. In the following we define each of items listed above.

- Device: The device or “Thing” that is at the core of IoT technology. The device enables control, data processing and storage, sensor operation, actionation, and monitoring of target objectives.

- Connectivity: The IoT echo-system requires connectivity to be of utility to the users. Moreover, It is strictly required to have communication capability to be classified as an IoT device/system. Through communication, monitoring, control, and data collection can be performed on the target device from user applications or programs.

- Data Processing: Data collection and processing is at the core of an IoT system, where data can be used to produce and deduce informed decisions, whether manually or through decision trees, Neural networks, SVN or other types of classification and pattern recognition. Measurable improvements in customer experience, home efficiency, and worker satisfaction should be the result of optimum use of an IoT framework.

- User interface: User interfaces should allow administrators and users to access the above information and controls over devices and data. A user interface can be a Web Application, Mobile Application or a custom interface integrated on the IoT device as an embedded system.

Smart city initiatives are a major factor and drive in the acceleration of technological developments in the IoT field. At its core, a smart city contains thousands of interconnected devices, forming a large network systems that enables municipalities to respond to emergencies more efficiently, among other uses. Smart cities essentially are a large-scale implementation of an IoT system. We argue the evolution of smart devices is a positive trend in the adoption of IoT systems, however, with a lack in communication and data analysis. What was previously defined as “Smart Device”, was only a means to automate a basic task, mostly without the interconnection of modern IoT systems [1-3].

1.4 System Overview

In this section, the high level view of the major system/project component are presented, and the communication interfaces are detailed. The designed system can be divided into 4 subsystems, where 2 are of the same class. In figure 1, the “Door” and “Room” elements are IoT end devices where data acquired and transmitted, not stored, and actions are taken based on user commands. For the IoT Door subsystem, the main function is to communicate with the backend and receive “open door” commands, with their accompanying information. For the Room element, multiple sensing functions are established and developed to give information about the room environment. The element “Backend/API”, provides an interface for IoT devices and user interfaces, a way of bidirectional communication. Moreover, other functionalities include user authentication, permission management, resources management, data storage and analysis. The final subsystem is the mobile application, where the user directly interacts with the system as a whole, by authenticating, viewing, and controlling the underlying system, as per permissions provided by backend; where the application, mainly interacts with the backend, not the end devices.

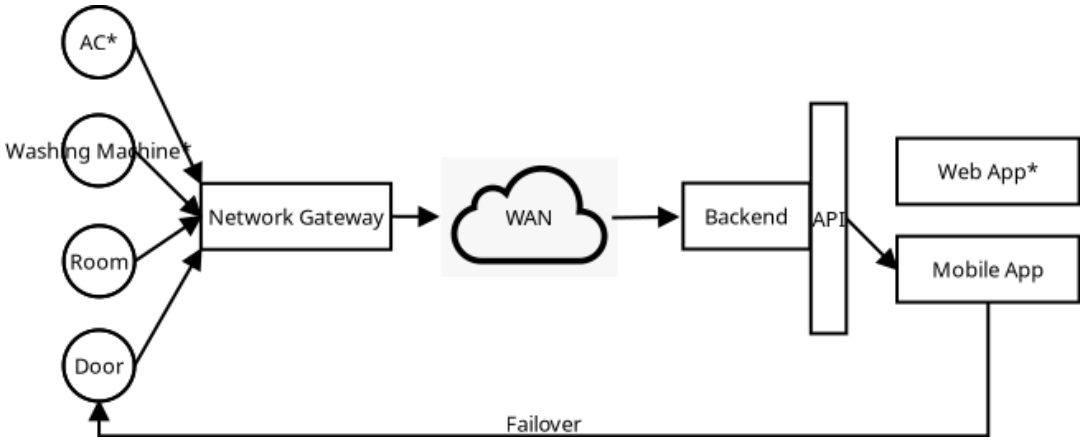

Figure 1: High level view of the designed system and architecture.
*denotes the element is not implemented, but is for demonstration

1.5 Modes of Operation

Normal mode: When the controller has access to the server and can send and receive data from it, this is the normal state. In this instance, the user must additionally contact the server to control the door and acquire information about the rooms to which he is permitted.

Abnormal mode: When the controller is unable to access the server, which might occur due to a Wi-Fi connection disruption or a server breakdown. Only by connecting directly to the controller via Bluetooth will the user be able to unlock the door in this circumstance. Furthermore, unless the server has previously granted him permissions, the door will not be unlocked for him. Because of that, the data of the users who have access to the door will be stored in the controller in normal mode.

1.6 Protocols & Standards

From previous sections, we denote communication without detailing the used protocols. In order to enable the IoT devices to be widely and easily reachable, the already implemented networks of the Internet are leveraged. There are two methods that are proposed to achieve this, both with their advantages. First, is to have the minimum addressable IoT device to be directly addressable through an IP address (internet routable IPv6 or NAT IPv4), where devices communicate directly with the backend, through the internet. This method gives more flexibility for the backend to have fine-grained control and data analysis at the cost of more complexity in logic and types, and there is not a single point of failure. The second method abstracts the types and devices behind a single gateway. Where the backend only communicates with, for example, a room where a door and monitor exists. This method essentially reduces complexity, however, it introduces a single point of failure in the gateway. In the implementation of this project, the first method is used, and the TCP/IP model layers are briefly discussed, regarding the design choices with a focus on the Network and application layers.

Table 1: Summary of Networking Standards (MAC, PHY) [3, 4]

Standard	Power	Range	Data Rate
IEEE 802.15.4 (ZigBee)	Low	10-100m	20-250Kbps
IEEE 802.11 (WLAN)	Medium	100m-3km	10-100+Mbps
IEEE 802.15.1 (Bluetooth, BLE)	Medium, Low	1-100m	1-3Mbps
IEEE 802.15.6 (WBAN)	Low	2m	2Mbps

The Standards denoted in Table 1 combine the physical layer (PHY) and the Media Access layers (MAC) or Data Link, which provides the low level communication below the Network layer (IPv4, IPv6). The ZigBee standard has multiple operation modes, one of which is appealing to this project called mesh. It is possible to make end devices communicate with each other in Machine-to-Machine mode, and allows a device out of the range of the main hub to relay it's messages through another node in the network. We note that the IEEE 802.15.4 standard by itself does not include the Network and Application layers. Those are defined by ZigBee standards (ZigBee Wireless Networking) [5]. However, the performance, reliability and compatibility of ZigBee networks are low when compared to IEEE 802.11 or "Wi-Fi".

Bluetooth is an older standard and is well established and is implemented for many applications such as speakers, computer mice, keyboards among others. However, the standards libraries developed to leverage Bluetooth technology are targeted for Machine-to-Machine, or Point to point communication by using the Media Access Control (MAC) and Physical (Radio) layers, without implementing further layers to leverage the full TCP/IP stack. While it is certainly possible to implement this solution, it is outside the scope of this work. We note, however, Bluetooth is used as a fallback method for the mobile application and Iot devices to communicate directly with each other in case of the absence of the backend or other networking failures.

The IEEE 802.11 (Wi-Fi) standard is the most widely used wireless standard in most applications that require network access. There is a high availability of programming interfaces and libraries that leverage the Wi-Fi hardware. On the other hand, the standard uses more power, to ensure a high data rate and packet delivery in a highly congested frequency band. But for this application, the devices are not energy-constrained. The ESP32 board includes hardware for both Wi-Fi and Bluetooth, and exposes the hardware through their standards development environment and SDK. Comparing the hardware availability of Wi-Fi and ZigBee, Wi-Fi is still more widely available, and offers compatibility with older revisions of the standards. Thus, for this project, the Wi-Fi standard is used for IoT devices to communicate with the network and Backend.

Table 2: Summary of Application Layer Standards

Standard	Overhead	Encryption
HTTP	Medium	TLS
CoAP	Low	DTLS
MQTT	Low	TLS Payload Encryption

For the application layer, the standards listed in Table 2 are commonly used for IoT solutions. HTTP is at the core of most web communication, and is widely adopted for not only web sites and web applications, but also for applications that are not specific to the web browser. Essentially, HTTP defines a method for request-response communication and standard response codes, usually over TCP/IP. Furthermore, it defines metadata of the raw data exchange specific to the application. From [6], the definition is provided as “HTTP/1.0, as defined by RFC 1945, improved the protocol by allowing messages to be in the format of MIME-like messages, containing metainformation about the data transferred and modifiers on the request/response semantics.” The general architecture of HTTP data exchange is Server-Client, where the server contains a process listening on a specific, fixed port, for example 80 or 443, and the Client application initiates a request containing the desired endpoint, headers, and data, as per the HTTP standard.

Example of HTTP request-response:

```
> GET /users/me/ HTTP/1.1
> Host: 192.168.0.10:4040
> User-Agent: curl/7.83.1
> accept: application/json
> Authorization: Bearer eyJ0
nQQRL1KNCAy5XzCLMeGQCsKkhyc
```

Figure 2: Example of outgoing HTTP request with endpoint “/users/me/”, host address 192.168.0.10:4040, and headers “accept”, “User-Agent”, and “Authrization”

```
< HTTP/1.1 200 OK
< date: Sun, 12 Jun 2022 12:42:2
< content-length: 270
< content-type: application/json
```

Figure 3: Example of incoming HTTP response with Response Code 200 or OK, heads “content-length”, and “content-types”

The HTTP Request-response model is great for multiple, independent users (Clients) connecting to a single (Or multiple, by DNS manipulation of replicated service servers) server to fetch and exchange data. However, in a large-scale IoT deployment, this model requires added layers for scaling and maintaining a stable, and low-latency.

The new CoAP standard is similar to HTTP in terms of architecture, and how the metadata is formed. However, it is named “Constrained Application Protocol”, due to its very limited use of energy, mainly for devices that are battery powered. The implementations of CoAP are limited in abundance, and study and development materials are lacking. Still, in our testing, CoAP proved to be quite simple to use and draws many of the aspects of HTTP, in an efficient manner.

MQTT Standard takes a different approach to tackle the scalability and introduces a method named “Subscriber-Publisher”; where a central hub, named *broker*, is used to receive a *publish* message, and relates it to whoever is connected and is *subscribed* to the *topic*. This method, with a large number of devices, allows the publishing device to only send the message once, thus, offloading the work required to send the message to each listing device to the *broker*. The developers/Engineers who design the system, are required to define the topics/channels with the required granularity to achieve the desired outcome of the system. This approach is advantageous in many aspects. However, in this project, the user application is using HTTP (HTTPs) to communicate with the backend and the devices are not energy constrained. Thus, it is decided to use HTTPs for all the systems as the application layer protocol; minimizing the design overhead and complexity for this demonstration.

2 Backend & Backbone

The backend is the main hub of the IoT system that we are designing. It relates messages from IoT devices to user applications, via a stateful means. Thus, we define this subsystem as a Hub for authenticated user interaction with IoT devices, and a data store for access logs, connection history, and emergency notification, via Representational State Transfer Application Programming Interface (REST API) [7]. The term “Backend” is used to distinguish and compartmentalize subsystems of a larger system from “Frontend”; where “Frontend” is defined as the user application, that communicates with the backend that contains the authentication, database, business logic, among others depending on the application. This separation allows for flexibility that is now required to support user applications that are on multiple platforms, such as Windows PC, Linux, iOS, Android, and others. That is, a single backend application can interact with any frontend that implements its interface properly.

In the following sections, the details of the design backend server are showcased. Moreover, the subsections 1, 2 and 3 discuss the libraries and frameworks that are used in the implementation of the server, and sections 4, 5 and 6 discuss the steps taken to achieve the desired functionality.

2.1 Protocols

From the previous section, the HTTP protocol is used to implement the API endpoints. The general architecture of the system is a Server listening on port 4040, addressable by an IPv4 address, in a private Local Area Network or Wide Area Network. Thus, we leverage the HTTP request-response model by using the predefined status codes, or response codes in Tables 3 and 4.

Table 3: HTTP response code categories as per RFC 7231 [8]

Category	Response Code Range

Informational	100 - 199
Successful	200 - 299
Redirection	300 - 399
Client Error	400 - 499
Server Error	500 - 599

Table 4: HTTP response codes of use in this project [8]

Code	Category	Reason-Phrase
200	Successful	OK
400	Client Error	Bad Request
401		Unauthorized
404		Not Found
500	Server Error	Internal Server Error

From Tables 3 and 4, we note the Error codes in the categories “Client Error” and “Server Error” require to be distinguished. “Client Error” is only returned by the server if the server logic is certain of an error on the client side or in the sent data format and semantics. “Server Error” category errors are returned whenever the server faces an unexpected error during runtime. Having drawn this distinction is critical to the development of frontend applications. Server errors are the responsibility of the backend to ensure they don’t happen due to internal logic, rather only happen if an external factor affects the running logic. For example an internet outage on another server that is required for the application, may introduce an unexpected error. However, it is possible to also account for this error in a different matter, and report what exactly happened, rather than use the generic error.

In the following we showcase who the HTTP server is written in python leveraging FastAPI; a framework developed in the Python programming language.

2.2 FastAPI

FastAPI is used in this project to expose the backend API to the network via HTTP standard methods. The framework is written in Python and is packaged as a library that exposes functions and decorators that implement the HTTP request types. However, the FastAPI decorators by themselves are not standalone, and are not a server. Thus, as per FastAPI official documentation, a WSGI server is required to bind to an address on the host and start listing and processing request; described by the FastAPI methods. Other critical libraries used to achieve the desired functionality are listed below.

- FastAPI (Declare HTTP logic)
- Uvicorn (Web Server)
- NGINX (Advanced Web Server)
- Pydantic (HTTP input validation and ORM linkage)

- SQLalchemey (Database, ORM)

```
@app.get("/users/me")
def  get_my_information(db : Session = Depends(get_db)):
     return { 'key' : 'value' }
```

The above Python code is the general pattern to use FastAPI to expose the functions (endpoints) to the WSGI server. We infer the HTTP method from the above is 'GET' and the endpoint is '/users/me'. We note the parameter db is a *dependency* that is met by executing the function 'get_db', that yields a Database session. A database Session is not required for all use cases, but in our project it is required. The return value of the function is received by the client as a json body, and an implicit HTTP code 200 OK.

```
@app.post("/users/register")
def  get_my_information(db : Session = Depends(get_db),
                        request: UserCreate):
     return create_new_user(request.username, request.password)
```

The above example showcases an HTTP POST method that enables the client to include a *body* containing plaintext or json, depending on the declared type of UserCreate. From the FastAPI documentation, leveraging the library pydantic, input *schemas* are defined in python classes in the form:

```
class UserCreate(BaseModel):
     username: str
     password: str
     Email: str
```

Using the above class as the type hint for *request* parameter, tells FastAPI logic to enforce this schema in JSON. Thus, if the user violates this strict typing, a 'Client Error' status code is returned. In other words, the Pydantic Python classes allow us to declare the json key schemas and content types of the value pairs.

Summarizing how uvicorn and FastAPI interoperate, a Uvicorn instance is running with one worker that is the main running process on the host operating system. It binds to the user specified address/port, and listens for incoming requests. An incoming request, if it is a valid HTTP request (Here HTTP Protocol standard is enforced), is then relayed to a running python interpreter with the specific endpoint declared in the FastAPI implementation. Whatever the python function returns in body and status code/reason is served up by the Uvicorn server. Note, it is mentioned that one worker is running. Uvicorn, and other servers, allow running multiple instances of the FastAPI interpreter. This technique is used in many modern high traffic applications to reduce latency and leverage the async nature of applications and the underlying hardware. Linux operating systems allow for the same application, given it is the same binary, to bind multiple instances on the same port, while the Linux kernel load balances the incoming connections among the instances.

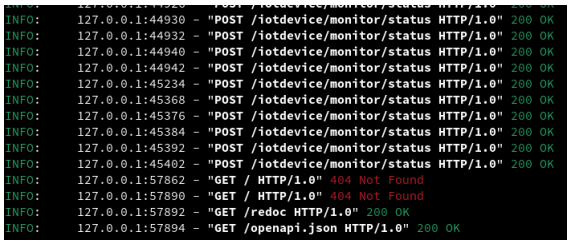

Figure 4: Uvicorn output showing HTTP request type, endpoints and response code. Note the incoming IP address are only form 127.0.01 or localhost, since Uvicorn is instructed to listen on localhost, while NGINX is used as a reverse-proxy that listens on 0.0.0.0 and forwards to Uvicorn as IPC over TCP/IP

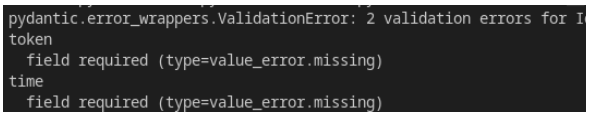

Figure 5: Pydantic schema enforcement validation error. Key-value pairs missing.

Table 5: User Endpoints '/users/'

Endpoint	Function
/reg	Create User
/me	Get my Details
/open	Open Door
/close	Close Door
/accesslist	Personal Access list
/updatepassword	Change password
/tkn	Generate Access Token (Login)

Table 6: IoT Device Endpoints '/iotdevice/'

Endpoint	Function
/door/status	Polling for doors
/monitor/status	Polling for monitors
/iotdevice/door/users	Serves a list of authorized usernames
/iotdevice/door/tkns	Serves a list of authorized users temporary passwords

Table 7: Admin management endpoints '/admin/'. (Curly braces '{}' denote a URL parameter)

Endpoint	Function
/users/	Lists all users
/iotentities/	Lists all Iot Entities (Doors)
/monitors/	List all monitors
/iotentities/create	Creates a new Iot Entity (Door)
/monitor/create	Create a new Monitor
/users/{user_id}	Get user profile
/users/allowdevice/id	Authorized User to Door
/users/disallowdevice/id	Remove User to Door authorization
/users/{user_id}/deactivate	Deactivate user
/users/{user_if/activate	Activate user
/users/iotdevice/gentoken	Generate permanent IotDevice Token
/link/monitor/{monitor_id}/door/{door_id}	Link a monitor and door (room)
/user/accesslog/email	Get access log for user (email)
/user/accesslog/username	Get access log fot user (username)
/roominfo/{door_id}/now	Get current room sensor values
/roominfo/{monitor_id}/now	Get current room sensor values
/roominfo/{monitor_id}/last/{count}	Get sensor reading history
/roominfo/accesslog	Get access log for door

Emergency triggers occur as soon as the Room Monitor reports temperature readings above 40 degrees Celsius or smoke sensor readings above 1000 units (parts per million) of toxic gas compounds. In a case of emergency, the doors associated with the room are unlocked to remove the need of using the push button. This is due to the possibility of low visibility during and emergency of this nature. A higher goal is to leverage the information of a full building to *guide* people to the nearest exit, to reduce the evacuation time.

2.3 Database & ORM

For the backend to achieve the desired functionality, persistent storage is required. Fortunately, the FastAPI also documents the process of initializing a database engine and basic database operations using SQLalchemey; since it is common for REST APIs to have a backings store. Moreover, it is further detailed and emphasizes the programming method of Object Relational

Mapping or ORM. ORM Allows for SQL database quarries to be abstracted behind a Python class object. During the early programming stages of this project, this concept was confusing at first, but after it was fully understood, it proved to be quite efficient in terms of programming time.

In pure SQL relational database terms, there are multiple forms of relations between table entries. In order to proceed in showcasing the database schema, these types of relations are to be presented. A basic independent database entry is a row in a table, with features (values, information, etc…) are in columns. In our database, the "user_accounts" table contains all the user information. A basic implementation would be:

Table 8: Basic user_accounts database schema

user_accounts	Data Type
id	INTEGER
username	VARCHAR
email	VARCHAR
hashed_password	BYTES
salt	BYTES

This implementation, if used, does not leverage any of the relational features of modern Database engines. The only advantage of using this table over a basic text csv, is the indexing feature that allows faster lookup times.

In this project we require the user to have a defined list of available or authorized devices to use. To accomplish this, a relation is defined to each authorized device, by having this relation we are required to implement a many-to-many relationship. A third association table, alongside users and devices, is required between Devices and Users, containing a user_id and device_id for each row. Thus, each User has a relation to many devices, and each device has relations to many users. To accomplish this in Python and SQLAlchemey declarative nature, we do the following.

```
class User(Base):
      __tablename__ = "user_accounts"
      id = Column(Integer, primary_key=True, index=True)
      email = Column(String, unique=True, index=True,
nullable=False)
      username = Column(String, unique=True, index=True,
nullable=False)
      hashed_password = Column(String, nullable=False)
      passwd_salt = Column(String, nullable=False)
      is_active = Column(Boolean, default=True, nullable=False)
      last_token = Column(String, nullable=True)
      connections = relationship("UserConnectionHistory")
      authorized_devices=relationship("IotEntity",
secondary="user_iot_link", back_populates="authorized_users")
```

```
    access_log = relationship("DoorAccessLog",
back_populates="user")
```

Note the python constructor `Column()` is the basic type used to declare columns. The relationship constructor does not declare a column, but in the background via the `secondary` parameter tells SQLAlchemey to reference the association table “user_iot_link”. We then can leverage this relationship in runtime by using the class database object User and fetch the authrized devices by

`devices = user.authrized_devices`

This yields a python list of IotEntity database objects. In the background, SQLAlchemey executes an SQL query on the database using the User object relationships to fetch the devices using the association table in the case of many-to-many relationships. Thus, the programmer can now use python objects as if they are in memory, while at runtime, they are only loaded when they are referenced (lazy loading).

An example of many-to-one relationships from the above example SQLAlchemey Python declared table, is the access log field. Each access log entry is linked to exactly one user (and door). Thus the ForgienKey field contains a key from the user table. To access the database objects, the same semantics for authorized devices are used.

`access_log = user.access_log`

Where the result is a list of access log table elements.

```
In [15]: user.access_log
Out[15]:
[<sql_app.models.DoorAccessLog at 0x7f32c6fe8700>,
 <sql_app.models.DoorAccessLog at 0x7f32c6fe81c0>,
 <sql_app.models.DoorAccessLog at 0x7f32c6fe8220>,
 <sql_app.models.DoorAccessLog at 0x7f32c6fe8280>]
```

Figure 6: ORM database access example. DoorAccess log is a defined table in python.

It is common practice to use indexes on columns that are frequently queried or used as lookup for an entry. Note the parameter `index=True` In the user account table. Using indexes is greatly beneficial, however, it can increase the size of the data on disk if a naive approach of using indexes on every column. We summarize the full database schema in the following.

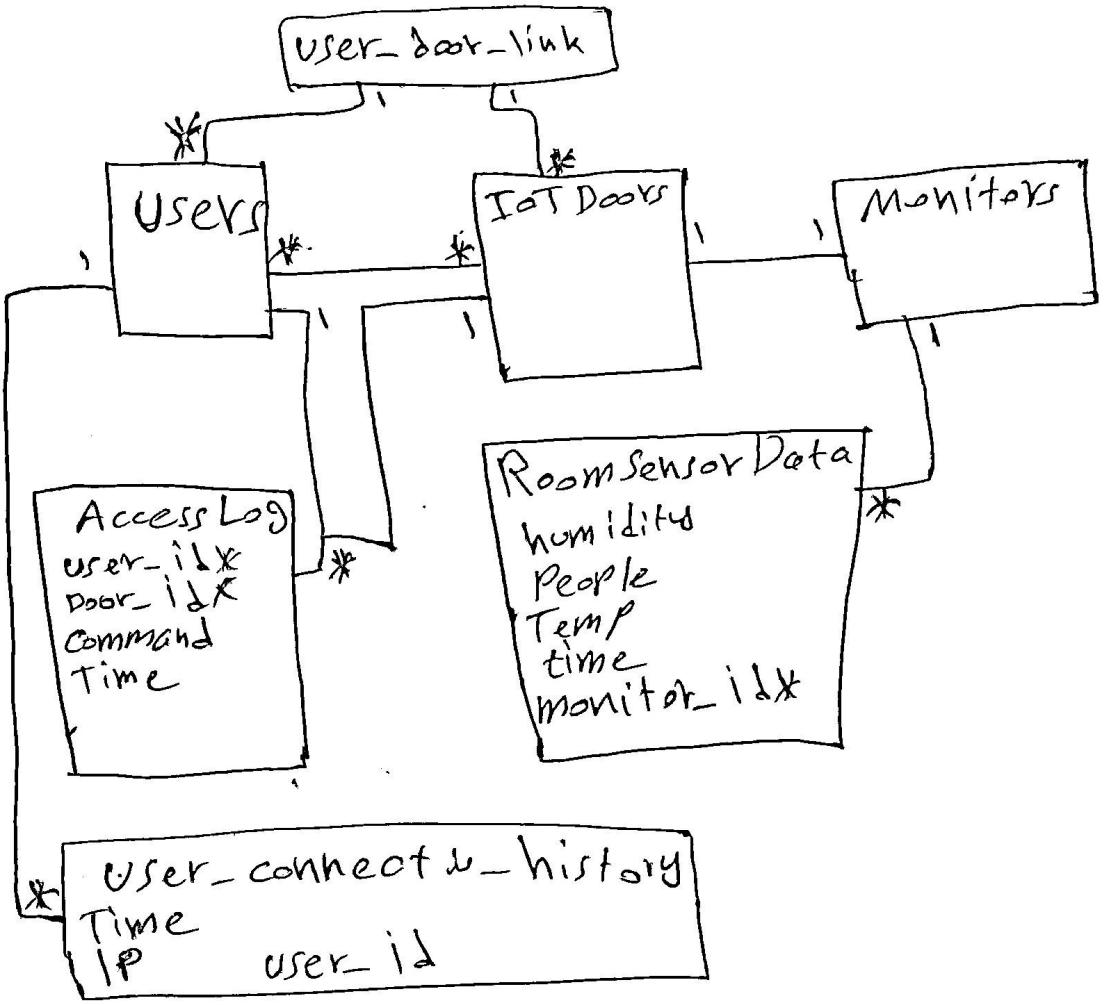

Figure 7: Database schema and layout of relationships. 1-1 is one-to-one, 1-* is one-to-many and *-* is many-to-many

The database is initialized with initial data for faster testing and redeployments. The user passwords are defined in environment variables, discussed in later sections.

2.4 Cloud Hosting

During different programming stages, the HTTP server was hosted on multiple servers to expose the server over the internet for remote access. The first test of hosting was on a Personal deployment of Proxmox Virtual Environment cluster, as a container (process level virtualization). Containers have an advantage of having a high level of isolation with less overhead when compared to full hardware virtualization, since containers use the host kernel, hardware and other resources except the network and file system. Another advantage of using containers is that the virtual machine is decoupled from the hardware, thus, it can be moved to another host/node while running if the need arises. The host hardware is in a residential area and connection. Port forwarding is a technique to allow for a host/server to receive incoming connections from the internet, while using Network Address Translation, by inserting a rule in the home router/gateway to forward packets with port number 4040, to the LAN address 10.0.0.5, for example. With this, the IP supplied by the ISP, routable in the internet, is now listening and responding on port 4040, through the internal container logic.

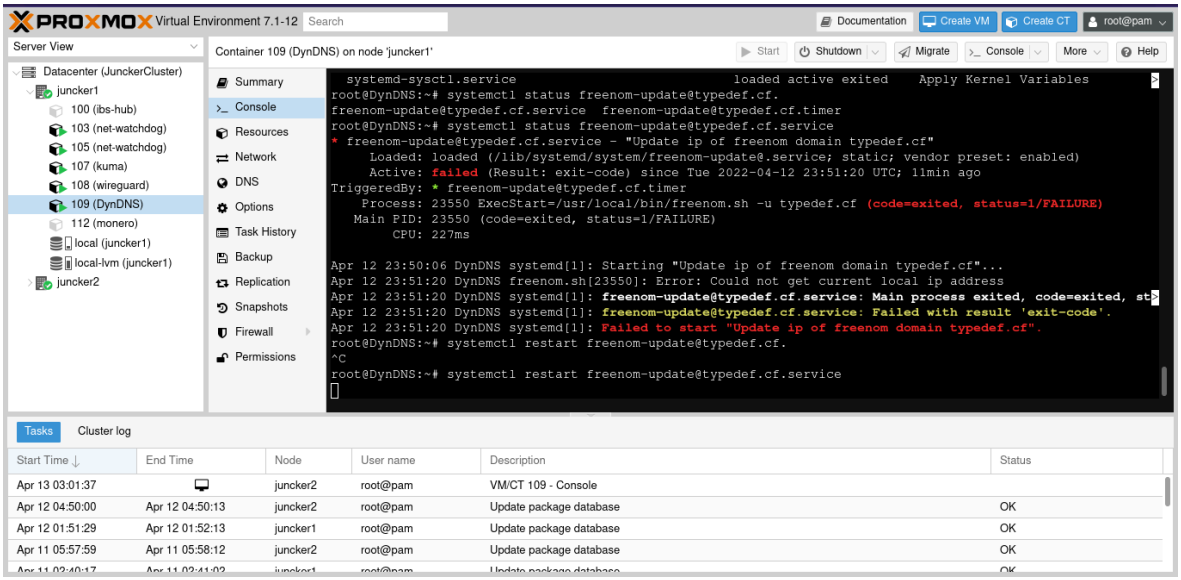

Figure 8: Proxmox Virtual Environment

Due to uptime and availability issues, the server was moved to a cloud provider that offered better uptime and availability.

A systemd unit file was made to ensure the service (web server) is running when the host is rebooted, or a crash occurred in the application. A systemd unit offers the option to auto-restart on crash or unclean exit.

```
[Unit]
After = network.target

[Service]
User=ibs
Group=ibs
WorkingDirectory=/srv/ibs/ibs
ExecStart=/srv/ibs/ibs/run-tls
Restart=on-failure
RestartSec=15

# Security
ReadWritePaths=/srv/ibs/ibs
PrivateDevices=yes
PrivateMounts=yes
PrivateTmp=yes
PrivateUsers=yes
ProtectClock=yes
ProtectControlGroups=yes
ProtectHome=yes
ProtectKernelLogs=yes
ProtectKernelModules=yes
ProtectKernelTunables=yes
```

```
ProtectProc=yes
ProtectHostname=yes

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

This unit file ensures the correct order during the boot sequence by “After=network.target” and is “WantedBy=multi-user.target”, which is the most common run level on UNIX/Linux systems. Other security measures are enforced on the unit. For example, the web server should never want to change the system clock, read other process tmp directory, read other users home directories, and so on.

2.5 Addressing Security

In today's technology environment, cyber security is at the top of most programmers and engineers, to avoid being exploited or threatened due to relaxed security standards. In this project, we have taken steps to ensure the highest level of security is maintained, to the best of our knowledge. A good general rule is to have every access rejected, and go through a process of allowing only those who are concerned to gain access. This, in most cases, can reduce the attack surface of the target system. In the following we discuss some of the measures taken to get closer to a secure system.

In the previous sections describing HTTP standard and protocol, encryption was no menioted. In most use cases of HTTP currently, a layer of encryption is added to avoid eavesdropping and malicious attacks. Transport Layer Security (TLS) is the standard used with HTTP and overall is called HTTPs. TLS standard provides methods of exchanging public Asymmetric keys to finally agree on a session symmetric encryption key for fast transmission. Diffie-Hellman is a common method to do the handshake. A TLS tracks a session to reduce the overhead of a handshake for every HTTPs connection. An HTTPs request content is fully encrypted, and only the end-application can read the contents. Meaning, the lower layers of Transport and Network are not encrypted to achieve their goals. Only above the Transport layer is encrypted. To be able to use HTTPs, a certificate should be present. For public websites, the certificate needs to be digitally signed by a well known and acknowledged authority, so users can trust the certificate. A small script is developed to generate self-signed certificates using openssl library.

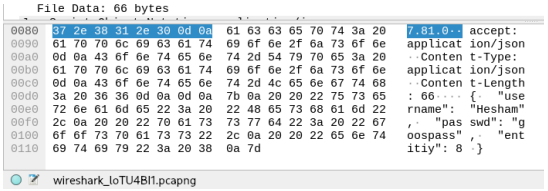

Figure 9: Unencrypted HTTP data is clear to read

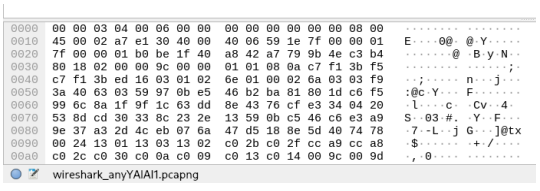

Figure 10: Encrypted HTTP using TLS hides all HTTP content (Application layer)

The password storing mechanisms are of grave importance. If user passwords are stored as is, and a data breach occurs, the users can be quickly hijacked by criminals without a chance of notifying them to update passwords. Thus, it is standard practice to store passwords in a hashed format, and compare hashes when a user enters a real password. Hash functions such as SHA1, produce a different pattern for a given input.

- “hello, world” cd50d19784897085a8d0e3e413f8612b097c03f1
- “Hello, world” 7b4758d4baa20873585b9597c7cb9ace2d690ab8

Note that the hash function is sensitive to single bit changes. The resulting hash, in a stronger hash function (SHA256), can not be reversed easily. Thus, if a data breach occurs, a significant amount of time is required to be able to reverse the hash. To further increase the difficulty of finding the reverse hashes, a random string of bytes, called salts, is appended to the clear password, forcing the bad actors to rehash and test a significantly larger space of search. This is to combat pre-calculated hash tables. Also, salts allow exact user passwords to be stored as different passwords in the database, or at least appear different to the bad actors. Hashing functions are also used for other purposes, such as file integrity; since they are sensitive down to single bit changes. File integrity systems are used to notify administrators of unauthorized file change in a secure environment. Moreover, SHA256 is also used in blockchain technology and provides the “chaining”, to ensure the blocks are not altered.

Authentication for the backend is done using JSON Web Tokens (JWT). JWT is a standard used to track sessions and authenticate users using bearer tokens. Bearer tokens are like signed paper that contains the user information and expiration date. Say, User Hesham is allowed to use this up until next Tuesday, and so on.

- eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJibHVldG9vdGhfbWFjIjoic3RyaW5nIn0.ELl5AfBR1NdM4_OFhl_SCTm9EMPpqjiCKOSS0CrOJps

The above example of a JWT token is signed with a secret key, also generated alongside the certificate. The red part contains algorithms used to sign and hash the data. The blue section is the data, to be later trusted if the Green part is correctly verified. A user can obtain a token using the ‘/users/tkn’ endpoint, by supplying username and password. Note, the JWT tokens are not encrypted, but are encoded in Base64. In other words, the token is a signed message, if altered, the decoding process rejects the signature, via mathematical means.

The above mentioned JWT tokens are used for session tracking and reduces the transmission of the real password. However, another form of authentication to manage the Backend as an admin is also implemented via an APIKey. FastAPI exposes Python types that can be injected a dependency on an endpoint, where it enforces the existence of a certain header, alongside user defined logic for authentication. The type *APIKeyHeader* is used to declare a key name required in the request headers. Internal logic compares an existing environment variable on the server with the incoming key to validate the administrator.

```
@app.get("/admin/iotentities/",
         response_model=List[schemas.IotEntity],
         tags=['Admin'])
def read_iot_entities(skip: int = 0, limit: int = 100,
                      db: Session = Depends(get_db),
          api_key: APIKey= Depends(auth_helper.valid_api_key)):
     iot_entities = crud.get_iot_entities(db, skip=skip,
limit=limit)
     return iot_entities
```

The *api_key* is an injected dependency on the endpoint and requires the function `auth_helper.valid_api_key` to be run without *raising* an HTTPException.

Both the JWT tokens and API key require secret generation with high entropy. Thus, a simple script is made to initialize the environment variables and enforce the read/write file permissions to be strictly for the running user (OS user). OpenSSL is used to generate a cryptographically secure random sequence. The next bash script generates random sequences.

```
if [ -z $(command -v openssl) ]
then
      echo "openssl not installed"
      exit 1
      fi
openssl rand -hex 32
```

The following script initializes the environment variables for APIKey, JWT secrets and the initial users passwords.

```
echo "API_KEY=$(./gen_secret.sh)" >> .env
echo "API_KEY_NAME=big_boy" >> .env
echo "jwt_secret=$(./gen_secret.sh)" >> .env
echo "jwt_algorithm=HS256" >> .env
read -s -p "First User password: " firstpass
echo
read -s -p "Retype First User password: " secondpass
echo
      if [ $firstpass != $secondpass ];
      then
            echo "Passwords dont match!"
            exit 255
      fi
echo "first_user_pass=$firstpass" >> .env
chmod 600 .env
```

Another security measure taken is to reduce the amount of information leakage. Information leakage can give an unauthorized person (program) the ability to map the devices and user information, via attempting false credentials, and interpreting the response carefully for changes. In other words, if a person is trying to list the email address stored in a naive database and backend implementation. The person can supply false email/password combinations, by their own defined list of test emails. If the backend server responds differently for false email and false password, enough information is given out to the attacking person to record which emails passed the internal “Email exists” check. To combat this form of attack, a universal response of “Email or Password is incorrect” prevents the leak. Thus, only when a valid set of username and password is supplied, then do we give

permission. In this way we do not give the chance of distinction between username failure and password failure. The same process is made for Iot Door and Monitors.

2.7 Unit Testing

Unit testing is a good practice and acts as a check on the changes, or it can be a declarative method of programming. A set of tests are written (pytest) with a prefix function name “test_”, will execute tasks on the target system logic, and assert what is expected. If the assertion fails, it is noted and reported. This allows the programmer to make changes without being afraid of a cascade of broken logic, to a string of changes.

```
def test_create_user():
      response = client.request("POST", "/users/reg",
                                json=get_user_json(test_user),
                                headers=common_headres)
      assert response.status_code == 200
      post_request(response)
```

Having written comprehensive unit tests, the programmer can make changes to the code, and run the tests to ensure that the changes do not break the expected behavior. This technique is used and declares the desired behavior, then the core logic is written *after*. Pytest, the testing framework used, is run on a directory, and by default it searches for python files with the prefix “test_”. Then within those python files, functions also with the same prefix are run. Helper functions for setup and teardown are allowed to be declared and used. When running the tests, a new test database is created and the API endpoints are tested. This method contains more options in terms testing paradigms and code coverage requirements that are outside the scope of this project.

3 Frontend And User Application

Front-end, also known as client-side development is the practice of producing an application so that the user can see and interact with other devices directly. The goal of frontend application design is to ensure that when the application starts, the users must see the information in a format that is relevant and easy to read. For this project we used the Flutter framework to create the frontend and UI. Flutter comes with a dynamic library of customizable widgets, making app development easy and fast. So, the application is going to have four main pages for the user to use which are: Register page, Login page, App page, Admin app Page.

3.1 Login page

This page is where the entire program starts, which accepts two inputs: Username and Password. Also, it has two other choices which are “Forget password?” and sign up, as shown in the figure 11:

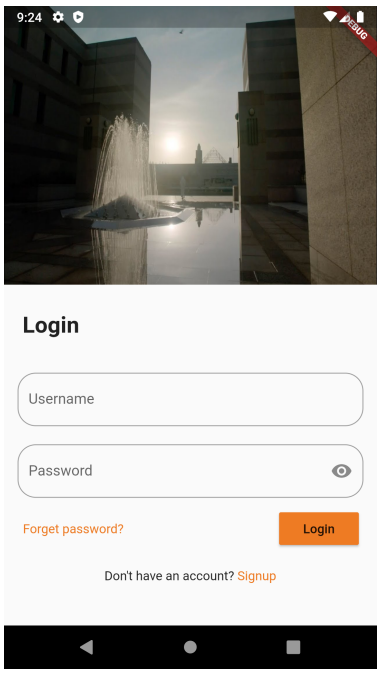

Figure 11: Screenshot for the login page.

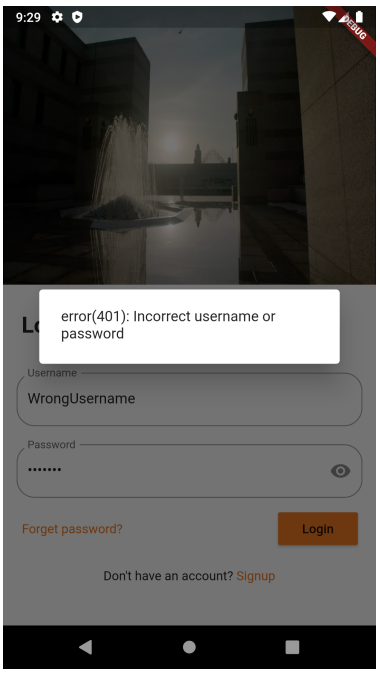

Figure 12: Screenshot for a wrong username input in login page.

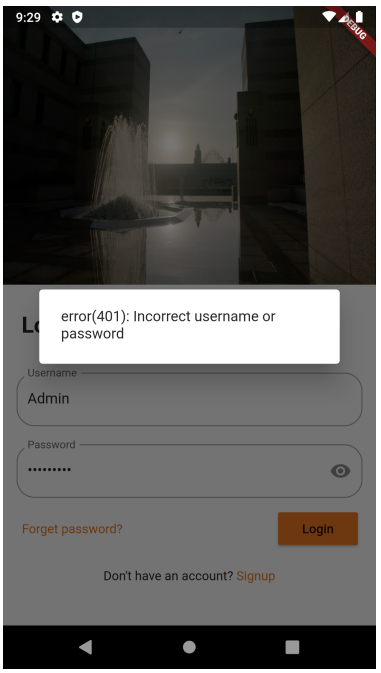

Figure 13: Screenshot for a wrong password input in login page.

When starting the program before making the login page the program will check from the local storage (see section 3.5) whether there is an existing user information or not; if there is user information the user will be immediately redirected to the App in case of a regular user or the Admin app in case of an admin user. However, if there is no information about the user, it will not direct him out of the login page, and it will wait for the user inputs (Username and Password). Then, the information will be sent to the backend using http for verifying whither he is a registered user or not; if he is a registered user he will be directed to the App in case of a regular user or the Admin app in case of an admin app. However, if he is not a registered user or the password is wrong, he will get a massage of what went wrong and will not be directed anywhere as shown in figures 12 and 13.

3.2 Register Page

This page is where the user should register to make a new account, which accepts three inputs which is: Username, Password, Email. Also, it has a text button to return the user to the login page if he already has an account. See figure 14.

When writing the register information there are some input validation for each input which are as following:

Username: must be unused, no special characters and between 5 – 30 characters.

Password: must include uppercase letter, lowercase letter, number and between 8 – 25 characters.

Email: must be unused and in this shape ”username@Example.com“

If one of the inputs has a mistake a massage will appear with the error detail, in figure 15, 16 you can see a clear example of a wrong registration. After checking the input validations, the information will be sent to the backend using http to make a new user with this information.

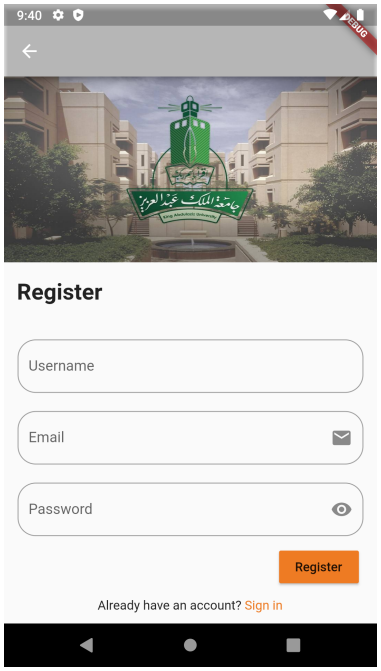

Figure 14: Screenshot for the the register page empty.

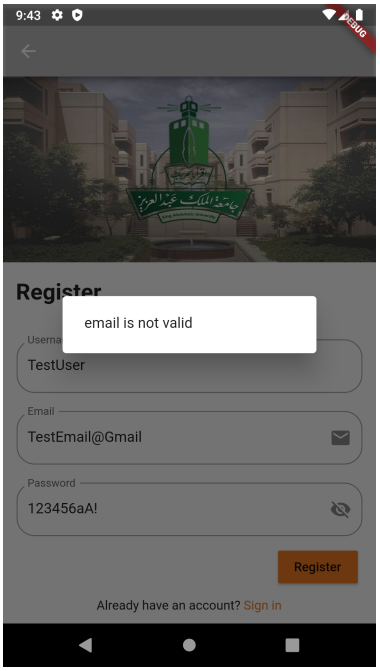

Figure 15: Screenshot for an example of wrong email format in registration.

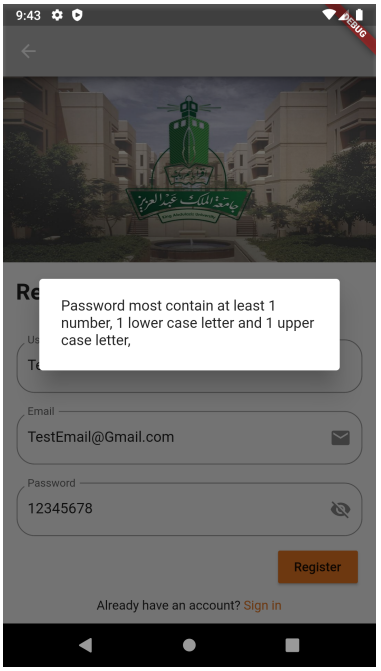

Figure 16: Screenshot for an example of wrong password format in registration.

3.3 The App

This is the main app in the program which have all the user information and the accessed doors list information on. The app has three main parts which are: the side menu, the access doors list, and the Bluetooth page (when there is a problem connecting to the Wi-Fi network). When the user get navigated to the app, all his app information will be updated from the backend using http and will be saved in the local storage every second to catch any change in the data received such as change in the access list and change in a sensor in the access list.

3.3.1 Side Menu

The side menu has three main buttons on it: Profile, about and logout as shown in figure 17:

When the user presses the profile button the user will be navigated to the profile page where he can see his information like email, authorized devices as shown in figure 18. Also, When he presses the profile button the user will be navigated to the about page, where he will find a massage about this project as shown in figure 19. Finally, if the user presses logout, he will be directed to the login page with all his information saved on the local storage being deleted.

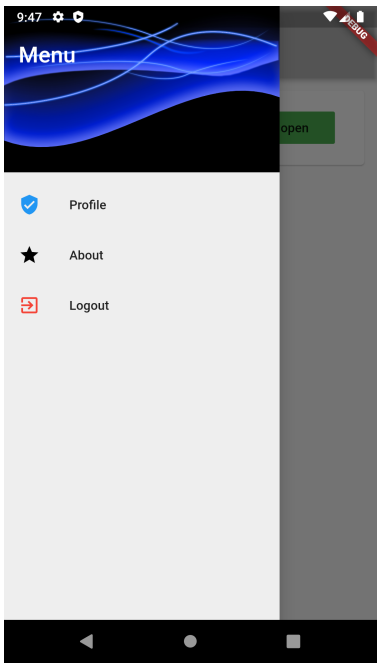

Figure 17: Screenshot of the menu of a regular user.

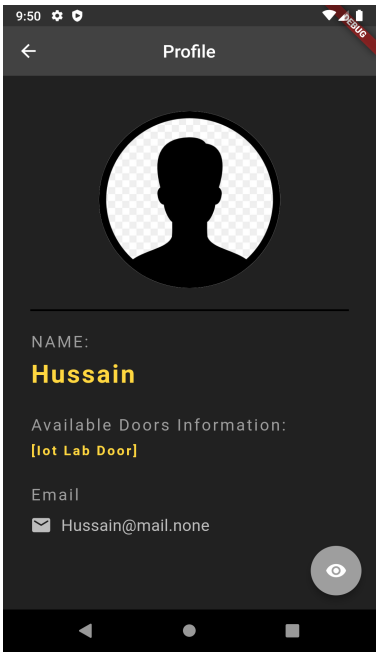

Figure 18: Screenshot of the user's profile in menu.

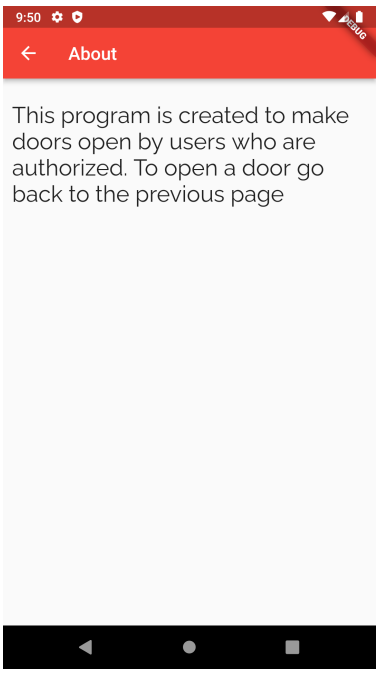

Figure 19: Screenshot of the about button in menu.

3.3.2 Access doors list

This list is first thing user sees when directed to the App in figure 21 you can see an example of having access to one door. Each door on the list will show the user the door's description and if the door is close there will be three number pickers for the hours, minutes, seconds respectively to give the duration that the user want the door to be opened for and there is an open button that if he pressed it, the door will be open, and the user will be directed to the room information page. however, if the door is already opened there will not be a number pickers and the button will be for closing the door. If the user pressed in any door on the list, he will be directed to the "room information page" which has all information about the room which are: mac address, status, Humidity, temperature, smoke sensor and number of people in the room as shown in figure 20.

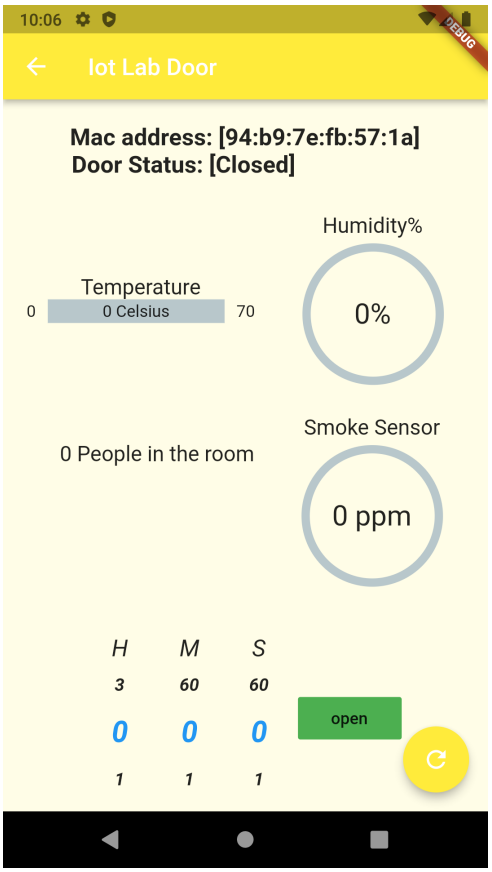

Figure 20: Screenshot of the room information after pressing on a door in the access list

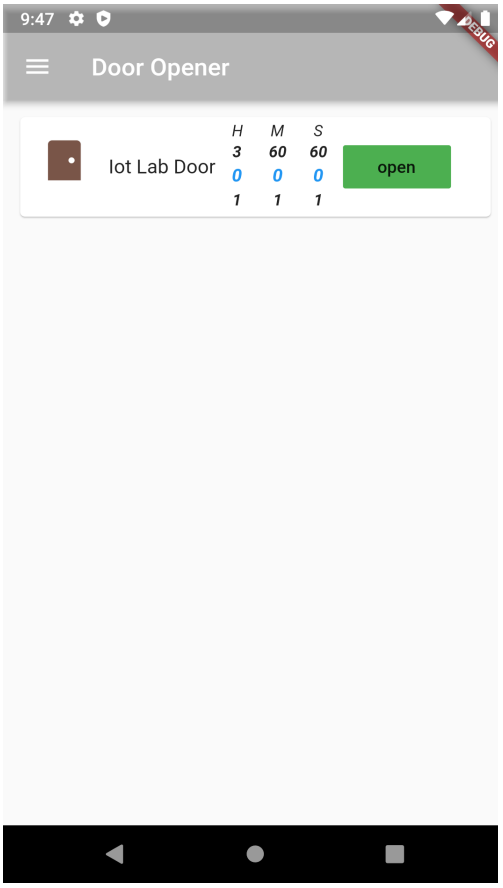

Figure 21: Screenshot of a user that have one door in his access list

3.3.3 Bluetooth connecting page

This page can't be reached unless there was a problem opening the door or close it using Wi-Fi. So, when the user tries to open the door with his Wi-Fi being closed, he will be navigated to a new page with all paired devices on as shown in figure H30; if the user has already paired with the device, he will see the devise and a connect button close to it. However, if the user hasn't paired with the device yet he won't see the device and will have to go to Bluetooth settings and pair with his wanted device. After that, the user will press at "connect" button and will be navigated to the "open via Bluetooth page". On the open via Bluetooth page the user will find a button to open or close the door, and a chat for other commands. See figure H11 for further explanations.

Figure H30: Screenshot of the bluetooth paired devices.	Figure H11.a: Screenshot after pressing connect in bluetooth paired devices and not yet connected.	Figure H11.b: Screenshot after connecting in one of bluetooth paired devices.

3.4 Admin app

The Admin app is where the admin will be navigated instead of the App. The admin app is similar to the App but with extra features in the menu, see figure H12.

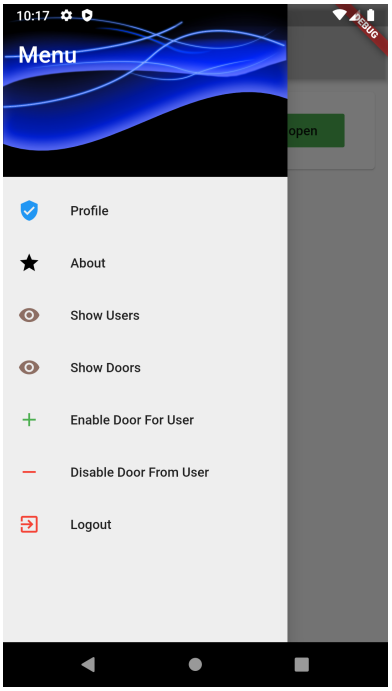

Figure 22: Screenshot of the menu of the admin app.

First, a button in the menu named “show users", which will show the admin all users registered, and their information see figure H13. Moreover, when the admin press on any user he will be navigated to a page that has his access logs, which has all the user commands with the exact date and time organized in a list see figure H14.

Second, a button in the menu named “show doors”, which will show the admin all the doors available for the users to open with its information, see figure H15. Also, when a door in the list pressed the admin will be navigated to the “room information page” which has all the information about the room see figure H6.

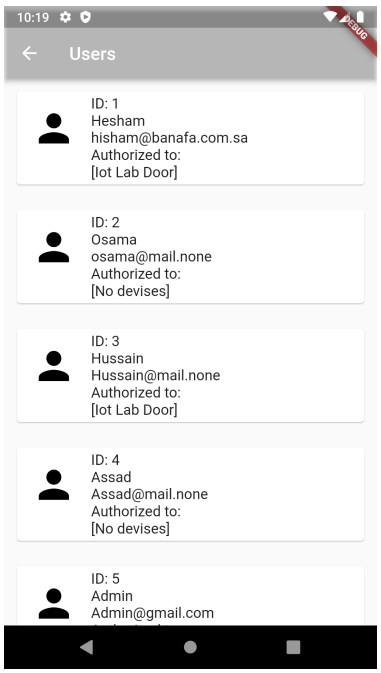

Figure 23: Screenshot of the "show users" page on Admin app menu.

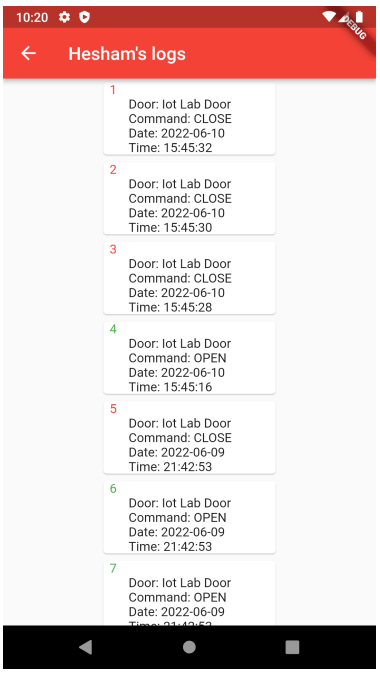

Figure 24: Screenshot of the "access logs" page of the user "Hesham".

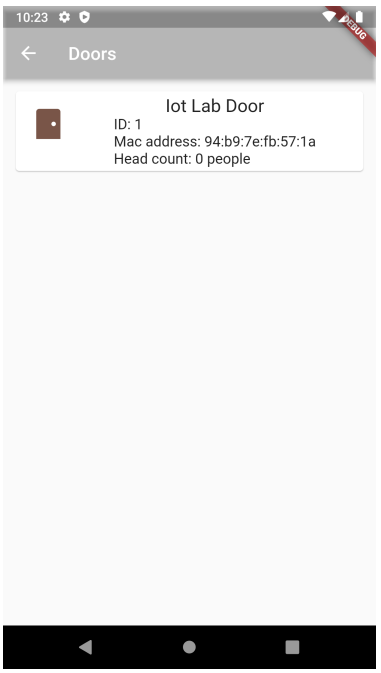

Figure 25: Screenshot of the "show doors" page on Admin app menu.

Third, a button in the menu named "Enable door for user", which will navigate the admin to a page called "Authorize user" which expects two inputs from the user which are: User ID and the doors ID. So, when the admin modifies the id of a specific user with and id of specific door, the user will be Authorized, see figure H100.

Forth and finally, a button in the menu named "Disable door from user", which will navigate the admin to a page called "Unauthorize user" which will preform the same thing that "Authorize user" dose except it will remove the authority from the user to the device, see figure H16.

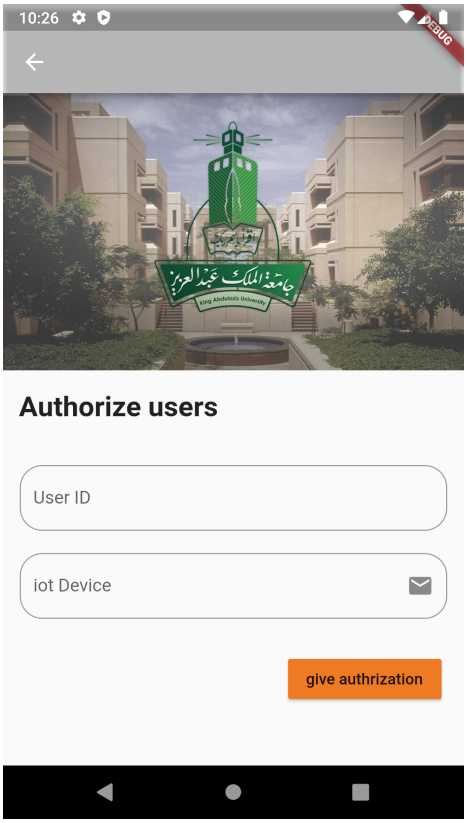

Figure 26: Screen shot of "Authorize user" page on Admin app menu.

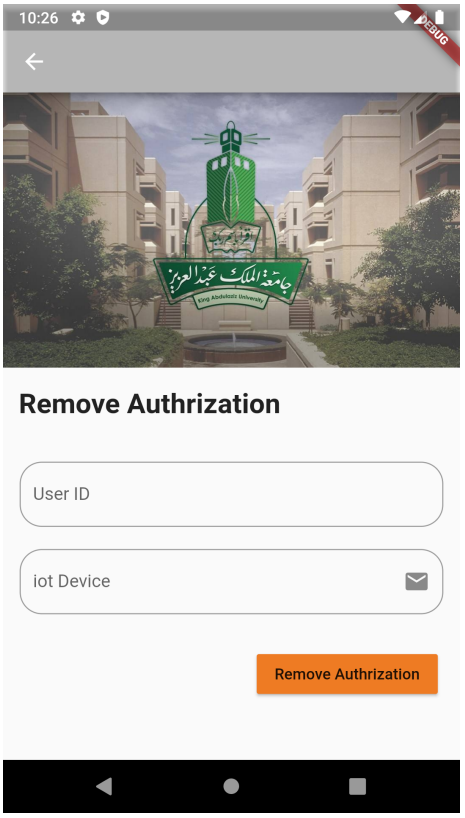

Figure 27: Screen shot of "Unauthorize user" page on Admin app menu.

3.5 Local storage (shared_preference package)

The shared_preference is a flutter package that saves the data in storage and enable the user to read it or change it or even delete it later. Which helps keeping the data if the Wi-Fi is down and no further calls to the backend accepted. The values that you can save in the storage must be String, int, double, bool or list.

3.5.1 Saving data

When saving the data in the storage, it will be shaped in the shape of Map<Key, value>, the key is what the variable is going to be called and the value is what value is the one that should be saved. For example, when the user gets token from the backend it will be saved in the program to use it multiple time across the application see figures H17, H18.

```dart
static saveString(String key,String value) async {
  final prefs = await SharedPreferences.getInstance();
  prefs.setString(key, value);
  print("The key: $key, $value saved");
}
static saveBool(String key,bool value) async {
  final prefs = await SharedPreferences.getInstance();
  prefs.setBool(key, value);
  print("The key: $key, $value saved");
}
static saveInt(String key,int value) async {
  final prefs = await SharedPreferences.getInstance();
  prefs.setInt(key, value);
  print("The key: $key, $value saved");
}
```

Figure H17: Code of the static methods that have been used to save data.

```dart
for(int i = 0;i!=doors.length;i++){
  print("doorBluetooth_mac$i");
  Util.saveInt("doorsLength", doors.length);
  Util.saveString("doorDescription$i",doors[i].description);
  Util.saveString("doorBluetooth_mac$i",doors[i].bluetooth_mac);
  Util.saveInt("doorId$i",doors[i].id);
  Util.saveInt("doorHumidity$i",doors[i].humidity);
  Util.saveInt("doorPeople$i",doors[i].people);
  Util.saveInt("doorSmoke_sensor_reading$i",doors[i].smoke_sensor_reading);
  Util.saveInt("doorTemperature$i",doors[i].temperature);
  Util.saveBool("doorState$i",doors[i].state);
}
```

Figure H18: Example code of the usage of the static saveString, saveBool, saveInt.

3.5.2 Reading data

After saving the in the shared_preference we will need to use these data which would be an easier task than saving, since you will need to use the key of the variable that we saved only see figures H19, H20.

```dart
static Future<String> readString(String read) async {
   final prefs = await SharedPreferences.getInstance();
   var thing = prefs.getString(read);
   thing ??= "";
   return thing;
}
static Future<int> readInt(String read) async {
  final prefs = await SharedPreferences.getInstance();
  var thing = prefs.getInt(read);
  thing ??= 0;
  print("$read: $thing");
  return thing;
}
static Future<bool> readBool(String read) async {
  final prefs = await SharedPreferences.getInstance();
  var thing = prefs.getBool(read);
  thing ??= false;
  return thing;
}
```

Figure H19: Code of the static methods that have been used to read data.

```dart
readNoInternet()async{
  int length = await Util.readInt('doorsLength');
  doors = List.filled(length, null);
  for(int i = 0; i !=length;i++){
    doors[i] = Door(
        await Util.readString('doorBluetooth_mac$i'),
        await Util.readString('doorDescription$i'),
        await Util.readInt('doorId$i'),
        await Util.readInt('doorHumidity$i'),
        await Util.readInt('doorPeople$i'),
        await Util.readInt('doorSmoke_sensor_reading$i'),
        await Util.readInt('doorTemperature$i'),
        await Util.readBool('doorState$i'));

  }

}
```

Figure H20: Example code of the usage of the static readString, readBool, readInt.

3.6 http (flutter_http package)

We used http protocol to communicate between the backend and frontend. The flutter_http package is responsible for all type of communication which are: get and post data to the backend. There are another important package in flutter which is “convert” package which convert the massages between the frontend and backend to Json data which is easier to deal with. In figures H21, H22, you can see clear example for posting and gettingdata with the backend.

Figure H21: Example of the usage of HTTP protocol getMethod to fetch user.

Figure H22: Example of the usage of HTTP protocol postMethod to open the door.

3.7 serial Bluetooth communicating (flutter_bluetooth_serial)

There are too many Bluetooth packages in flutter, but we used this package because it's up-to-date and it preform what we need exactly which is transforming data serially via Bluetooth. First, we made "Bluetooth main page" which will check for connected devices; if there is connected device it will navigate the user to the page that has the chat and the open command button. If he is not connected yet it will give the user, the list of paired devices and wait him to connect. After that, when the user reaches the chat page with the button, if the user typed any massage in the chat and send it, this massage will reach the door serially. However, if the user just pressed the open button, his username and password will be read from the local storage and will be sent to the device with a space in between them for the device to understand where the username is and where is the password in the form of "Username password". The device will understand the massage and will open the door or close it depending on the door status.

4. Door controller

Door controller is the part responsible for locking and unlocking the door at the request of the user and the powers given by the administrator. To function correctly, this part communicates directly with two other major parts, namely the backend and the mobile application. Hence, in normal operating mode, the door controller communicates with the backend to receive operational commands. These commands consist of unlocking the door with the duration time to open it, forced closing of the door lock even if the duration time has not expired, and the data of users who are allowed to open the door. However, In the case that the controller experiences an irregular operating circumstance, it connects directly with the user (mobile application) to open the door for him if the user has the authorization to do so. Moreover, the door lock can be opened manually by a button located inside the room.

There are two main sections in this part which are controller programming and circuit schematics.

4.1 Controller Programming

The ESP32 was employed as a door controller, allowing for wireless control of the door lock. The ESP32 has the ability to communicate wirelessly using Bluetooth and Wi-Fi. Additionally, it is free and open source as well as supported by several libraries that will make it easier to construct a complex system. As well as the PlatformIO with the Arduino framework was chosen for programming.

4.1.1 controller's program overview

In the beginning, the controller will try to connect to the Wi-Fi, and if it is unable to connect, the user will be able to connect directly via Bluetooth. Conversely, the controller will turn off the Bluetooth connection once it has been connected to the Wi-Fi. It will then try to connect to the server in order to send data and receive commands. The commands are to open the door, the duration of the opening, and to force the door to close, as well as a counter that represents the version of the data of the users. Whenever the controller cannot connect to the server, then the user will be able to directly connect to the controller via Bluetooth too. Figure 34 shows the flowchart of the ESP32 control.

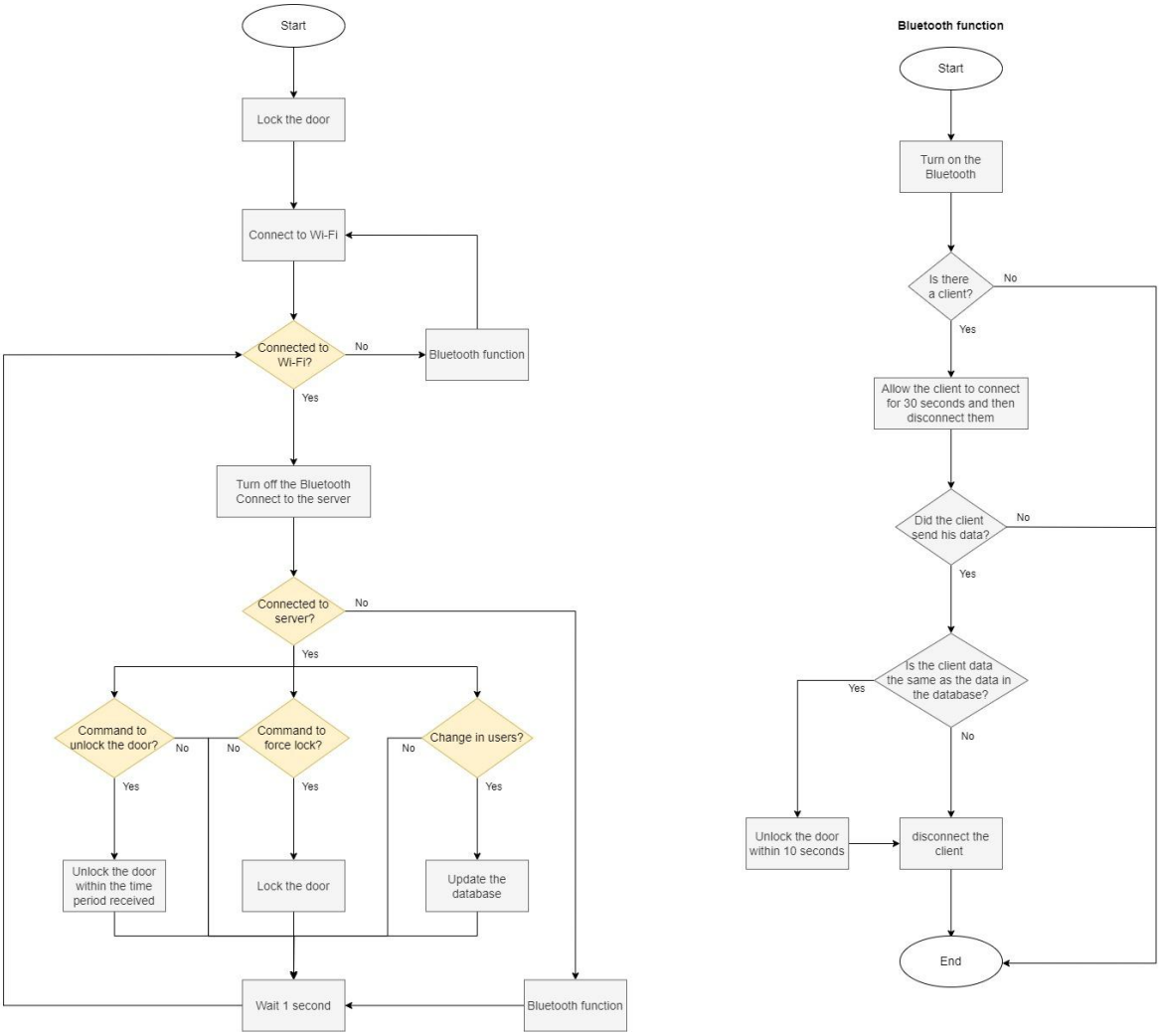

Figure 34 Controller's Flowchart

4.1.2 Communication with the server

The controller communicates with the server via HTTPS. It will post a static token and Bluetooth mac address for authentication with each request, as well as the state of the door (lock or unlock). On the other side, as a response to the request, the controller will get commands and data. The controller typically sends three requests for information to the server. The first is for retrieving commands from the server; this is the most often requested request, and it is posted on a regular basis. The other two requests, one for acquiring usernames and the other for getting passwords, are used to update the database. The data provided during the communication will be of the string data type, with the string encapsulated as a JSON object for post and get commands.

4.1.3 Bluetooth function

The Bluetooth feature is only used when conditions are abnormal and for opening the door for 10 seconds. In order to open the door, the user needs to provide a username and password. If both match the data in the database, the door will open, otherwise, the user will be notified the data sent was incorrect. Furthermore, the connection is limited in order to prevent the device from being suspended for one user. The connection will be calculated for a maximum of 30 seconds after the customer connects to the device; otherwise, the connection will be disconnected, and once the user sends his data and the decision has been made whether to open the door or inform, the connection will be disconnected as well.

4.1.4 Database

The database saves the data of the users who are permitted to access the door. Basically, the ESP32 has a storage flash memory that has a size of 4MB and can be accessed using SPIFFS (Serial Peripheral Interface Flash File System). In our implementation, there are two files one for usernames and the other for passwords. Also, there is a local counter in the program. So that, if the version counter received from the server differs from the local counter, the controller will obtain users from the server and write them to the username file, then obtain passwords from the server and write them to the password file, and finally update the counter.

4.1.5 Timers

By using the timer, the controller will be able to execute the program normally, and after a certain period of time, it will interrupt the main program to execute some commands and then return to where it stopped. We employed the timer for two main objectives in our implementation, the first of which was to open the door. Each time a command to open the door is issued, the door will be opened, and the timer will begin for the specified amount of time. When the timer has run out, the instruction for the timer will be executed, which will lock the door and stop the timer. The timer was also utilized to determine the status of the door, so it is only open when the timer is running. The timer will also be used in the Bluetooth function. When the user connects to Bluetooth, a 30-second timer is started, and if the timer expires while the user is still, the timer's instruction, which is to disconnect the user and stop the timer, is executed.

4.1.6 Push Button

If a person is inside the room, he does not need the application or the authority to open the door, and the exit should be as simple as possible. A push-button was installed inside the room that will open the door lock for 10 seconds, and this is the most common and standard design in most door lock systems.

4.2 Circuit Schematics

The two most important components in our circuit are the electromagnetic lock and the ESP32. However, the electromagnetic lock operates on 12V, whereas the ESP32 operates on 5V. As shown in figure 35, we designed two separate circuits: one for the electromagnetic lock and another for the ESP32. The first circuit will be connected to a normal open relay, and the ESP32 will send a signal to the relay to switch on/off the electromagnetic lock circuit. Moreover, A 12V/3A adapter will power the circuit, and a DC voltage regulator will step down the voltage to 5 volts to power the controller and switches.

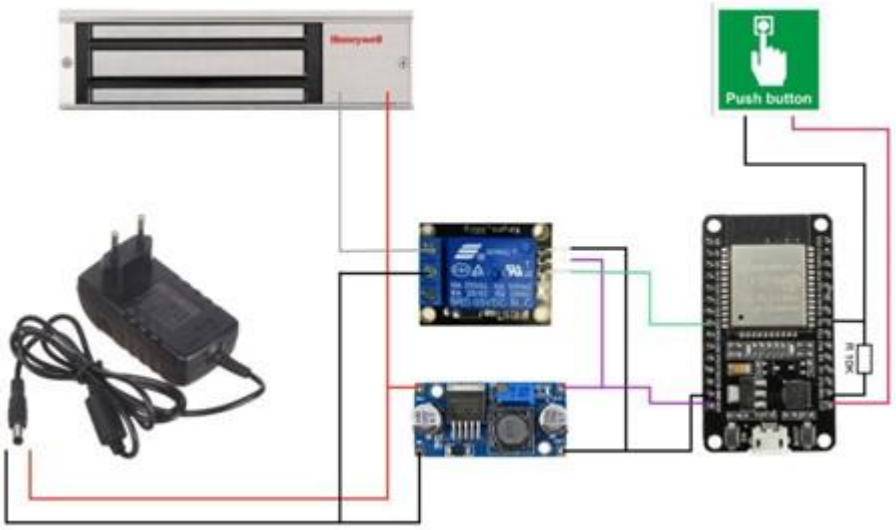

Figure 35: Circuit Schematics

5. Monitor

This part of this project is to monitor the room and collect data from the sensor that is used in the room and send it later to the main server to display it later in the mobile application, so the main system for the monitor system is ESP32 microcontroller to take control over the sensor, later for monitoring the room temperate to get the status for the temperature, DHT11 sensor is a good choice for this job because it has multitasked monitoring which is sensing the temperature and the humidity in the room and MQ-5 sensor for smoke and gas detecting. The last sensor is the ultrasonic sensor to detect the count of visitors to the room. So this section will demonstrate the Monitor system and disrobe each of the components that were used for this system, later the communication process between the Backend and monitor system, and the role of this system for the overall project.

The choice of the sensors was based on the requirements of the project to meet, for example, choosing cheap and power-efficient sensors that can get the job done so ESP32 is a good choice for this project it can do a multitasks and can connect to a Wi-Fi and Bluetooth easily and send the data to the main server.

5.1 Monitoring Circuit

This part of the project will be responsible for monitoring the temperature and humidity, Gas values, and counting the visitors inside a room.

choice for this project it can do a multitasks and can connect to a Wi-Fi and Bluetooth easily and send the data to the main server.

5.1.1 ESP32 DEVKIT DOIT

The main board that is used in the Monitor is ESP32 to take control over all the monitor sensors that are used in the circuit in this section will describe the ESP32 DEVKIT DOIT in specific and describe it.

Figure 36: ESP32 DEVKIT DOIT

Table 9: ESP32 DEVKIT DOIT Specifications

Wi-Fi	2.4 GHz up to 150 Mbits/s
Bluetooth	BLE (Bluetooth Low Energy) and legacy Bluetooth
Architecture	32 bits
Clock frequency	Up to 240 MHz
RAM	512 KB
Pins	30 or 36 (depends on the model)
Peripherals	Capacitive touch, ADC (analog to digital converter), DAC (digital to analog converter), I2C (Inter-Integrated Circuit), UART (universal asynchronous receiver/transmitter), CAN 2.0 (Controller Area Network), SPI (Serial Peripheral Interface), I2S (Integrated Inter-IC Sound), RMII (Reduced Media-Independent Interface), PWM (pulse width modulation), and more.
Input voltage	5V

5.1.2 Temperature and Humidity Monitoring

Monitoring the room temperature is very important for this project in case of any fire happened the temp sensor will detect that by measuring the change of the temperature inside the room and send it to the back end to trigger the emergency plan and open the door so for this purpose, the optimal solution is to choose power-efficient sensor with low cost to get this task do so later DHT11 sensor was chosen for this task.

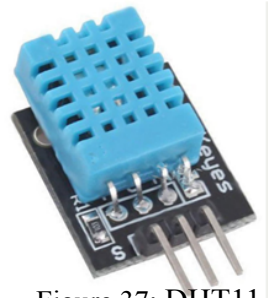

Figure 37: DHT11 Sensor

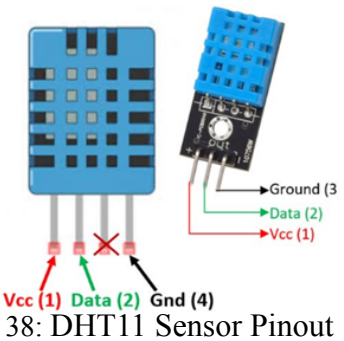

Figure 38: DHT11 Sensor Pinout

Table 10: DHT11 Pinout Configuration

No:	Pin Name	Description
1	Vcc	Power supply 3.5V to 5.5V
2	Data	Outputs both Temperature and Humidity through serial Data
3	Ground	Connected to the ground of the circuit

Table 11: ESP32 DEVKIT DOIT Specifications

Wi-Fi	2.4 GHz up to 150 Mbits/s
Bluetooth	BLE (Bluetooth Low Energy) and legacy Bluetooth
Architecture	32 bits
Clock frequency	Up to 240 MHz
RAM	512 KB
Pins	30 or 36 (depends on the model)
Peripherals	Capacitive touch, ADC (analog to digital converter), DAC (digital to analog converter), I2C (Inter-Integrated Circuit), UART (universal asynchronous receiver/transmitter), CAN 2.0 (Controller Area Network), SPI (Serial Peripheral Interface), I2S (Integrated Inter-IC Sound), RMII (Reduced Media-Independent Interface), PWM (pulse width modulation), and more.
Input voltage	5V

6 Results & Integration

<How it is a challenge to integrate all the subsystems, Show operation (open, close) diagram, photo from lab, door lock, sensors, applications, backend api>

In this project, integrating the different systems is a challenge of interoperability. The subsystems Monitor, Door controller, and User application interact directly with the backend, except in the special case where the backend is not reachable, the user application and door controller communicate directly via bluetooth. For this reason, a command to open a certain door start from the user application with the following information.

- Username
- Door identifier
- Open Time
- Access Token

This amount of information is sufficient to allow the backend to make a decision to accept or reject the request; on the basis of the user's authentication token and authorization list kept in the database. In the case of rejection, an HTTP response of 401 is returned and the reasons for rejection are omitted to reduce information leakage. In the case of accepting the user token, the association table linking users with doors is quarried by the door identifier. If the quarry yields an empty list of database objects, the request is rejected on the basis of unauthorization. The next step in case of valid authorization, is to set the “open_command” boolean column of the IoT door to True, and an access log entry is constructed and recorded in the “access_log” table. Since in out design the door is polling the backend on an interval of one second, the responses of which include the “open_command” boolean flag. Whenever the Door polls a True flag, it opens the door for an amount also provided in the polling response. The backend database clears the “open_commad” flag and assumes the door is open since the last poll occurred. In the next poll from the door, the state is reported as True by the door, then the backend updates the Column “state” associated with the door. At the same time the user application is reading the “state” flag to update the interface to allow for door closure, during the time it remains open. Assume the user requests to open a door for one hour, then desires to close in the last 10 minutes. The user application allows sending a request to the API endpoint “/users/close” with the same information used in “/users/open”. However, the database interprets this as a force_close command. Thus, another column, named “force_close” is needed to flag a close command to the door. On the next cycle of polling, the door reads the flag and cancels the internal timers and resets the default state of closed, and the database clears the flag on the first read of the flag the same way it is done with the “open_command”.

This process of integrating systems benefits greatly from the standardized JSON format for State transfer between the systems. The following are the JSON request response schemas used to enable this integration.

User open request HTTP body

```
{
  "username": "string",
  "bluetooth_mac": "string",
  "time_seconds": 0
}
```

User close request HTTP body

```
{
  "username": "string",
  "bluetooth_mac": "string"
}
```

Door Polling request body

```
{
  "bluetooth_mac": "string",
  "state": 0,
  "token": "string"
}
```

Door polling response

```
{
  "open_command": true,
  "acces_list_counter": 0,
  "time_seconds": 0,
  "force_close": true,
  "state": true
}
```

The value “access_list_counter” is used to notify the door of a change in the users authorized to access the room. Whenever the endpoints “/admin/users/allowdevice/id” or “/admin/users/disallowdevice/id” are used, the counter is incremented. When the door read a new number, it fetches the updated list of usernames to cache locally using “/iotdevice/door/users”, to use them in the case of an unreachable backend.

Fortunately, FastAPI uses OpenAPI to generate an HTML web page and template to view and use the backend functions, and shows the proper methods and schemas to use when doing so. The page can be opened in a web browser using the API hostname at the endpoint "/docs". The page also allows authentication, by requesting a token endpoint (defined in source code), and stores the token in the browser cache.

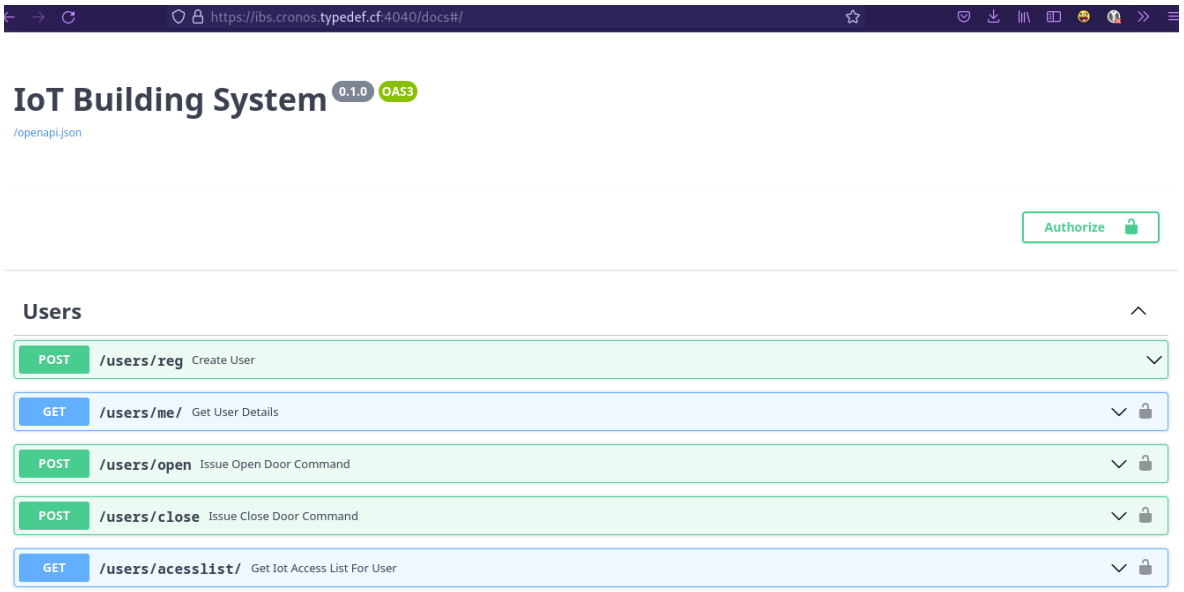

Figure 39: API documentation page generated using OpenAPI

7 Future Work

The project final results satisfied the initial expectations. However, there is always room for improvement. In terms of scalability, the polling method may overload the backend if for example the same architecture was used with 500 devices. This will result in 500 requests per second, that use up connectivity resources, CPU time, and IO time on the server. To combat this, multiple solutions can be proposed. First, a more capable server can enable this theoretical amount of devices, having other factors such as the backing network capable of handling the traffic. A second solution is to develop a layered approach. This project can be extended to be only a subsystem of a larger system, controlling only a subset of the devices. A different layer is developed to oversee these subsystems. In this approach, the user applications communicate with the upper layer, which then communicates with the associated underlying system. This approach can produce increased complexity without really solving the high traffic problem. A third solution can be the use of web sockets, or keeping http connections alive. The result will be as if the IoT devices are listening on ephemeral ports. An example of this would be the IoT door and monitors connect to the backend server, and keep the connection open (listing). Thus, whenever the user requests to open the door, and the backend deems it valid, the backend sends the request to the door. In this case the polling is eliminated completely.

Another improvement is to let the backend track the IoT devices and mark them as online or offline, based on the expected polling rate. This information can help the user application to make an informed decision to use bluetooth if the door can not reach the backend, but the user can. Basically a watchdog timer.

During early development of the user interface, the IoT devices were not online at some times due to external reasons. Manual intervention was used to emulate the IoT device polling and behavior. Thus, IoT device emulators can be developed using the basic request-response model to allow robust testing and faster development of the user interface. This was done partially but not completed in the source code directories of the backend.

Since we discussed the reasons for separating the backend and frontend, it is encouraged to develop another frontend to expose admin controls if this project was to be implemented as a business solution. A well versed web developer can easily implement the API interface and produce a suitable interface for system management. This was also partially done in the backend source code git tree on branch “frontend-tests” with a basic login page that requests a token and stores it in the browser. However, it is not usable.

8 Conclusion

In this project, the team set out to build a comprehensive system that allows building managers and visitors/workers to view and control door access, via developing an IoT system, where Doors and Rooms become things accessible via the network. Furthermore, the system is equipped with a mechanism to respond to emergencies and is intended to decrease the evacuation time during an emergency. Building administrators are able to control facility access remotely, where the enrolling process is quick.

Fortunately, the goals of this project were met to a satisfactory level for the team. The door locking and unlocking system is working with a high degree of reliability. The maximum door response time can be calculated depending on the deployment environment with the following basic formula.

$$\textit{Response Time} = \textit{Door Polling Interval} + \textit{Door to Backend Network Latency} + \textit{Phone to Backend Latency} + \textit{Backend Response Latency (IO)} \mp \textit{Network Jitter}$$

The monitoring subsystem is reporting information about the associated room, and the backend stores the information for later use, for both sensor readings and access log. User interface is intuitive and simple to use, and offers all of the core functionality of the backend. The backend was developed to be universally available to any interface that implements the API, with great care to security practices.

References

[1] P. P. Ray, “A survey on Internet of Things architectures”, Journal of King Saud University – Computer and Information Sciences

[2] S. Villamil, C. Hernandez, and G. Tarazona, “An overview of internet of things,” *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 18, no. 5, p. 2320, 2020.

[3] S. Li, L. D. Xu, and S. Zhao, “The internet of things: A survey,” *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2014.

[4] S. Wasly, “INTERNET OF THINGS & ITS APPLICATIONS.” King Abdulaziz University, Jeddah, 26-Jan-2022.

[5] “Zigbee Protocol - an overview,” *ScienceDirect* . [Online]. Available: https://www.sciencedirect.com/topics/engineering/zigbee-protocol. [Accessed: 10-Jun-2022].

[6] The Internet Society, “ Hypertext Transfer Protocol -- HTTP/1.1,” Jun-1999.

[7] “What is a rest api?,” *Red Hat - We make open source technologies for the enterprise*. [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api. [Accessed: 11-Jun-2022].

[8] E. R. Fielding, “RFC 7231 - hypertext transfer protocol (HTTP/1.1): Semantics and content,” *Internet Engineering Task Force*, Jun-2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7231#section-6. [Accessed: 11-Jun-2022].