Palestine Polytechnic University

College Of Information Technology and Computer Systems Engineering

Department Of Computer Systems Engineering

IoT Management and Simplified Device Integration for Smart Homes

Mohammad Faqusa

Supervised by:

Dr. Hashem Tamimi

2024-2025

Acknowledgment

الحمد لله رب العالمين، الذي منحنا القوة والصبر والعلم لإتمام هذا المشروع

أود أن أعبر عن خالص شكري لمشرف مشروع التخرج، الدكتور هاشم التميمي على دعمه المستمر وتشجيعه لنا لإنجاز هذا المشروع

ونشكر أعضاء هيئة التدريس في كلية تكنولوجيا المعلومات وهندسة الحاسوب على بناء مسيرتنا الأكاديمية، التي اختتمت مع أبرز البنّائين وهم السادة المناقشين: الدكتور أيمن وزوز والأستاذ وائل التكروري.

لعائلتي، مصدر المحبة والدعم الذي لا يلين.

وأخيرا، الى كل من ساعدنا وأرشدنا وحفزنا على طول الطريق، تقبلوا منا خالص التقدير، لقد كان دعمكم قيمة لنا في جعل هذا المشروع ممكنا

Abstract

Making smart-home IoT systems easier to install is important if we want electricians and other non-programmers to use them confidently. In this project, we present a simplified IoT integration platform that uses artificial intelligence to generate the code for devices and guide users with clear wiring instructions for their chosen sensors and actuators. The platform provides an intuitive interface for monitoring data in real time, controlling devices, and creating automation rules. Alongside this, a companion application helps with registering devices, installing the generated firmware, and showing step-by-step connection guidance. Our evaluation shows that users can set up multi-sensor nodes, control actuators, and define automation rules in just a few minutes, all without needing to write any code. By lowering both the cost and the technical difficulty, this solution supports the wider adoption of smart-home technologies.

Keywords:

Smart Home · Internet of Things (IoT) · Artificial Intelligence · Automation · Device Integration · Code Generation · Real-Time Monitoring · Actuator Control · Accessibility

Contents

List of Figures

List of Tables

Chapter 1

1) Introduction

1.1 Preface

The Internet of Things (IoT) links common devices to the internet, enabling us to manage them, observe activities, and make decisions based on data. It has become an essential element of contemporary automation—from intelligent homes to factory systems—enabling environments to react to individuals and their environment.

However, initiating IoT can still be challenging. Setting up and installing typically require knowledge of hardware and programming abilities. That technical barrier impedes adoption for numerous hands-on specialists—such as technicians and electricians—who can manage the physical tasks but might lack experience in embedded programming [2], [5].

This gap is the reason for the existence of this project. The goal is to create a system that enables users to deploy and operate IoT systems effortlessly, even without programming knowledge. The platform introduced lowers the technical entry barriers by combining automation, real-time communication, and code generation, thus making smart home integration attainable and scalable [1], [3], [4].

1.2 Problem Statement

Home automation systems are being used more and more for their convenience, efficiency, and security. Installation, configuration, and maintenance of the system typically, though, demand specialized expertise—covering hardware installation to network configuration, device activation, and continuous troubleshooting—that most inhabitants and even on-site engineers do not have. This makes it inaccessible, more expensive, and causes greater reliance on experts.

In response, certain turnkey kits and ready-to-use, "plug-and-play" devices have since become available in the market. They facilitate easier initial setup work but tend to be one-size-fits-all: superficial integration with adjacent infrastructure, minimal customization, closed

systems, and vendor lock-in (often with recurring subscription fees). They thus rarely develop to accommodate unique use cases, shifting requirements, or local/offline control necessities.

In response to this, our suggestion offers a unified platform that streamlines the entire life cycle of intelligent connected devices. It directs installation through an intuitive interface, streamlines setup, simplifies software updates, and offers centralized monitoring, management, and rule-based automation. The outcome is a scalable, affordable, and accessible approach to smart home integration for a broad audience

1.3 Aims and objectives.

In this project, we are suggesting a system that will make it easier to deploy and set up IoT devices for those who do not possess technical skills. The following should be accomplished in an attempt to achieve this goal:

a) Create and roll out an easy-to-use interface with which no coding knowledge is required to register devices and choose the components they are attached to.
b) Use a desktop application for automatic embedded firmware creation and installation based on user selection.
c) Help users with the physical installation process through offering step-by-step, device-specific hardware wiring instructions.

We want to build a platform that provides a homogeneous interface for real-time monitoring and controlling the registered devices. The following goals must be accomplished in a manner to accomplish this target:

a) Install a responsive dashboard by which users can control actuators and monitor real-time feedback from corresponding sensors.
b) Use real-time data channels to enable low-latency communication among the user interface, remote devices, and backend system.
c) Securely store user accounts, device settings, and activity logs in a long-lasting database.

Personal automation rules between connected devices are another objective of the system. The following objectives must be accomplished in order to accomplish this function:

a) With a graphical user interface, enable users to choose input conditions, target devices, and response actions to construct automation logic.
b) Install the automation logic onto the pertinent device to be able to independently monitor conditions and initiate actions.

c) Make sure devices can reliably and in real time communicate with each other so that one can act upon when certain things are fulfilled.

1.4 Requirements

1.4.1 Functional requirements

1. Features for account management, such as user registration, login, and authorization, will be offered by the system.
2. Using a graphical user interface, the system will enable users to register IoT devices and choose connected peripherals.
3. Using the registered configuration, the system will automatically create and install device firmware.
4. The system will offer concise, text-based instructions for joining hardware parts.
5. A dashboard will be provided by the system so that users can track sensor data and manage actuators in real time.
6. Using conditional logic, the system will enable users to create and implement automation rules between devices.
7. For access control, the system will link every device to a user account.

1.4.2 Non-functional requirements

1. Usability: Users without technical or programming experience should be able to easily navigate the platform's interface.
2. Performance: Under typical operating circumstances, all control and monitoring operations must be finished in no more than two seconds.
3. Portability: The application must work with contemporary web browsers and be responsive on screens that are at least 360 pixels wide.
4. Security: To safeguard user information and guarantee safe access, the system will employ authorization and authentication procedures.

1.5 System Description

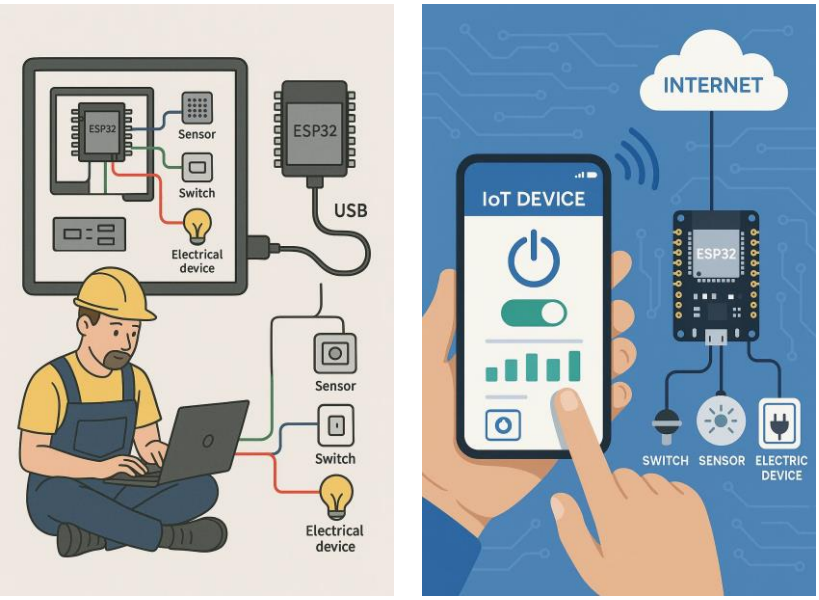
The proposed system is a stack-focused platform that utilizes AI to assist non-technical users in effortlessly integrating, installing, and monitoring IoT devices. It links a graphical user interface to distributed ESP32 microcontrollers and backend services through a modular client-server

framework available through a desktop application and a web dashboard. An AI service automatically generates the required MicroPython firmware, incorporating pin mappings and library imports, when an electrician registers a device. The application subsequently writes the firmware onto the ESP32

The pin map and peripheral list for each device are converted into understandable connection instructions by the same AI engine, which also generates step-by-step wiring guidance. Configuration metadata is linked to the user's account and kept in a central database, As shown in figure 1.1 (a).

Devices use MQTT to communicate with the server once they are online. from a single dashboard, users can set conditional automation rules, monitor status, and issue manual controls. These rules are then transmitted to the relevant ESP32, where they operate independently to initiate actions in response to real-time sensor readings, as show in figure 1.1 (B).

Through the use of AI-driven services to handle code development, wiring assistance, and network setup, the platform eliminates embedded programming and hardware knowledge, making it simple and quick for electricians to install fully functional smart-home systems.



(A) (B)

Figure 1.1: System Description

1.6 Limitations and constraints

In spite of the system's ambition to make IoT integration easier for non-technical end-users, some limitations and constraints were found during the development and current implementation stages:

1. **Local Hosting Environment**: The current version of the platform is on a local environment. Deployment on the cloud has not yet been done, and this lowers accessibility to users on the same network and impedes remote use cases.
2. **Device Memory and Processing Capacity:** The devices embedded in the system both have restricted processing capacity as well as memory. This puts a limit on the level of complexity of the automation logic that can be supported and the amount of simultaneous operation that can be performed locally on the device.
3. **Absence of Visual Hardware Connection Instructions:** Although the system gives text-based instructions for wiring, interactive or graphical schematics for peripheral connections are yet to be developed. For the uninitiated, this can be a difficulty.
4. **Physical Peripheral Compatibility Checking:** The operation relies on the user to check physically that the chosen peripherals are compatible with the device being used. Electrical and pinout constraints are not checked by default within the selection process.
5. **Single-Developer Scope:** The entire system was implemented and developed by one person. Due to time and resource limitations, this could have restricted the scope of testing, feature addition, and UI/UX refinement, although it ensured concentrated implementation.
6. **Network Conditions Affects Real-Time Performance:** Real-time control and monitoring involve communication over local or external networks. Traffic and capacity of messages can affect latency and reliability, particularly when updates are high or the automation logic is complex.
7. **Restricted User Permissions and Roles:** No fine-grained permission controls and advanced roles and shared or organizational-level access are provided in the system, which provides only basic user account functionality.

1.7 Schedule

The project schedule as shown on table 1.1

Table 1.1: Project schedule in the first and the second semester

	The first semester			The second semester			
Week	1 - 4	5 - 10	11 - 15	1 - 5	6 - 9	10 - 14	15
Selection of project Idea							
Collecting the Data							
System Design							
System Implementation							
System testing							
system operation							
Documentation							

1.8 Report outline

The six chapters constituting the framework of this report, each discusses a unique stage of the project. The theoretical foundation and underlying technologies concerned with the system are discussed in Chapter 2, followed by an exposition on related work to put our approach into context. The system design is debated in Chapter 3. The hardware and software architecture are described in detail, principal design choices are outlined, and schematic diagrams are shown. Chapter 4 deals with the implementation process and also reveals the issues faced during the development and construction of the system. Testing processes that have been used on the components of the system and their functionality are assessed in Chapter 5. Chapter 6 concludes the work by summarizing the project results, reflecting upon them, and making recommendations for the future.

Chapter 2

2) Background

2.1 Preface

The fundamental building blocks of the Simplified IoT Integration and Management system are presented in this chapter. Not going into explanations of how these pieces fit together and how the system is organized, it deals with the hardware and software components chosen for building the platform. Based on usability, performance, compatibility, and integration ease aspects, each equipment piece is selected from the range of options. Theoretical foundations of the technologies are outlined at the beginning of the chapter and followed by these chosen components explained in terms of the reasons for choosing them through structured comparisons. Literature review targeting current systems and how this project provides an easier and friendlier alternative to non-technical users is located at the end.

2.2 Theoretical background

This chapter gives a general overview of the main technologies in the system. These technologies, from hardware layers to software layers, communication protocols, and AI assistance, all support the system in allowing the deployment of IoT by non-technical users.

2.2.1 Microcontroller

The core of every Internet of Things device is the microcontroller. It serves to run firmware logic, communicate with the server, and exchange data with sensors and actuators. A good microcontroller should be programmable through high-level languages appropriate for embedded systems, have digital and analog input/output, and wireless communication[4].

2.2.2 Peripherals (Sensors and Actuators)

The peripherals are the devices that are attached to microcontrollers and enable communication with the outside world. They constitute motors, relays, display modules, RFID readers, motion

sensors, temperature sensors, and others. Hardware design must take into account the specific communication and power needs of each peripheral[5].

2.2.3 Protocol MQTT

MQTT (Message Queuing Telemetry Transport) is an publish-subscribe, low-power, and low-bandwidth messaging protocol. The devices in this system interact with the master server via MQTT. It provides real-time monitoring and control through allowing devices to subscribe and publish messages for particular topics[6].

2.2.4 Front-end and back-end web applications

The GUI offered by the frontend allows users to register devices, configure peripherals, read information, and control outputs. The backend builds and saves device configurations and also manages authentication, processing of user requests, and communication between the user interface and real devices. In order to send commands at the device level, the backend also communicates with the MQTT broker[7-10].

2.2.5 Socket Communication

To facilitate real-time interaction with no page refreshes, the system utilizes socket-based communication between the backend and the frontend. This enables the dashboard to continuously send and receive real-time data, thereby refreshing the interface with the latest readings and providing instant response to user input[11].

2.2.6 AI Integration

The system also employs an artificial intelligence API to facilitate code generation and hardware control. Upon selection by customers of specific peripherals, the chatbot provides text-based wiring diagrams and helps produce suitable embedded code. To unsuspecting customers who might not be familiar with programming or electronics, this is a relief[12].

2.2.7 JWT Verification

A JSON Web Token (JWT) is a URL-safe, compact string that passes digitally signed identity claims from client to server. The payload carries claims like userId and exp; the header indicates

the signing method; and the computed signature—a server-side secret—prevents tampering with the token. With merely the presentation of the token, the server authenticating and authorizing requests allows stateless, low-latency security, thereby avoiding additional database lookups. Upon successful registration or login, the backend within this project issues a JWT saved by the browser inside an HTTP-only cookie and appended to all authenticated REST or socket IO requests to secure device registration, real-time monitoring, control, and automation streams[13].

2.2.8 Electron Desktop Framework

By combining the Chromium rendering engine with the Node.js, electron is an open-source runtime that enables developers to package standard web—HTML, CSS, and JavaScript—to build cross-platform desktop applications for Windows, macOS, and Linux. A one or more Chromium "renderer" processes render the user interface; every Electron application has a "main" process (Node.js) able to access the local file system, USB devices, and operating-system APIs.

Electron in this project wraps the same web user interface experienced in the browser but with Node.js modules that detect a connected ESP32 through USB and call esptool.py to flash the AI-designed firmware. From a familiar desktop interface, this method makes electricians able to execute code installation and device registration from inside and share the codebase with the equivalent online version[14].

2.2.9 Python Method Wrapper

In Python, method wrapper is an abstraction to envelop callable objects such as functions or methods in order to implement custom behaviors upon the time of calling. In this project, method wrappers are used in order to offer dynamic and generic method invocation to hardware peripherals. Instead of a traditional function call such as peripherals.accelerometer.read(), the system uses a more generic and uniform syntax: peripherals["accelerometer"]["read"][[]]. It enables method invocations to be called in a generic manner:
peripherals[peripheral_name][peripheral_method][peripheral_parameters].
This type of structure is particularly valuable when method names, parameter values, or peripheral types are not specified in advance, as in control logic driven by MQTT or automation rules. By overloading the __getitem__() method of a homebrew wrapper class, the platform makes it possible to pass arguments via subscript syntax. This eliminates the need for the use of hard-coded

if-else chains and enables scalable support of a very large number of peripherals and operations, increasing code readability and extensibility[15].

2.3 Chosen from alternatives

2.3.1 Microcontroller (ESP32)

Three main criteria guided the microcontroller's selection: built-in wireless communication, fit for high-level scripting languages, and enough I/O support for several sensors and actuators, as shown on table 2.1.

Table 2.1: micocontrollers comparaison

Feature	Raspberry Pi (Zero/3/4)	Arduino Mega 2560	ESP32 (Selected)
Built-in Wi-Fi	Yes	No (external module required)	Yes
GPIO Pin Count	26–40	54	34
Programming Language	Python	C/C++	MicroPython
Real-time Capability	Limited (Linux-based)	Basic	Suitable
Power Consumption	High	Low	Low
Cost	High	Moderate	Low
Community Support	High	High	High
Ease of Firmware Upload	Moderate	Easy	Easy
Cloud/Edge Compatibility	Yes	Limited	Yes

Simplified code generation and real-time smart home applications would find the ESP32 ideal as it boasts built-in Wi-Fi, low cost, low power consumption, and MicroPython compatibility.

2.3.2 Microcontroller programming language (MicroPython)

The programming language was chosen considering development simplicity, suitability for dynamic code generation, simplicity in accessing hardware, and runtime consumption of resources. The following table 2.2 compares three possible languages:

Table 2.2: Comparison of Language Options

Feature	C/C++	CircuitPython	MicroPython (Selected)
Syntax Simplicity	Requires knowledge of pointers, memory management, and low-level APIs.	Python-based syntax, beginner-friendly.	Python-based syntax, minimalistic and easy for non-experts.
Memory Efficiency	Very efficient due to direct compilation into machine code.	Requires interpreter and uses more RAM.	Includes interpreter overhead, but optimized for embedded systems.
Runtime Performance	Fastest, as code is compiled and runs directly on the hardware.	Slower due to interpreted execution.	Slower than C/C++, but fast enough for typical IoT tasks.
Library Support	Broadest range, but requires complex setup and configuration.	Limited to boards explicitly supported by Adafruit.	Offers good support for hardware and community-developed modules.
Hardware Control Access	Full control over all registers and pins via direct manipulation.	Simplified APIs, but limited low-level access.	Exposes both high-level and low-level APIs for GPIO, I2C, PWM, etc.
Code Generation Flexibility	Difficult to generate C code dynamically due to syntax complexity.	Easy to generate as syntax is lightweight Python.	Easy to generate readable code snippets via AI tools or templates.
Firmware Size	Smallest binary footprint since no interpreter is needed.	Requires larger firmware with full Python runtime.	Smaller than CircuitPython, but larger than compiled C/C++ firmware.

MicroPython was selected because it offers a good compromise between low-level hardware control, ease of use, and dynamic code generation. Although C/C++ is superior performance- and memory-wise, its complexity eliminates it for both code automation and non-expert users, CircuitPython was also a candidate but is not sufficiently flexible outside specific hardware platforms.

2.3.3 User interface (web application)

In order to enable users to communicate with IoT devices for configuration, control, and monitoring purposes, the interface needed to be accessible, intuitive, and enable real-time communication. Three types of user interfaces were in consideration: mobile applications, desktop applications, and web applications. The evaluation criteria were accessibility, development complexity, platform support, and integration with backend services, as shown on table 2.3.

Table 2.3: Comparison of Interface Platforms

Feature	Mobile Application	Desktop Application (Native)	Web Application (Selected)
Accessibility	Requires installation from app stores; device-specific.	Limited to installed OS; must be downloaded and updated manually.	Accessible via browser; no installation required.
Platform Compatibility	Requires separate versions for Android and iOS.	Requires OS-specific development (e.g., Windows, macOS).	Cross-platform on all modern browsers.
Ease of Development	Requires mobile SDKs and device emulators.	Requires platform-specific languages (C#, Swift, etc.).	Built with HTML, CSS, and JavaScript.
Real-Time Support	Supported, but integration depends on mobile stack.	Supported with desktop APIs or web wrappers.	Easily supports Socket.IO and WebSocket-based communication.
Update & Maintenance	Needs store approval and user updates.	Manual updates or auto-updaters needed.	Instant updates when deployed to server.
Integration with Backend	Requires secure API setup for mobile environments.	APIs must be tailored to native platforms.	Easily communicates with backend over HTTP and WebSockets.
Development Overhead	High (multiple builds + testing per device type).	Medium to high (based on target OS).	Low; single codebase maintained across platforms.

A web-based application was used because it is platform independent, easy to maintain, and presents low barriers to user access. The users can use any modern browser to log in and access the system without having to install any additional software. For installing firmware on devices that need USB access, the same web application is wrapped in a desktop setting by Electron to run the exact interface as a desktop app with hardware access. This hybrid approach maximizes flexibility without replicating development work.

2.3.4 Backend runtime environment (Node js)

The backend is the system's core processing unit, handling user authentication, device registration, real-time messaging, and database management. The key considerations in choosing the backend runtime were asynchronous event handling, real-time communications support, ecosystem maturity, and ease of integration with both MQTT and frontend technologies, as shown on table 2.4.

Table 2.4: Comparison of Backend Environments

Feature	Python (Flask/Django)	Java (Spring Boot)	Node.js (Selected)
Asynchronous Support	Requires additional libraries (e.g., asyncio, Celery).	Multi-threaded model; heavier resource usage.	Natively event-driven and asynchronous.
Real-Time Communication	Possible with WebSocket support, but less native.	Supported but complex with high concurrency.	Built-in support via Socket.IO and WebSocket.
Ease of Integration	Easy with Python libraries, but less flexible for frontend.	Requires more configuration for cross-platform frontend communication.	Seamlessly integrates with JavaScript-based frontend.
Learning Curve	Moderate	Steep	Low to moderate (JavaScript-based).
Ecosystem & Libraries	Mature, especially for data-heavy apps.	Extensive but enterprise-focused.	Large ecosystem, ideal for lightweight web apps.
Performance (Light Loads)	Good for small to medium tasks.	High performance, but overkill for small-scale systems.	Excellent for I/O-bound applications.
Development Speed	Moderate; more boilerplate code.	Slower; requires strict structure.	Fast due to lightweight and modular design.

Node.js was used as backend runtime since it was event driven, non blocking in nature and therefore the optimum choice for real time IoT application. It provides ease of frontend integration, facilitates efficient processing of simultaneous device events, and supports socket natively for real-time updates to dashboard. Additionally, developing using the same language (JavaScript) for frontend and backend for most applications made things easy and provided support for rapid prototyping.

2.3.5 IoT protocols (MQTT)

Communication protocols in IoT solutions are crucial to facilitate the communication of devices . The protocol should allow lightweight messaging, minimize power utilization, and provide real time communication between devices and the server. Various protocols were examined for this project based on messaging architecture, overhead, delivery guarantees, and integration simplicity, as shown on table 2.5.

Table 2.5: Comparison of IoT Communication Protocols

Feature	HTTP/REST	CoAP	MQTT (Selected)
Messaging Model	Request-response (client-server)	Request-response with lightweight binary messages	Publish-subscribe (decoupled messaging)
Overhead	High (verbose headers and stateless)	Very low (minimal headers, UDP-based)	Low (lightweight but TCP-based)
Power Efficiency	Low	High	High
Delivery Guarantees	Limited; no built-in QoS	Basic; unreliable over UDP	Built-in QoS levels (0, 1, 2)
Real-Time Suitability	Poor	Better than HTTP	Excellent for near-instant communication
Integration Complexity	Easy but not scalable	Moderate; less supported	Easy with many available libraries
Scalability	Limited under many devices	Scalable but less mature	Highly scalable for many devices

MQTT was selected due to its low latency, publish subscribe architecture enabling real time communication between devices and server. Its low overhead, with reliable messaging using Quality of Service (QoS) capabilities, and robust under network restricted environments make it suitable for smart home applications . MQTT also enjoys broad support and automatically integrates into Node.js, permitting efficient handling of device telemetry, control messages, and automation events

2.3.6 MQTT broker

An MQTT broker supplies the IoT system's central messaging point that facilitates communication between devices and the server through the publish-subscribe model. When selecting a broker, the key points of consideration were performance under load, ease of setup, scalability, adherence to protocol, and community support, as shown on table 2.6.

Table 2.6: Comparison of MQTT Broker Options

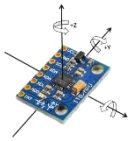
Feature	HiveMQ	EMQX	Mosquitto (Selected)
Open Source	Community version available	Open source and enterprise options	Fully open source
Resource Usage	Moderate to high	Moderate	Very low
Performance	Scalable for large-scale apps	High-performance, clustered-ready	Reliable and efficient for small to mid systems
Ease of Setup	Requires GUI and config tuning	More complex to configure	Simple setup with default config
Protocol Support	MQTT 3.1, 5.0	MQTT 3.1, 5.0, CoAP, LwM2M	MQTT 3.1 and 5.0
Web Interface	Yes (limited in community edition)	Yes	No built-in GUI
Community & Documentation	Good	Growing	Strong and widely adopted
Cloud Compatibility	Yes	Yes	Yes

Mosquitto was chosen due to its lightweight nature. For the present scale and local deployment configuration of this project, Mosquitto offers the optimal trade off between performance and ease of use. It also provides the possibility of future cloud deployment without any major architectural modification.

2.3.7 Sensors and actuators

The sensors and actuators selected were made on several usability factors: existence of such hardware in local markets, ease of interfacing to generic-purpose microcontrollers, relevance to smart home scenarios, and active communities. Devices were picked in order to offer some representative functional domains such as environment sensing, user input, and physical actuation such that the system is representative and extensible, as shown on table 2.7.

Table 2.7: Sensors and actuators are used

Device	Description	Device	Description	Device	Description
	**DHT22** – Temperature and humidity sensor; used for environmental monitoring		–MPU6050: Accelerometer and gyroscope; used for motion/orientation tracking		**Relay Module –** Electromechanical switch for controlling high-voltage loads
	**MQ-2 Gas Sensor** – Used to detect combustible gases; integrated into safety/alert systems		**PIR Motion Sensor** – Detects human motion using infrared; used for intrusion detection		**Buzzer –** Provides audio alerts; used for alarms or feedback signals
	**Rotary Encoder** – Detects rotational position; useful in user input or position feedback systems		**Push Button** – Basic digital input; used for manual triggers or local control		**H-Bridge Motor Driver –** Allows bidirectional control of DC motors; used in robotics or automation
	**Ultrasonic Sensor** – Measures distance to nearby objects; used for obstacle detection or level measurement		**Servo Motor** – Actuator with angular control; used for mechanical positioning or movement		**OLED Display –** Small graphical output screen; used to display dynamic data or status info

These peripherals were selected to enable users to build effective use cases across a range of control and monitoring types. The diversity also helps in flexible automation logic. Overall, the peripherals that were selected provide extensive coverage while being simple enough for guided integration.

2.3.8 Code Generation (AI Prompt) Strategy

The selection of a code generation method is based on four main criteria: scalability, reliability, development time, and maintainability. Every alternative was considered in terms of how well it handles dynamic device configurations and supports non-technical end-users, as shown on table 2.8.

Table 2.8: Code Generation Strategies Comparison

Strategy	Description	Advantages	Disadvantages
1. Manual Native Code	Developers write the entire codebase manually for each configuration.	- Full control over structure and quality- No AI dependency	- Time-consuming- Hard to maintain at scale- Error-prone
2. Fully AI-Generated Code	AI generates the complete firmware based on prompts or structured metadata.	- Fast and scalable- Low technical barrier- Dynamic and adaptive	- Requires strong prompt design- May generate inconsistent or invalid code
3. Hybrid (AI + Manual Template)	AI generates only peripheral-specific parts, merged into a fixed boilerplate.	- Balanced approach- Ensures consistency in shared logic- Lower AI overhead	- Merging logic needed- Still requires AI model integration

The selection of a code generation method is based on four main criteria: scalability, reliability, development time, and maintainability. Every alternative was considered in terms of how well it handles dynamic device configurations and supports non-technical end-users.

2.4 literature review

To contextualize this project within similar academic work, two relevant graduation projects were selected for comparison: **IoT Smart Home Controllers[16]** and **Smart Aqua Guardian[17]**. Both projects share architectural and functional similarities, such as using ESP32 and MQTT, while offering different scopes in automation, deployment, and user interaction.

The **IoT Smart Home Controllers** project focused on controlling home appliances via a cloud dashboard (Ubidots), with optional voice integration using IFTTT and Google Assistant. However, it required users to manually write and upload firmware, lacked hardware connection guidance, and offered limited automation capabilities tied to third-party platforms. In contrast, our project automates firmware generation, guides the setup process, and supports more flexible rule-based automation executed locally.

As seen in comparison details on table 2.9

Table 2.9: literature review comparison table

Criteria	IoT Smart Home Controllers	This Project
Main Focus	Appliance control via mobile/cloud	Simplified IoT setup and monitoring
Code Generation	Manual	Fully automated
Hardware Wiring Guidance	None	Text-based AI-guided instructions
Automation Execution	Cloud-based (Ubidots/IFTTT)	Rule-based, runs directly on ESP32
I/O Complexity	Basic sensors and relays	Mixed — common sensors, displays, switches, motors
MQTT Broker	Ubidots (cloud)	Mosquitto (local/cloud planned)
User Interface	Mobile app, IFTTT	Web dashboard + Electron desktop app
Target Users	Tech-savvy users and hobbyists	Electricians and non-technical users
Deployment Complexity	Medium	Low (guided, no coding required)
Real-Time Feedback	Limited	Real-time via MQTT + Socket.IO

2.5 summary

This chapter reviewed the selected hardware and software components, along with related projects for comparison. It demonstrated how this project simplifies IoT deployment compared to existing solutions. The insights gained here guided the system design choices in the next chapter.

Chapter 3

3) System Design

3.1 Preface

This chapter addresses the structural and architectural design of the Simplified IoT Integration and Management system. It explains how the components of the system are organized, how data is transferred among them, and how the interactions of the user are arranged. The architecture is based on modularity, real time execution, and usability by the non technical user, and it serves as the master plan that guides the implementation of the system in the next chapter.

3.2 System Architecture

The system is modeled on a modular client-server architecture (Figure 3.1: System Architecture) that separates concerns into three tidy layers—the user interface (UI), the backend server, and the IoT devices. This layering supports scalability, real-time communication, and natural user experience in setting up, monitoring, and automating smart-home devices.

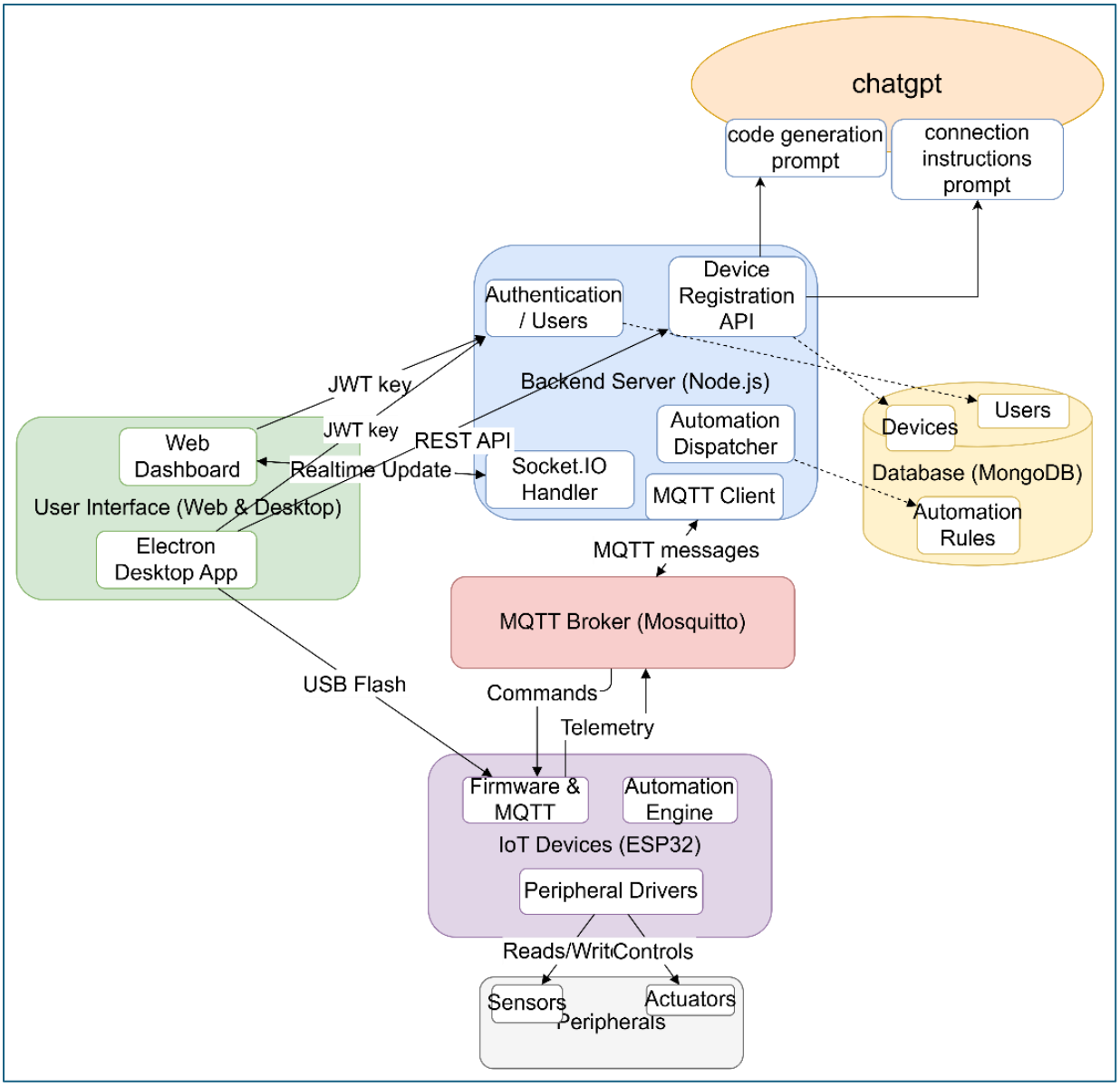


Figure 3.1: system design

3.2.1 User Interface (Web & Desktop)

built with standard web technologies (HTML, CSS, and JavaScript) and also released as an Electron desktop application for USB-based interactions—gives users direct access to all of the system functions, from device registration to real-time dashboard monitoring, control panels, and simple automation setup.

3.2.2 Backend Server

acts as the central controller of the system, handling user authentication, device registration, dispatching of automation logic, and real-time communication. It talks to the database for user and device data storage, uses MQTT for publishing and subscribing to messages with edge devices, and offers real-time dashboard updates through Socket.IO.

3.2.3 IoT Devices (Microcontrollers)

The IoT devices are assembled with auto-generated firmware that allows them to parse incoming commands, perform read/write operations against connected peripherals, and verify user-provided automation logic. The devices all connect to the MQTT broker, subscribe to its related control topics, and report their status to the system at regular intervals.

3.2.4 MQTT Broker (Message Layer)

The MQTT broker, coded with Mosquitto, facilitates asynchronous and light communication from the backend server to distributed devices in IoT. It ensures safe queuing of messages, subscription based on topic, and optimal delivery for both device-to-device and server-to-device communication.

3.2.5 Database Layer

Persistent data in the form of user accounts, registered devices, chosen peripherals, and stored automation logic are stored in the database layer, allowing safe access and context-aware device management across the platform.

3.2.6 ChatGPT Service

As Figure 3.1 shows, the backend calls a special ChatGPT micro-service whenever a person needs firmware or wiring help. Two hardcoded prompt templates drive this service: a code-generation prompt that produces flash-ready MicroPython for the peripherals on which the user

wishes to operate, and a wire-connection prompt that produces plain-text, step-by-step wiring instructions. Both prompts receive structured JSON from the backend, generating deterministic AI output and freeing users from needing to write code or decode pin maps.

3.3 Data Flow and Interaction Scenarios

3.3.1 User Registration Flow

To explain the sequence diagram 3.2

1. The user opens the registration page in the web application.
2. The user type email address and password.
3. After user submit, the client side sends the user registration request to the server
4. The server side:
    a. Verifies the if email exists
    b. the password is encrypted using hashing technique
    c. Stores the email, and encrypted password in the database.
5. After registration is successful, a JWT (JSON Web Token) is sent back to the client by the server.
6. The JWT is stored by the client in the browser session (HTTP only cookie), in order to use it in all subsequent operations :
    a. Device registration
    b. Real-time monitoring
    c. Real-time control
    d. Automation logic configuration

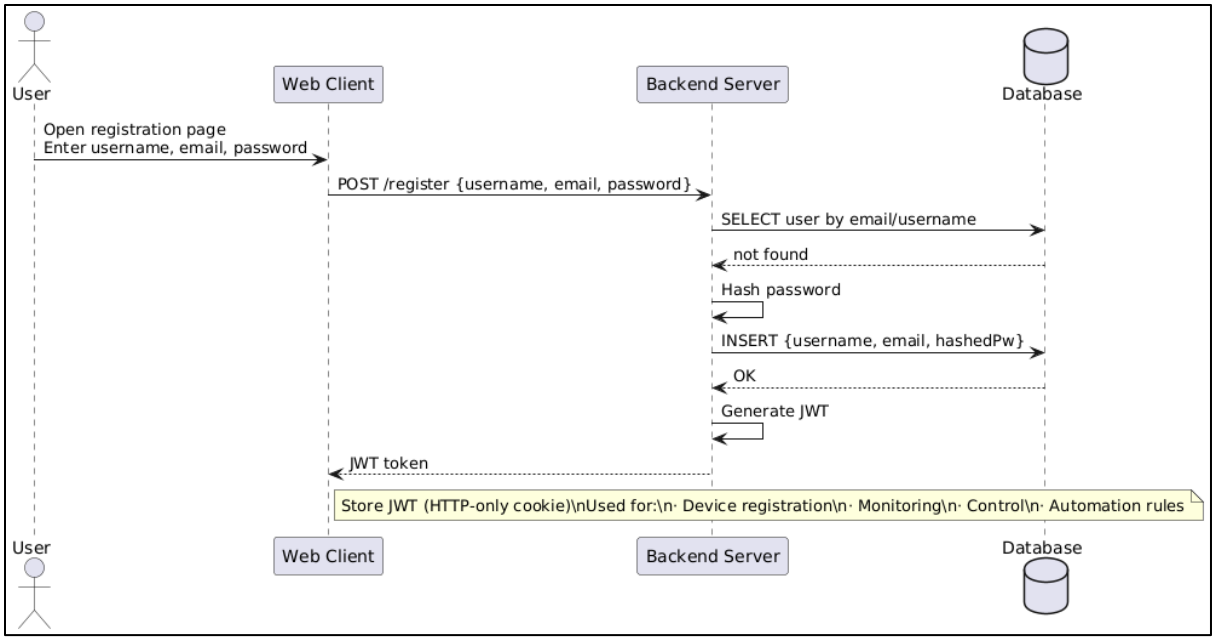


Figure 3.2: user registration sequence diagram

3.3.2 User Login Flow

To explain the sequence diagram on figure 3.3

1) The user open login page on the web/desktop application
2) The user enters email and password
3) After use submit, the client send request handling user logging to get authenticated from the server.
4) The server check the logging info, if the credentials are valid, then generates a JWT (JSON Web Token).
5) The JWT is sent to the client and stored in the browser session (HTTP-only cookie), This token is attached to all future authenticated requests for:
    i) Device registration
    ii) Real-time monitoring
    iii) Real-time control
    iv) Automation logic configuration

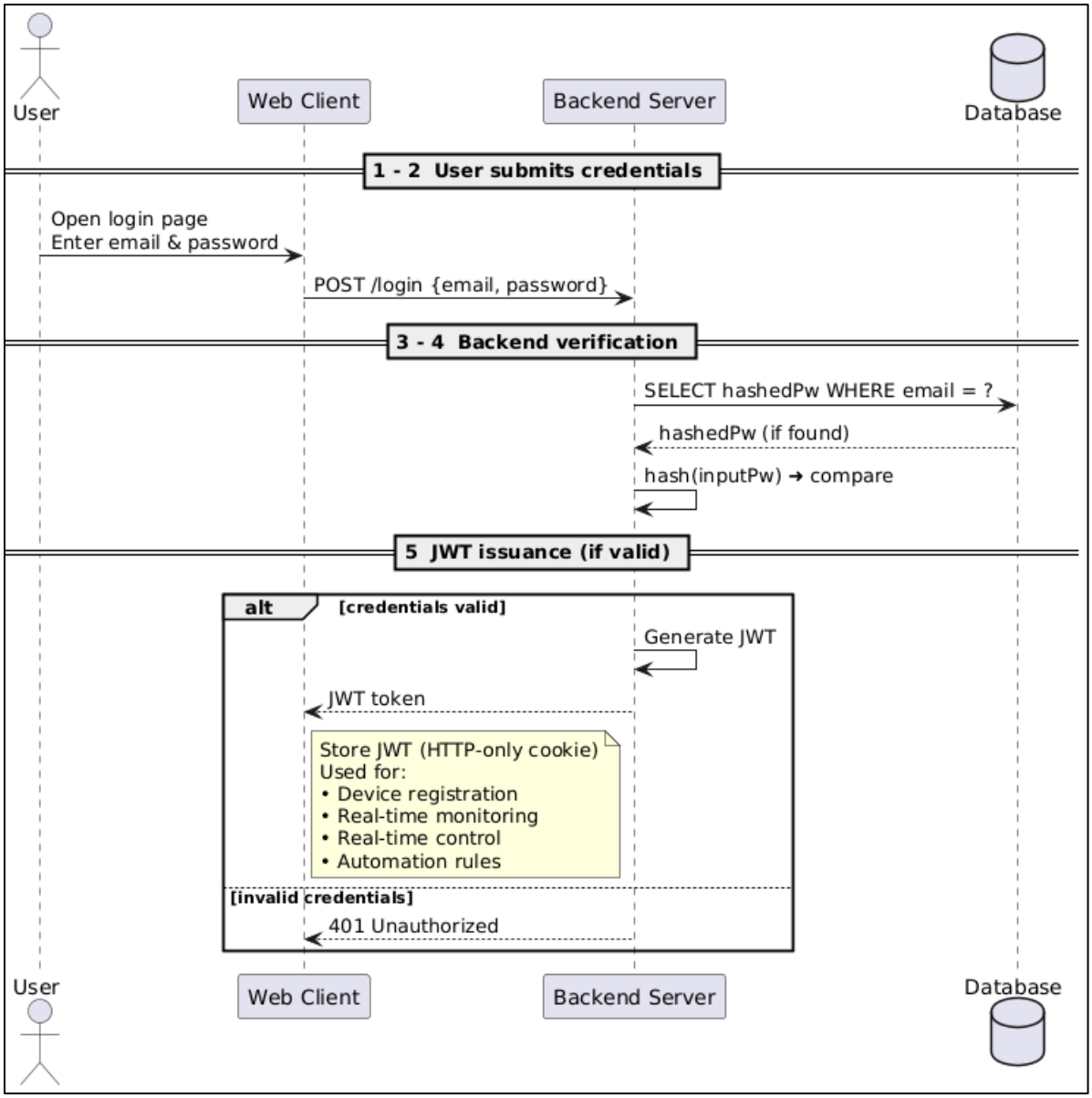


Figure 3.3: user login sequence diagram

3.3.3 Device Registration Flow

To explain the sequence diagram on figure 3.4

1) The **user (electrician)** opens the **desktop application** and login with his credentials (email, password), and get authenticated from the server.
2) The user navigates to the **device registration page**

3) The user enters the **microcontroller name** and selects the **peripherals** to be connected like : DHT sensor, switch, servo motor, etc.
4) After submission, the client (desktop app) open socket connection with the server
5) The client sends the submitted data (microcontroller name and selected peripherals) through the socket connection.
6) The server:
    a) Extracts the **user ID** from the JWT.
    b) Generates a **unique device ID**.
    c) Stores the device information in the **database**, with reference to **user id**
7) The server initiates the **code generation process** by calling a **ChatGPT API**, passing the selected peripherals and device metadata from the user.
8) The **generated code** includes:
    a) **Wi-Fi and MQTT broker credentials**.
    b) Required **library import links** for the selected peripherals.
    c) Asynchronous functions for peripherals operations (read/write)
    d) A **callback handler** to process incoming MQTT messages.
9) The server sends the generated code and library references back to the client (desktop application)
10) The **desktop app installs the code and libraries** onto the connected ESP32 via USB
11) The esp32 connects to WiFi, MQTT broker to become fully operational

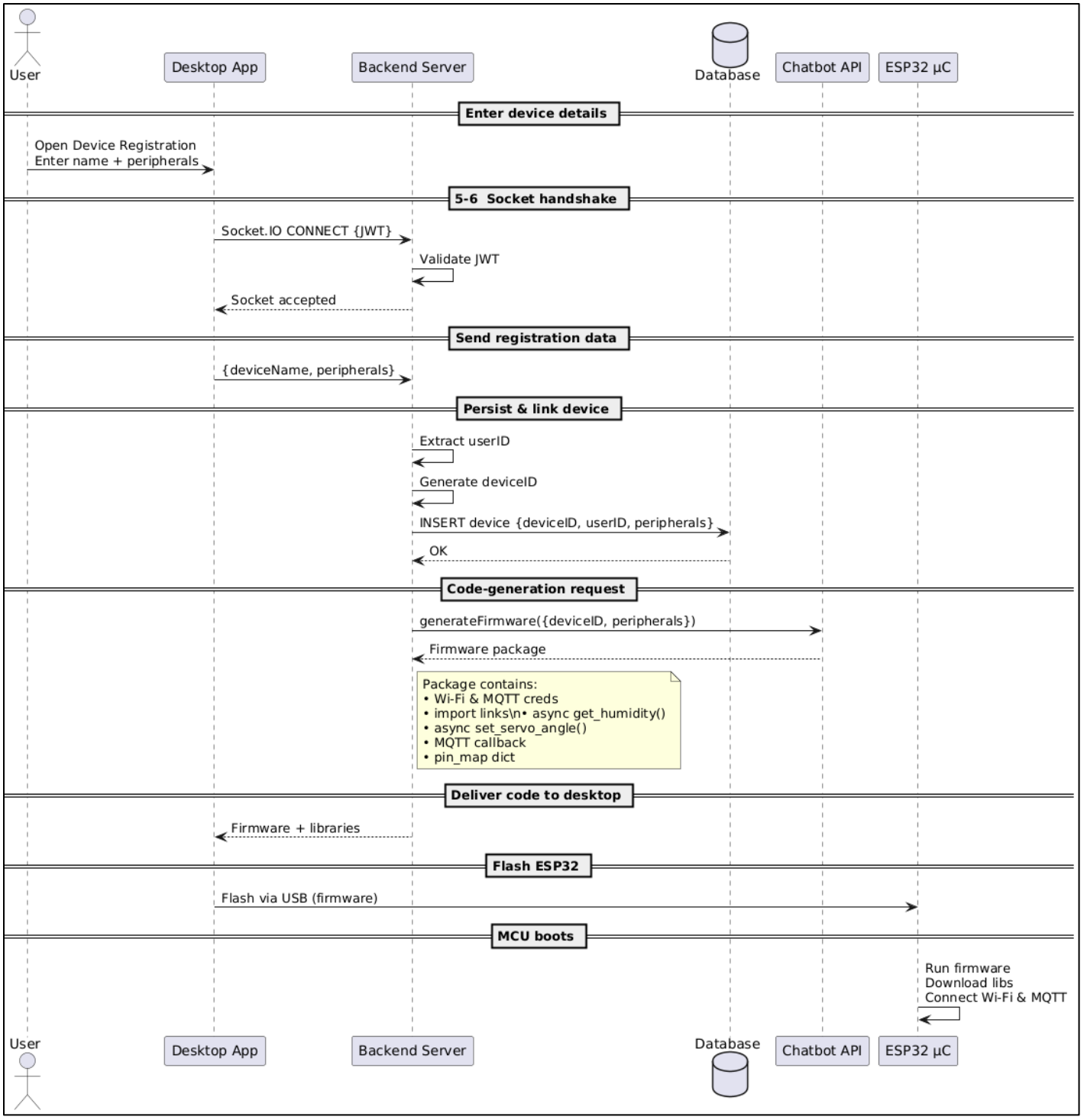


Figure 3.4: device registration sequence diagram

3.3.4 Dashboard Page Loading Flow

To explain the sequence diagram on figure 3.5

1. the user (customer) open the dashboard in web application to read/write the installed IoT device (esp32 microcontroller connected to peripherals) via Internet
2. the client (web application) establish socket connection to the server.
3. the server return the list of IoT devices to the client and the list of cards to put in dashboard, that are saved in database

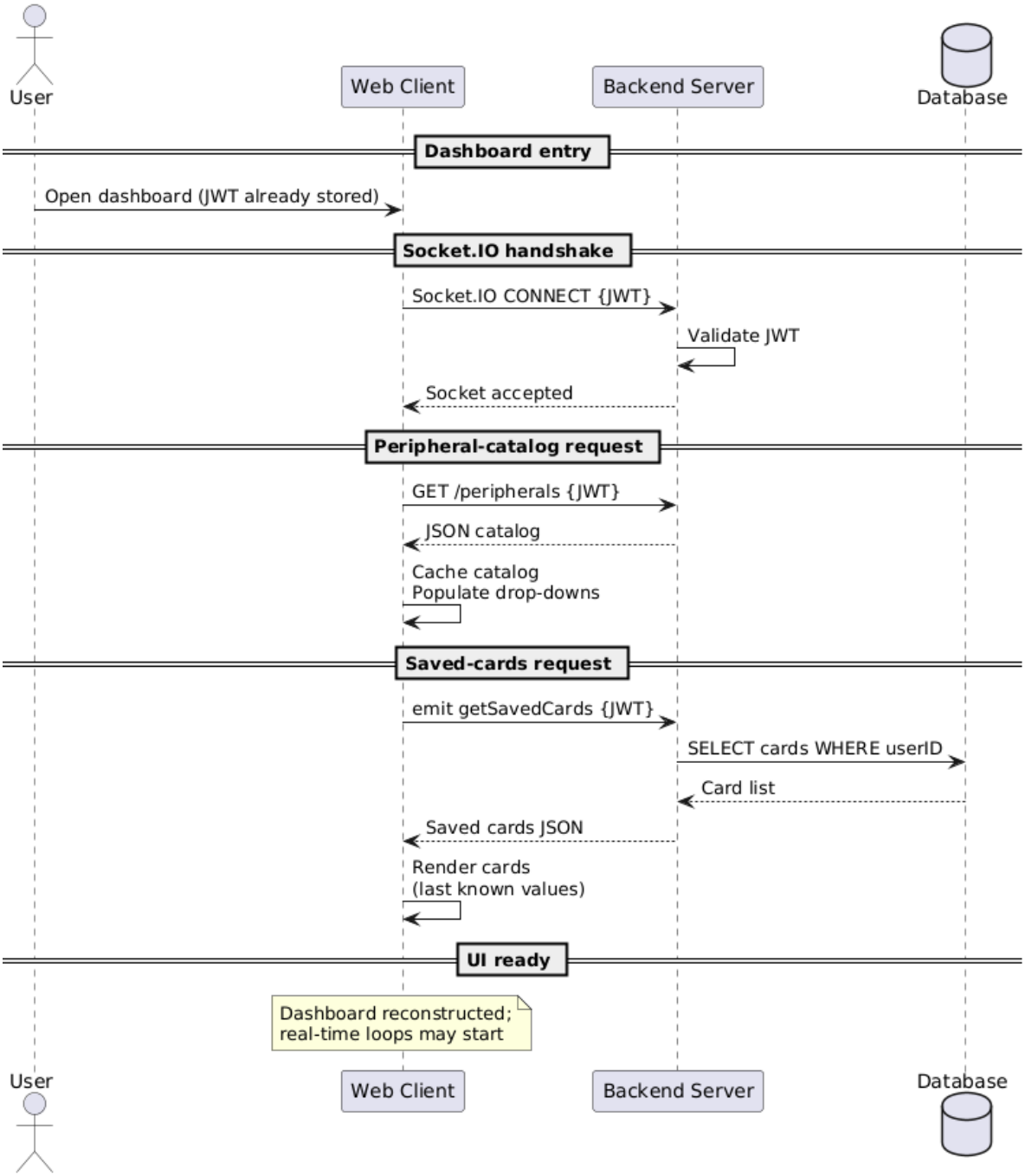


Figure 3.5: Dashboard page loading sequence diagram

3.3.4 Dashboard Card Creation

To explain the sequence diagram on figure 3.6

1. on the dashboard that user click (Add Card) button
2. the client(web app) open a window to configure the card
    a. card type : choose **read or write** card on IoT device
    b. choose **microcontroller** from list, for example (esp32 kitchen node)
    c. choose **peripheral** that is connected to microcontroller, for example (servo motor)
    d. choose the function, for example (read ange) if the card type is read or (set angle) if the card type is write
3. the client automatically configure the remained fields, like the range of angle (0 – 180)
4. after card creation, the card is stored on database to be fetched later, when loading the dashboard page

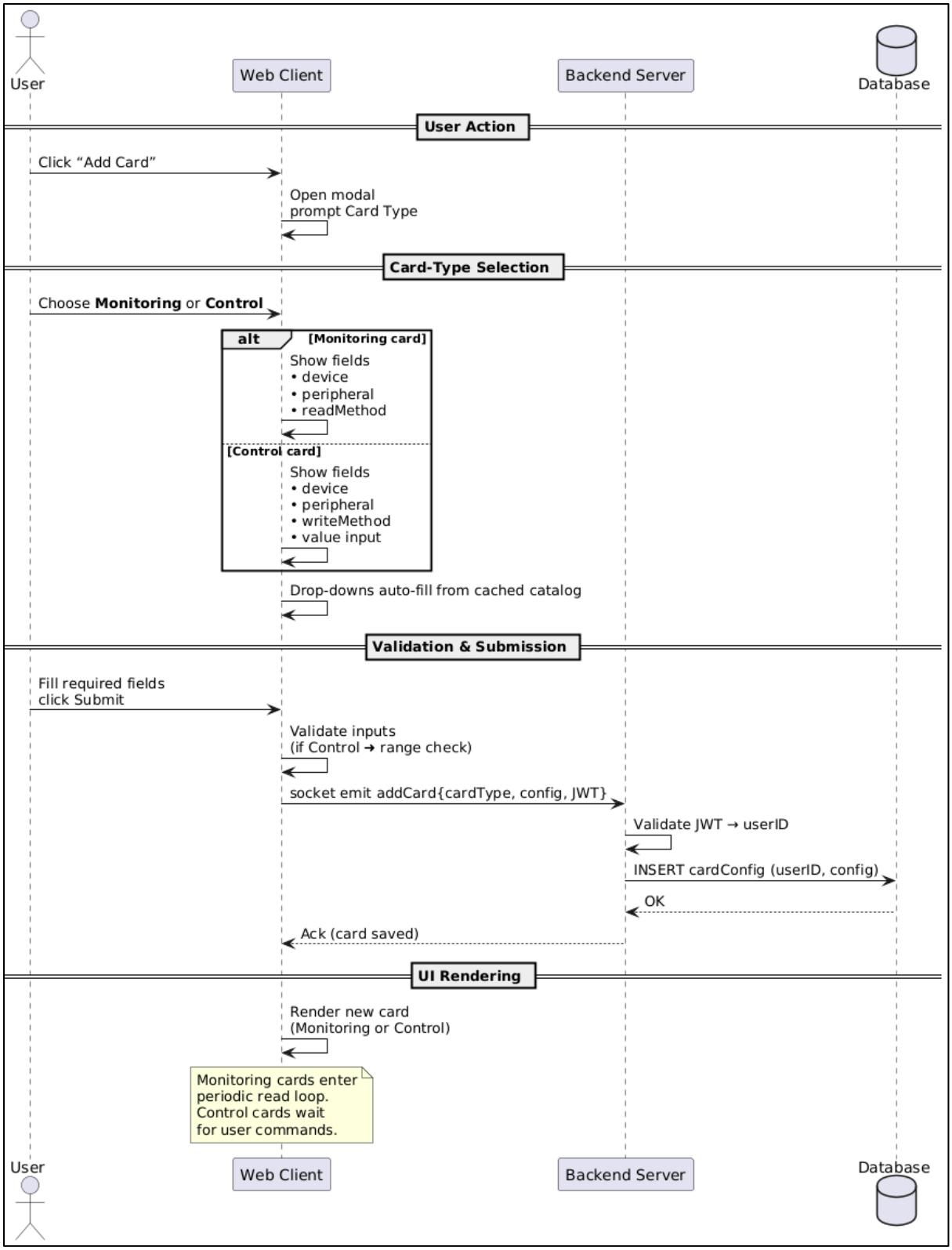


Figure 3.6: Dashboard card creation sequence diagram

3.3.4 Real-Time Monitoring Flow (Client trigger)

To explain the sequence diagram on figure 3.7

1. when the read type card is rendering on dashboard, (for example card of servo motor to read the angle periodically)
2. the client (web app) each read request periodically to the server
3. the server handle the request and forward it to the specified IoT device, for example (esp32 kitchen node – servo motor)
4. the esp32 recieves this request via MQTT, and public the angle value back to the server
5. the server forward the retuned value to the client (web app)
6. the client render the value in the card to the user

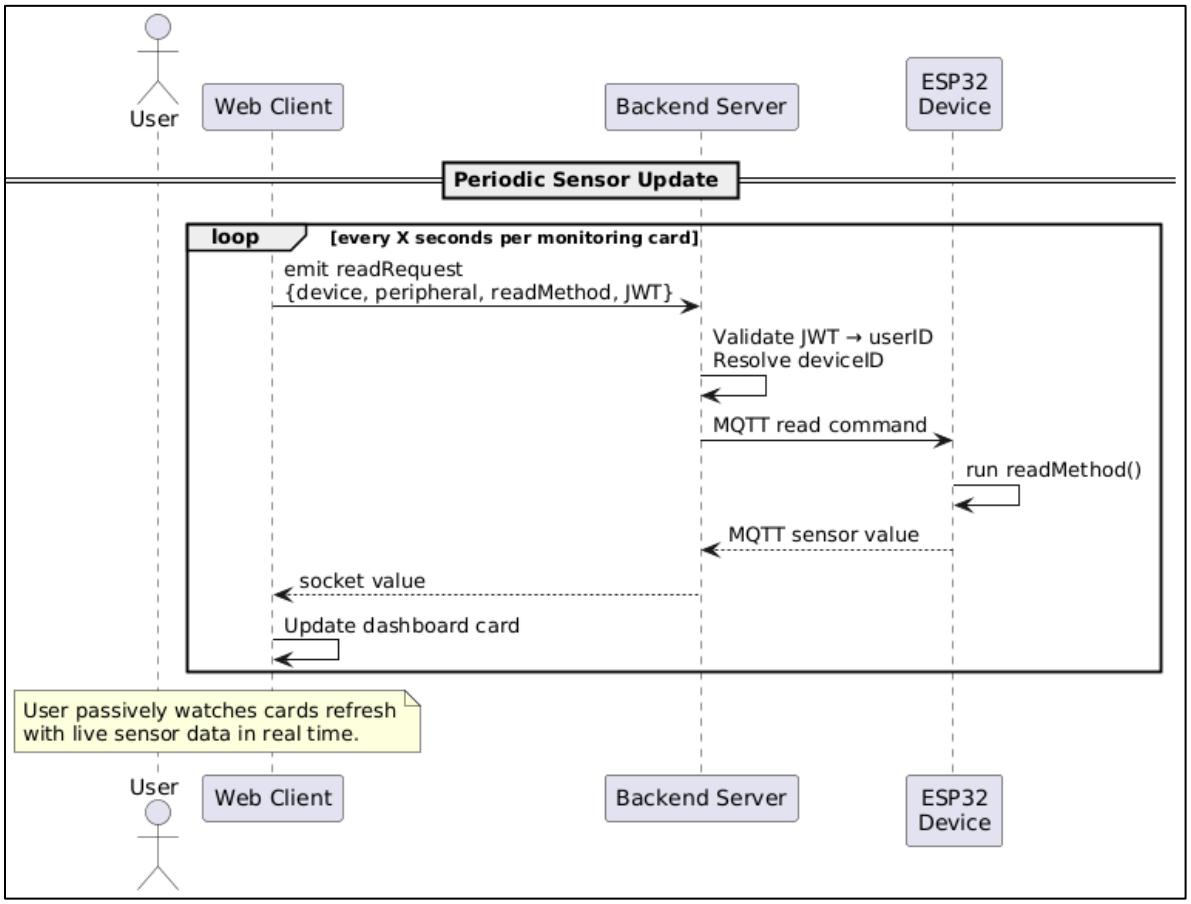


Figure 3.7: real-time monitoring(client trigger) sequence diagram

3.3.4 Real-Time Monitoring Flow (IoT device trigger)

To explain the sequence diagram on figure 3.8

1. after the card creation on dashboard, and saved on database by server
2. the server send request to IoT device specified in the card, for example (esp32 kitchen node – motion sensor – moving()) , to send the value of angle only if changed
3. the IoT device receives the request, and set callback method to Motion sensor object in the code, to get the value if changed as interrupt (sensor trigger)
4. the sensor raise an interrupt signal when motion happens
5. the esp32 microcontroller publish the trigger to the server
6. the server handle the receives the message from IoT device then forward to client
7. the client (web app) receives and render the value on interface card for the user (customer)

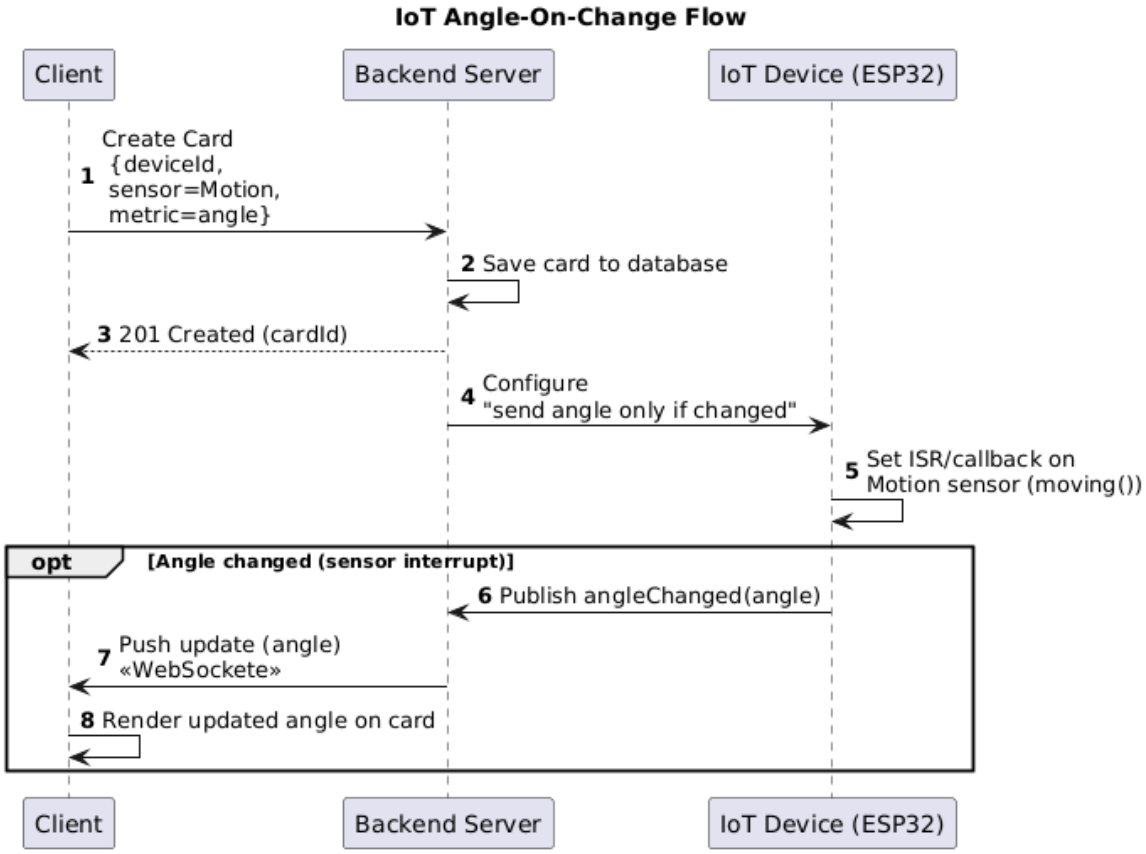


Figure 3.8: real-time monitoring(IoT device interrupt) sequence diagram

3.3.5 Real-Time Control Flow

To explain the sequence diagram on figure 3.9

1. after the card creation on dashboard, and saved on database by server
2. the user write a value on the card to control the IoT device, for example type the angle of servo motor on the card to command the IoT device, for example : (esp32 kitchen node – servo – setAngle(55))
3. the client (web app) send the command to the server via socket connection
4. the server forward the command to the IoT device specified in the command (esp32 kitchen node)
5. the IoT device (esp32) receives the command via MQTT from the server, and execute the command accordingly, for example : to set the angle of servo motor 55 degrees
6. the IoT send acknowledgement and the new angle value to the server
7. the server forward the new value to the client (web app)
8. the client render the value on the card to show it to user

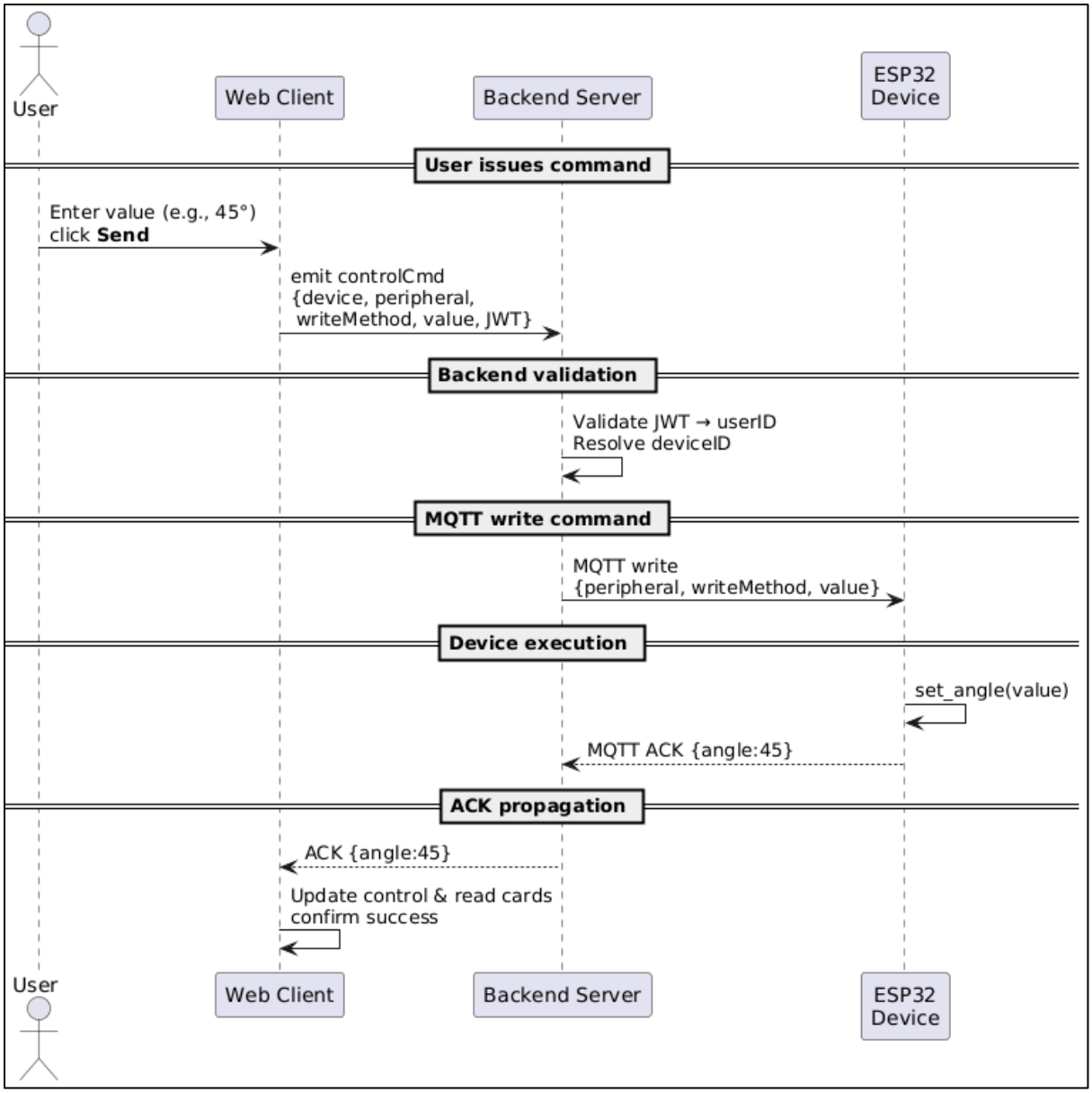


Figure 3.9: real time control sequence diagram

3.3.6 Automation Execution Flow (Periodic Loop Check)

To explain the sequence diagram on figure 3.10

1. on the dashboard, the user click on “Create Automation” button
2. the client (web app) displays a model to determine **Automation Login** :
    a. Input IoT device
        i. Select esp32 node from the list, for example (esp32 door node)
        ii. Select the connected peripheral from the list, for example (dht22 sensor)
        iii. Select the read function, for example (readTemperature())
        iv. Select the threshold value, for example (if temperature > 25)
    b. Output device
        i. Select the esp32 node from the list, for example (esp32 living room)
        ii. Select the connected peripheral, for example(relay that is connected to a Fan)
        iii. Select the write function, for example (turn_on())
3. The model is filled and submitted to the server via socket connection
4. The server receives the **Automation Logic** info and forward it to the input IoT device
5. The Input IoT device receives the Automation Logic, and trigger a command when the condition of logic is true, for example (if temperature > 25 , then send turn_on() command to the Output IoT device
6. The Output IoT devices receives a command from the IoT input device, and run specified function in the command (turn_on() the relay)

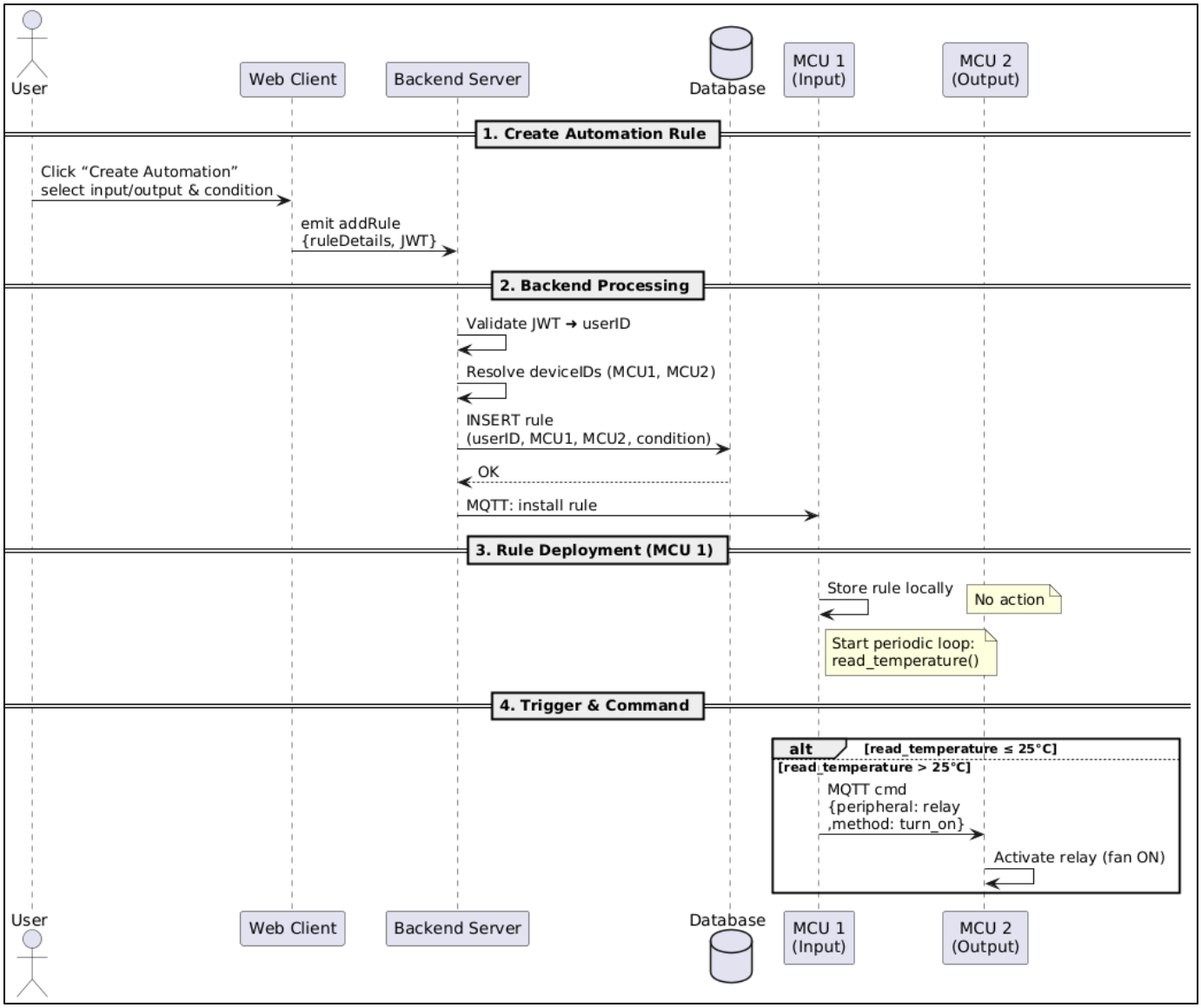


Figure 3.10: Automation Execution (periodic Loop check) sequence diagram

3.3.7 Interrupt-Driven Automation Flow

To explain the sequence diagram on figure 3.11

1. on the dashboard, the user click on “Create Automation” button
2. the client (web app) displays a model to determine **Automation Login** :
    a. Input IoT device
        i. Select esp32 node from the list, for example (esp32 door node)
        ii. Select the connected peripheral from the list, for example (dht22 sensor)
        iii. Select the read function, for example (readTemperature())
        iv. Select the threshold value, for example (if temperature > 25)
    b. Output device
        i. Select the esp32 node from the list, for example (esp32 living room)
        ii. Select the connected peripheral, for example(relay that is connected to a Fan)
        iii. Select the write function, for example (turn_on())

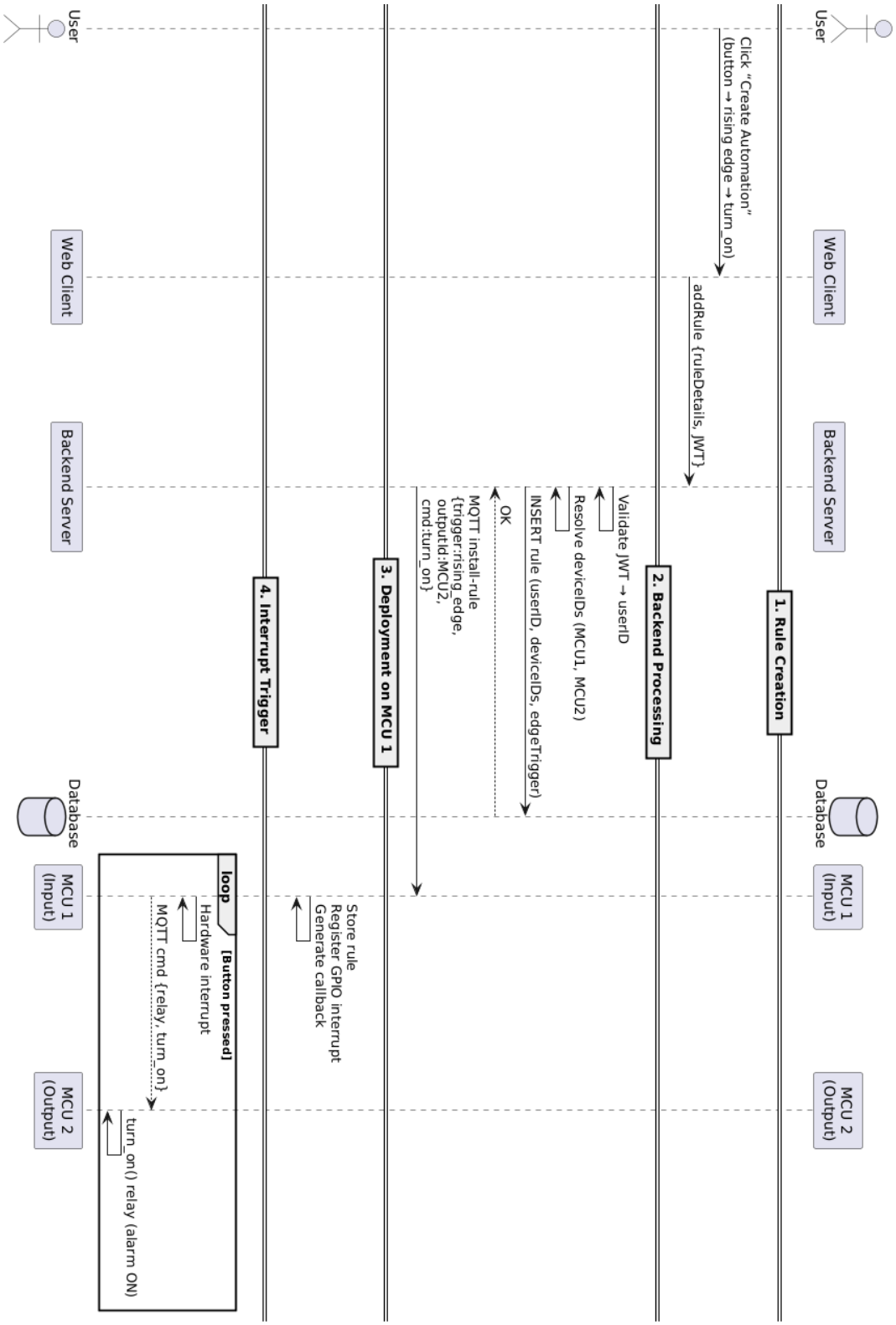


Figure 3.11: Interrupt-Driven Automation Sequence Diagram

3.3.8 Automation Execution Flow (Single-Controller Rule)

Same to 3.3.6 or 3.3.7 but the Input IoT device and the Output IoT device are at the same esp32 node.

3.3.9: Connection guidance (esp32 – peripherals) flow:

To explain the sequence diagram on figure 3.12

1. after *3.3.3 Device Registration Flow* scenario, the user (electrician) go to devices page in web/desktop application
2. the user selects the registered device between the devices, to open device details model
3. on the model click “show connections” of the device
4. the client (web/desktop app) send request to server with the selected device Id, to get the connection instructions in text format
5. the server receives the request, and call the AI API with specific template prompt filled with pin connections in JSON format that are fetched from the database, for example
    a. {servo_motor : 13, ultra_sonic : {11, 12}} , (key : peripheral type, value : pin number on esp32)
6. the AI API generates the connection guidance text and return it to the server
7. the server forward the text to the client
8. the client (web/desktop) app shows the connection guidance on model window

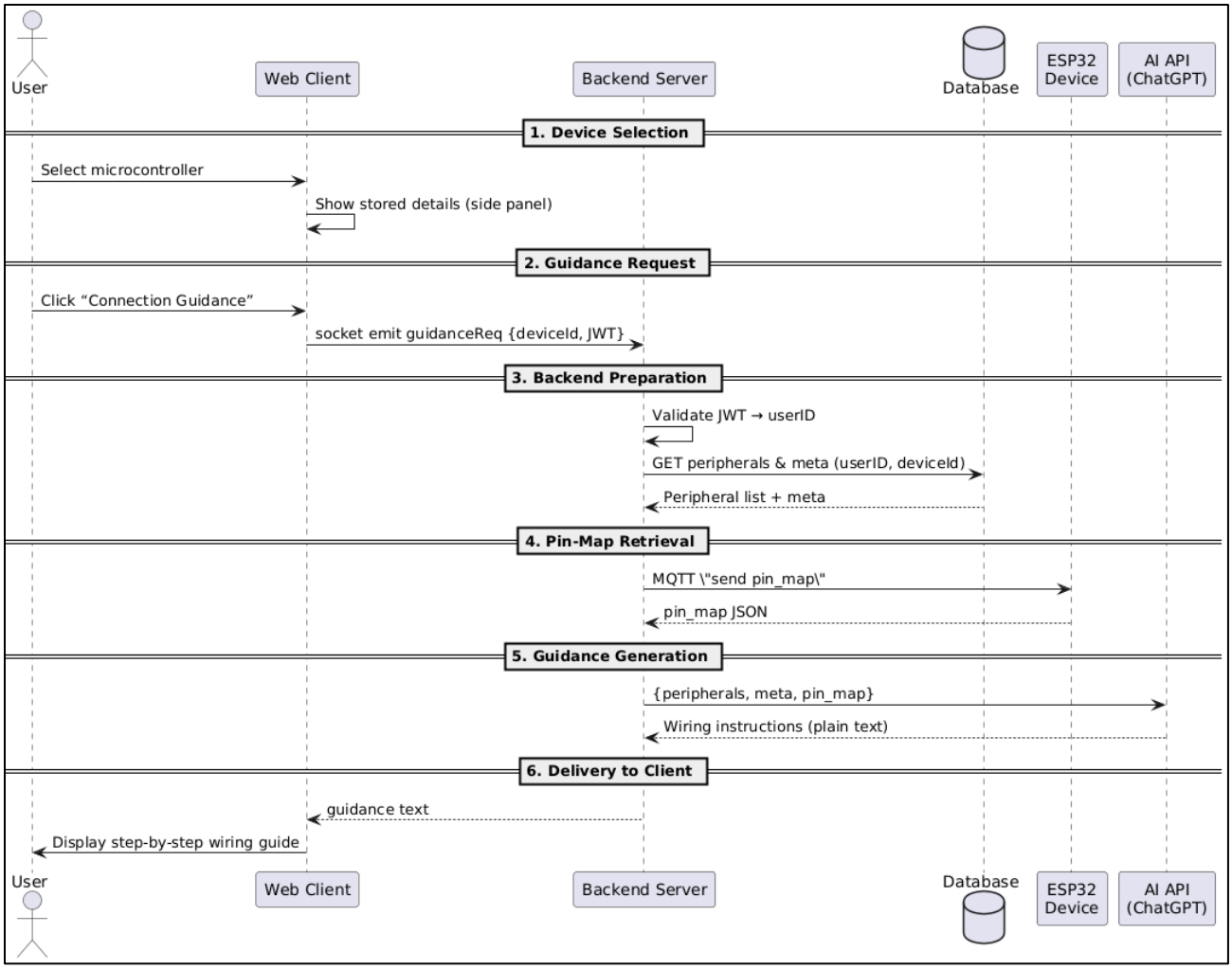


Figure 3.12: Connection guidance

Chapter 4

4) Implementation

4.1 preface

This chapter outlines the practical implementation of the proposed IoT platform, covering both software and hardware aspects. It details how the system was built to support device registration, code generation, connection guidance, and real-time monitoring and automation. The implementation is structured around key user scenarios, supported by interface screenshots and data flow explanations. Together, these elements demonstrate the system's ability to simplify IoT integration for electricians and end-users.

4.2 Implemented System Architecture

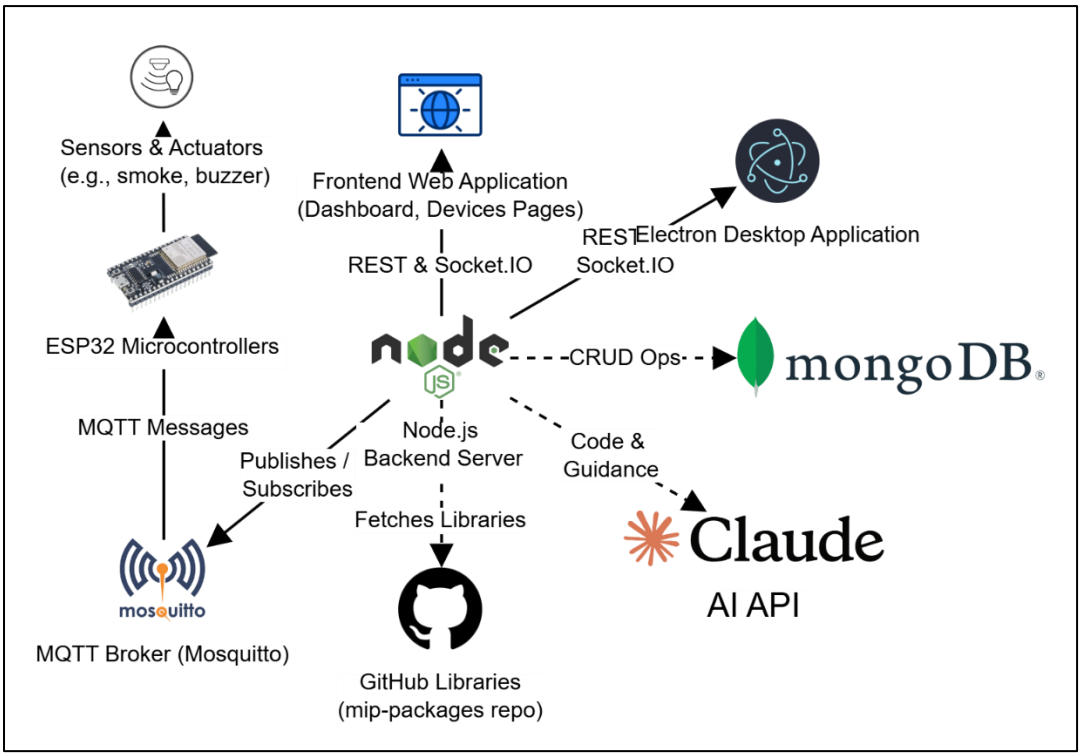


Figure 4.1: system architecture

As shown on figure 4.1, The system is built using client-server architecture, by integrating the user interfaces, which are a desktop application (Electron) and a web application, with the back-end

server (Node.js) that acts as a central manager, combining with an IoT platform (MQTT Broker and the connected ESP32s)

- **Electron Desktop Application:** a user interface for an electrician to register IoT devices on a customer account, by selecting the peripherals to connect, configure the WiFi SSID and Password, flash the generated code from the server on the ESP32 via USB, and show the connection guidance.
- **Frontend Web Application:** a user interface for customers, for IoT device management that is installed from the electrician. It includes:
    - A **Dashboard Page** for real-time device monitoring and control, displaying sensor readings and allowing actuation.
    - A **Devices Page** that lists all registered microcontrollers and enables users to retrieve connection guidance for each device.
- **Node.js Backend Server:** central management for user accounts and their IoT devices (real-time communication, code generation). It acts as a central bridge between the user interface and the IoT platform.
- **MQTT Broker (Mosquitto):** A central hub in IoT platform, that acts a communication bridge between the backend server and the IoT devices (esp32s with connected peripherals)
- **MongoDB Database:** Stores user accounts, IoT devices data : esp32 node, the connected peripherals, the automation logic.
- **ESP32 Microcontrollers:** execute the generated code, ready to send/receive data from the user interface or to other ESP32 for automation logic.
- **GitHub Libraries:** store all the peripherals libraries, and are ready to download remotely from esp32 as needed
- **Sonnet AI API:** used for code generation and connection guidance, by providing prompt from back end server which is a template filled with user specification.

4.3 Implementing Data Flow Scenarios

4.3.1 User Registration

The user registration process allows new users to create a secure account, which is required to access the system’s device management and control features. This scenario involves form

submission from the client (Electron app or web interface), backend validation, and database insertion.

Step 1: Navigating to the Registration Page

The user opens the **web application** and selects the **Register** option to create a new account.

Step 2: Filling the Registration Form

The user enters their **name**, **email**, and **password** into the registration form. shown on figure 4.2

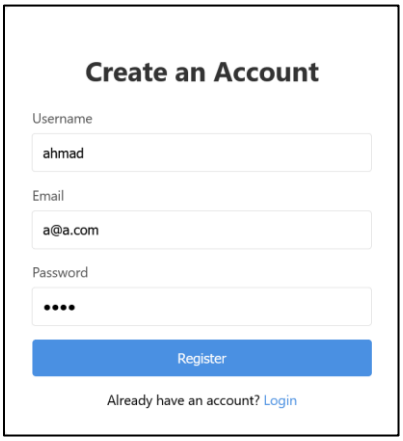


Figure 4.2: Registration Form with Input Fields

Step 3: Submitting the Form to Backend

```
{
  "name": "ahmad",
  "email": "a@gamil.com",
  "password": "1314"
}
```

Step 4: Backend Validates and Stores User

The server validate the registration data, encypt the password and store the new user into the database , for example it is stored in this format :

```
{
  "_id": "612334dsf4…",
  "name": "ahmad",
  "email": "a@a.com",
```

```
  "password": "$21asdf43r3rs2334…”
}
```

Step 6: Backend Responds with Confirmation

If registration is successful, the backend sends a confirmation response, optionally including a JWT key

```
{
  "message": "User registered successfully",
  "token": "eyJhbGciOiJIUzI1sdfasdf3435wasJ9..."
}
```

The token is stored on the client (Electron Store or localStorage of web application) for authenticated access.

4.3.2 User Login

Step 1 – Open the Login Page

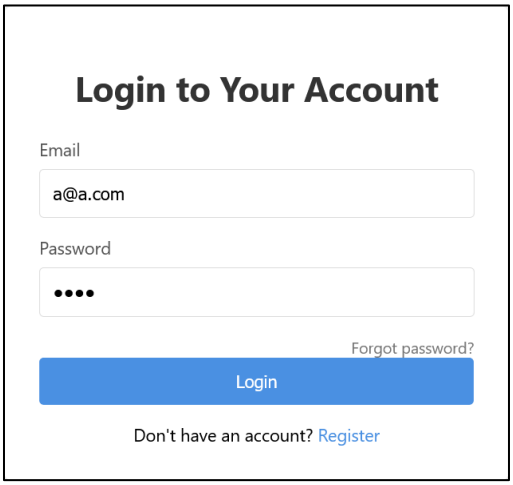


Figure 4.3: login form

Step 2 – Enter Credentials
the user type the email and password

Step 3 – Submit to the Backend

The user click login to submit the credentials to the back end server

```
{
  "email": "a@a.com",
```

```
  "password": "1314"
}
```

Step 4 – Backend Verification

The backend verifies the credentials and return a JWT key token for client

```
{
  "message": "Login successful",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

To use it in later authenticated operations

Step 5 – Access Granted

With a valid key token, the authenticated page is loaded **dashboard** page, **devices page,** or **add device page**

4.3.2 Add New Device

Step 1: Opening the Add Device Form

After the user got authorized access establishing socket connection, esp32 connected via USB. As shown on figure 4.4, open **Add Device From** to add new IoT device the form has the following fields :

- Device name (esp32 kitchen node)
- Device location (kitchen)
- Wi-Fi SSID and password
- List of peripherals (e.g., DHT22, switch, relay)

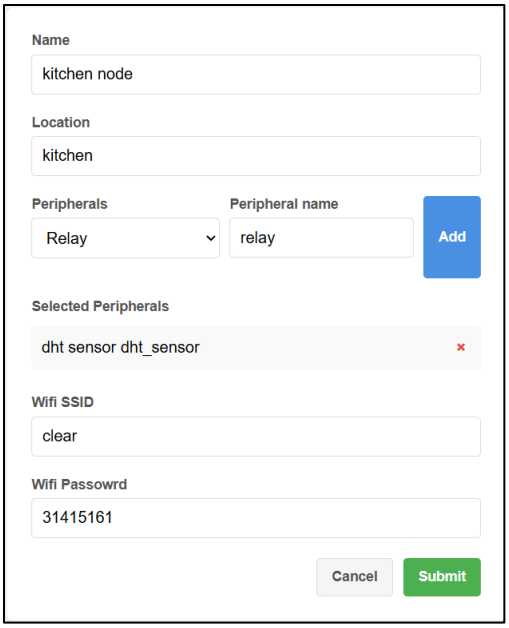


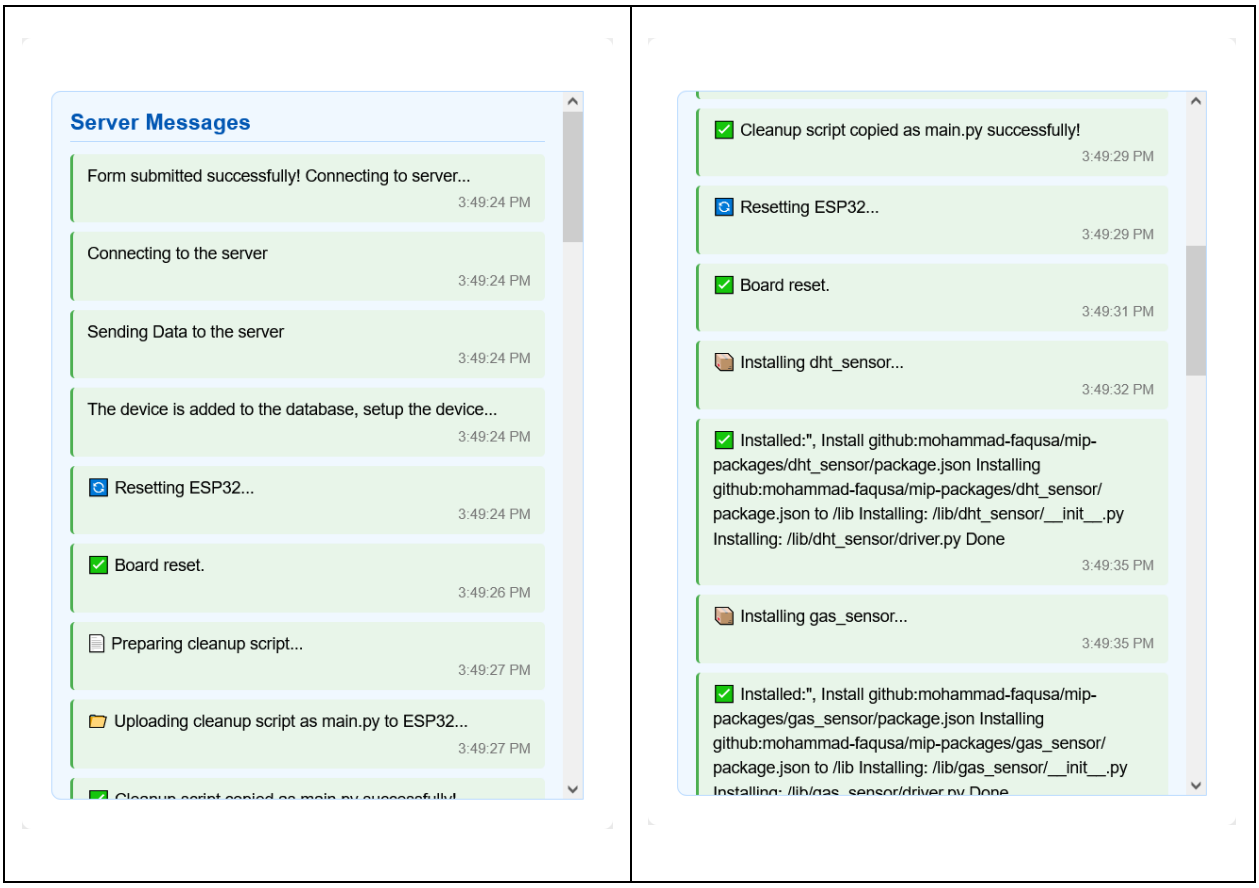
Figure 4.4: add device (microcontroller) form

Step 2: User submit the form

Once the form is submitted, the process starts with real time feedback from backend server and desktop application, which includes :

- Resetting the esp32 from client side (desktop application)
- Code generation with AI API from back end server
- Install the generated code and download libraries from github at client side (desktop application)
- Running esp32 with the installed code, and connects to MQTT broker

The following is the runtime feedback process :


Server Messages
Form submitted successfully! Connecting to server...
3:49:24 PM
Connecting to the server
3:49:24 PM
Sending Data to the server
3:49:24 PM
The device is added to the database, setup the device...
3:49:24 PM
Resetting ESP32...
3:49:24 PM
Board reset.
3:49:26 PM
Preparing cleanup script...
3:49:27 PM
Uploading cleanup script as main.py to ESP32...
3:49:27 PM
Cleanup script copied as main.py successfully!
3:49:29 PM
Resetting ESP32...
3:49:29 PM
Board reset.
3:49:31 PM
Installing dht_sensor...
3:49:32 PM
Installed:", Install github:mohammad-faqusa/mip-packages/dht_sensor/package.json Installing github:mohammad-faqusa/mip-packages/dht_sensor/package.json to /lib Installing: /lib/dht_sensor/__init__.py Installing: /lib/dht_sensor/driver.py Done
3:49:35 PM
Installing gas_sensor...
3:49:35 PM
Installed:", Install github:mohammad-faqusa/mip-packages/gas_sensor/package.json Installing github:mohammad-faqusa/mip-packages/gas_sensor/package.json to /lib Installing: /lib/gas_sensor/__init__.py

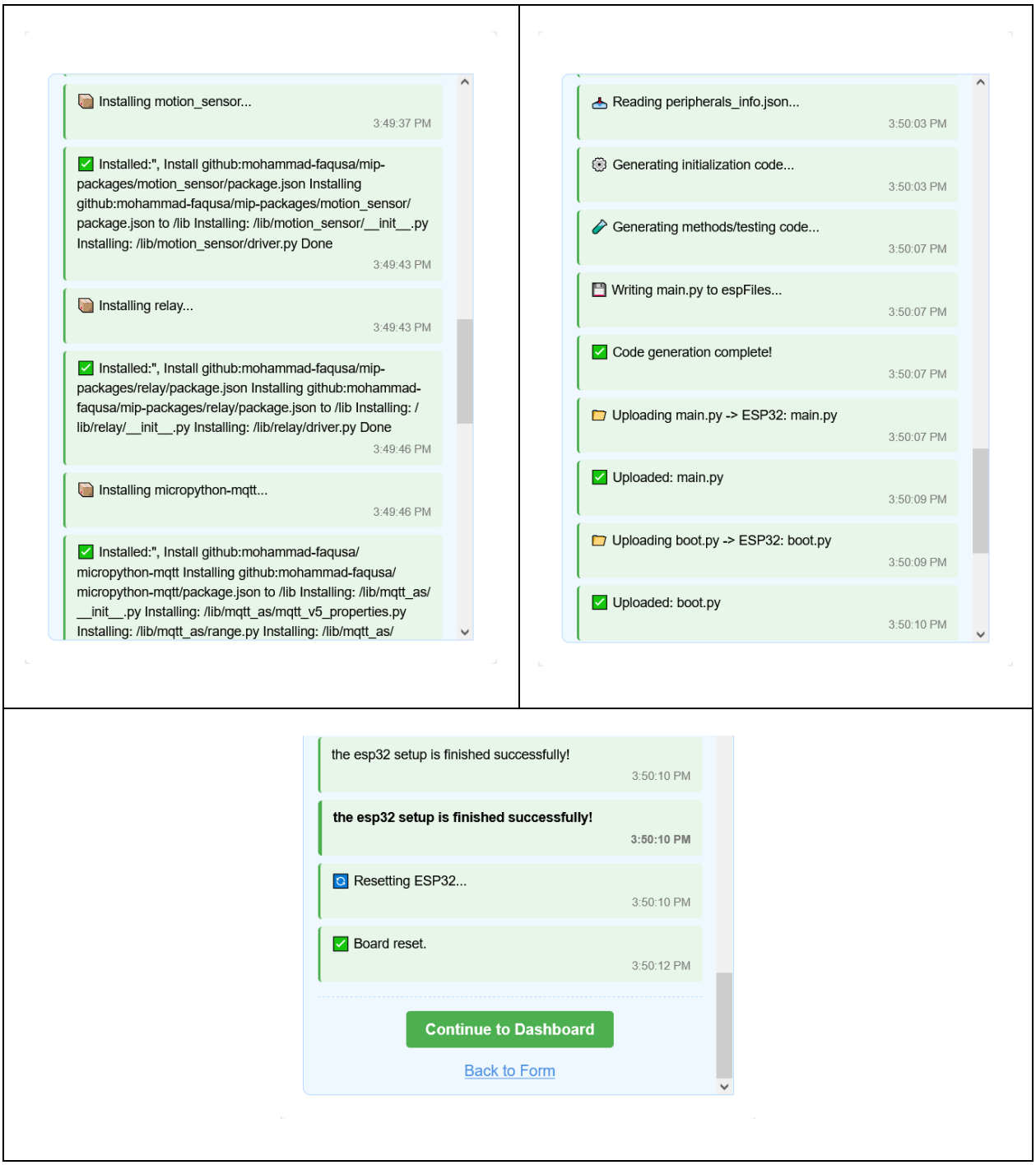


Figure 4.5: the runtime feedback process while device registration

AI prompt template for code generation :

```
function initializePeripheralsPrompt(peripherals_info) {
  return `
  you are micropython esp32 code generator
  just write a python code to copy it directly on python file 'main.py'
  I need to use the following peripherals : ${peripherals_info.map(
    (p) => p.name
  )},

  first import the class for each peripheral as following :
  ${peripherals_info
    .map((p) => `from ${p.library_name} import ${p.class_name}`)
    .join("\n")}

  initialize the 'peripherals_pins' dict, that take the peripheral name as key, and it's
connected pins as dict value, it will be used later to know pin connection.
  then initialize each peripheral class as following, and store it in peripherals dict
'peripherals' :

  ${peripherals_info
    .map(
      (p) =>
        `the class name for peripheral ${p.name} is ${
          p.class_name
        }, and the constructor parameters are : ${JSON.stringify(
          p.constructor.parameters
        )}
          and it is intialized like this : peripherals[${p.name}]=${
          p.class_name
        }(parameters...), consider the default values of parameters.
        then insert the connected pins to 'peripherals_pins[${p.name}]'
  `
    )
    .join("\n")}

    and don't write anything else.

  `;
}
```

The part of esp32 code written by AI :

```
# Initialize peripherals_pins dictionary
peripherals_pins = {}

# Initialize peripherals dictionary
peripherals = {}

# Initialize DHT Sensor (using pin 4, DHT22 type, simulation mode)
peripherals['dht_sensor'] = DHTSensor(pin=4, sensor_type="DHT22", simulate=True)
peripherals_pins['dht_sensor'] = {'pin': 4}

# Initialize Gas Sensor (using pin 34, analog mode, simulation mode)
peripherals['gas_sensor'] = GasSensor(pin=34, analog=True, simulate=True)
peripherals_pins['gas_sensor'] = {'pin': 34}

# Initialize Motion Sensor (using pin 5, simulation mode)
peripherals['motion_sensor'] = MotionSensor(pin=5, simulate=True)
peripherals_pins['motion_sensor'] = {'pin': 5}

# Initialize Relay (using pin 2, active high logic, simulation mode)
peripherals['relay'] = Relay(pin=2, active_high=True, simulate=True)
peripherals_pins['relay'] = {'pin': 2}
```

4.3.3 Wire Connection Guidance

After successful registration of the IoT device, it listed on **Devices page** , show the device details as following in figure ( ) :

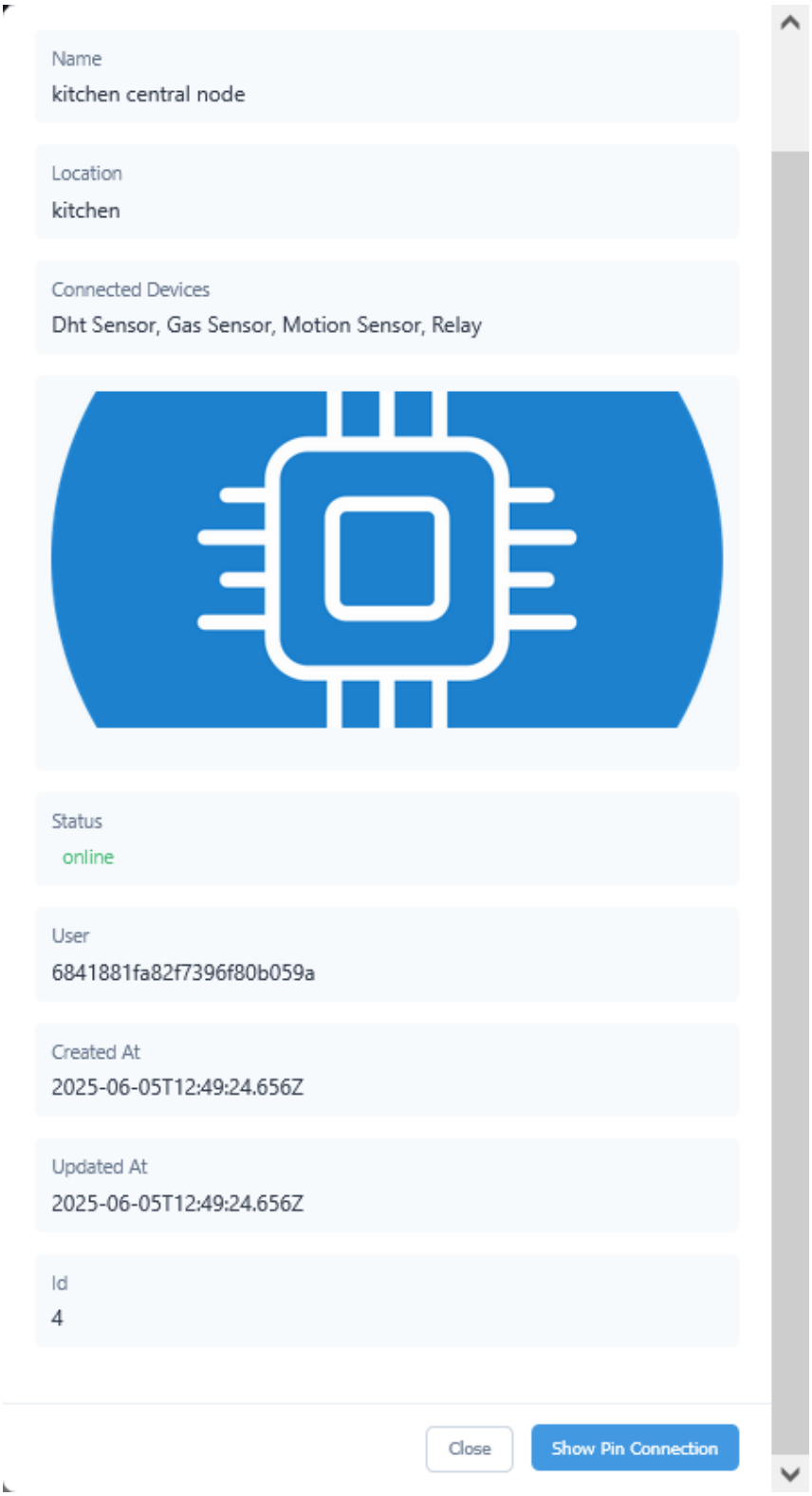


Figure 4.6: Device Details View in Electron App

The user (electrician) click on "**Show pin Connection"** to get instructions how to connect the peripherals (**dht, gas, motion sensors and relay) with the esp32,** which is **AI generated** and provided from the back end server, and the prompt template as follows :

```
function generateDeviceWiringPrompt(deviceName, pinsConnection, pinProperties) {
    const prettyJSON = (obj) => JSON.stringify(obj, null, 2);
  return `
You are working with an **ESP32-WROOM-32** microcontroller and the following peripheral
device: **"${deviceName}"**

Partial pin connections for the device:
\`\`\`json
${prettyJSON({ [deviceName]: pinsConnection })}
\`\`\`
All known pin properties of the device:
\`\`\`json
${prettyJSON(pinProperties)}
\`\`\`
🔧 Your task:
Generate a complete list of step-by-step instructions for wiring the **"${deviceName}"** to an
ESP32-WROOM-32 board.
📌 Rules & Tips:
- **Preserve** all existing connections.
- Use valid GPIO pins on the ESP32 (0–39). Avoid GPIOs 6–11 (used for flash) and GPIOs 34–39
(input-only).
- Assign safe and common GPIO defaults to unassigned required pins.
- Power pins (like VCC) should be connected to 3.3V unless specified otherwise.
- Optional pins can be connected if helpful, or left unconnected with a brief explanation.
- Do not include any JSON — just plain-text steps.

✅ Output format (example):
1. Connect VCC to 3.3V output of ESP32.
2. Connect GND to GND pin on ESP32.
3. Connect SDA to GPIO 21 (already assigned).
4. Connect SCL to GPIO 22 (already assigned).
5. Leave INT unconnected unless motion detection interrupt is needed.
6. Pull AD0 to GND to select I2C address 0x68.

Provide the steps in plain text format only.
  `.trim();
}
```

Then the generated **connection guidance** is displayed on the user interface, as follows in next pages

Connection details

Pin_connection

ESP32 DHT Sensor Wiring Instructions

1. Connect the DHT sensor's data pin to GPIO 4 on the ESP32 (already assigned).
2. Connect the DHT sensor's VCC pin to the 3.3V output of the ESP32.
3. Connect the DHT sensor's GND pin to any GND pin on the ESP32.

Additional Notes:
- The DHT sensor requires only three connections: power (VCC), ground (GND), and a data pin.
- The ESP32 can safely supply the 3.3V needed by the DHT sensor.
- If using a DHT22 sensor (the default type), it has better accuracy and range than the DHT11.
- A 10kΩ pull-up resistor between the data pin and VCC is recommended for more reliable communication, though many DHT modules already include this resistor.
- Make sure the total wire length between the ESP32 and DHT sensor is kept reasonably short to avoid signal degradation.

Gas Sensor Wiring Instructions for ESP32-WROOM-32

1. Connect the GND pin of the gas sensor to any GND pin on the ESP32.

2. Connect the VCC pin of the gas sensor to the 3.3V output pin on the ESP32.

3. Connect the AOUT (analog output) pin of the gas sensor to GPIO 34 on the ESP32, as already specified in your configuration.

4. Ensure the gas sensor is set to analog mode in your code since

4. Ensure the gas sensor is set to analog mode in your code since the analog pin is being used (this is the default setting).

5. The sensor is currently configured in simulation mode by default. When you're ready to use real sensor readings, update your code to set `simulate: false`.

Note: If you want to use the sensor in digital mode instead of analog mode, you would need to connect the DOUT pin of the gas sensor to GPIO 34 instead of the AOUT pin, and set the `analog` parameter to `false` in your code.
Motion Sensor Wiring Instructions for ESP32-WROOM-32

1. Connect the motion sensor's VCC pin to the 5V output of ESP32. (Most PIR motion sensors operate at 5V for optimal performance, though some can work with 3.3V)

2. Connect the motion sensor's GND pin to any GND pin on the ESP32.

3. Connect the motion sensor's PIR output pin (data pin) to GPIO 5 on the ESP32 (as already specified in your configuration).

4. If your motion sensor has a sensitivity adjustment potentiometer, you can adjust it as needed after installation.

5. If your motion sensor has a time delay adjustment potentiometer, you can adjust it to control how long the output stays HIGH after motion is detected.

Note: Make sure the ESP32 and motion sensor are powered off while making these connections. PIR motion sensors typically have only three pins (VCC, GND, and output), making them simple to connect. The sensor will output HIGH (1) when motion is detected and LOW (0) when no motion is detected.
Wiring Instructions for Relay Module to ESP32-WROOM-32

1. Connect VCC of the relay module to the 5V output pin of the ESP32. (Most relay modules require 5V for reliable operation, but

potentiometer, you can adjust it as needed after installation.

5. If your motion sensor has a time delay adjustment potentiometer, you can adjust it to control how long the output stays HIGH after motion is detected.

Note: Make sure the ESP32 and motion sensor are powered off while making these connections. PIR motion sensors typically have only three pins (VCC, GND, and output), making them simple to connect. The sensor will output HIGH (1) when motion is detected and LOW (0) when no motion is detected.
Wiring Instructions for Relay Module to ESP32-WROOM-32

1. Connect VCC of the relay module to the 5V output pin of the ESP32. (Most relay modules require 5V for reliable operation, but check your specific module's requirements)

2. Connect GND of the relay module to any GND pin on the ESP32.

3. Connect the relay control input pin to GPIO 2 on the ESP32 (already assigned).

4. If your relay module has an optocoupler or LED indicator, no additional connections are required as these are typically handled internally by the module.

5. If your relay module has a jumper for setting active high/low logic, ensure it matches the default "active_high" setting (true), or adjust the software accordingly.

6. For safety, ensure the high-voltage side of the relay (where you connect the device you want to control) is completely isolated from the ESP32 circuit.

Close

And here is the hardware connection result (sample not for production) :

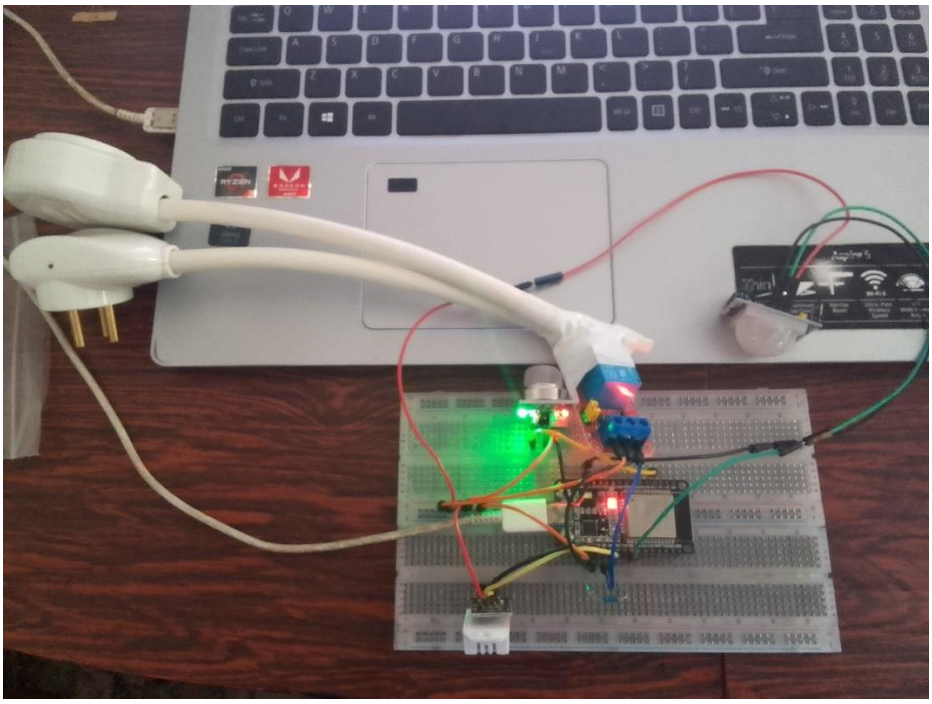


Figure 4.7: hardware connection example

4.3.4 Real-Time Monitoring

Once the IoT device is registered and installed with hardware connections, the user (customer) now can manage the IoT device and control/monitor the IoT device in real time from simplified and responsive web application user interface

Step 1: Accessing the Dashboard

After logging and got authenticated, the user open **dashboard** page to manage IoT devices

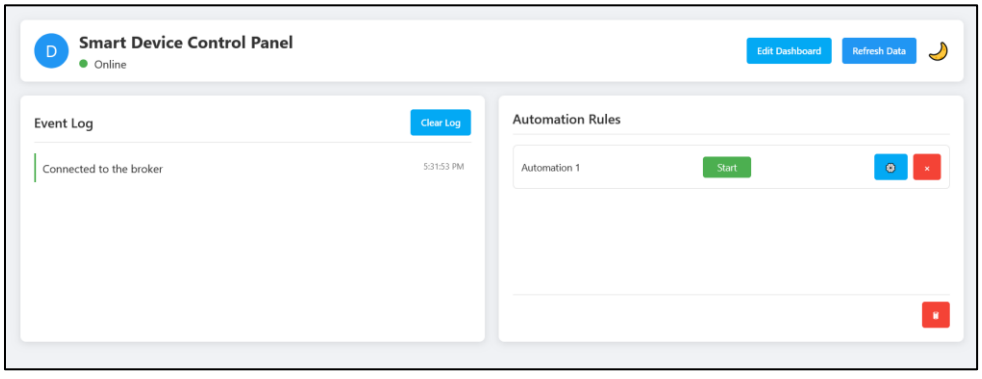


Figure 4.8: dashboard page

Step 2: User Adds a Monitoring Card

We need to add card on the dashboard to control/monitor the IoT device, which can be created dynamically according to the user specification. For we create a card to **monitor** the temperature

By clicking on **Edit Configuration** , and add **display component,** we have several types, and we choose **Guage** for example

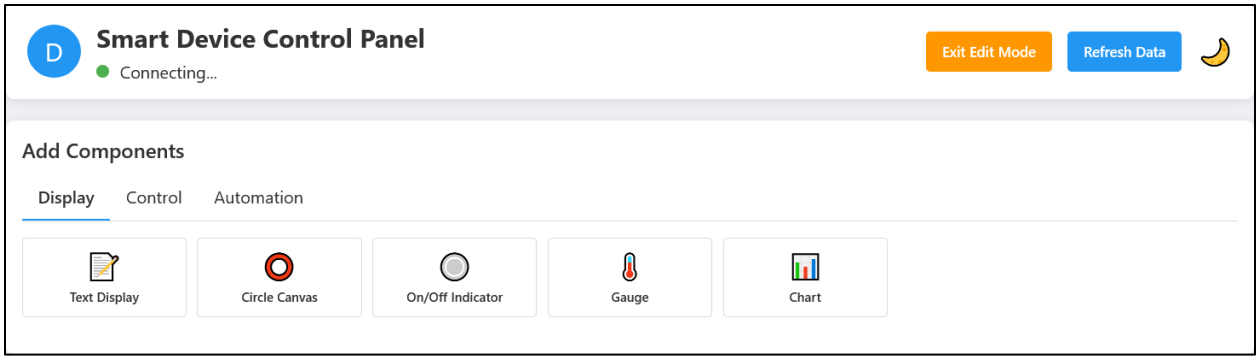


Figure 4.9: Dashboard choose type of display card

Then a configuration model is shown, to determine the IoT device, for example: :

- Microcontroller name : kitchen central node
- The peripheral : DHT sensor
- The method : read temperature

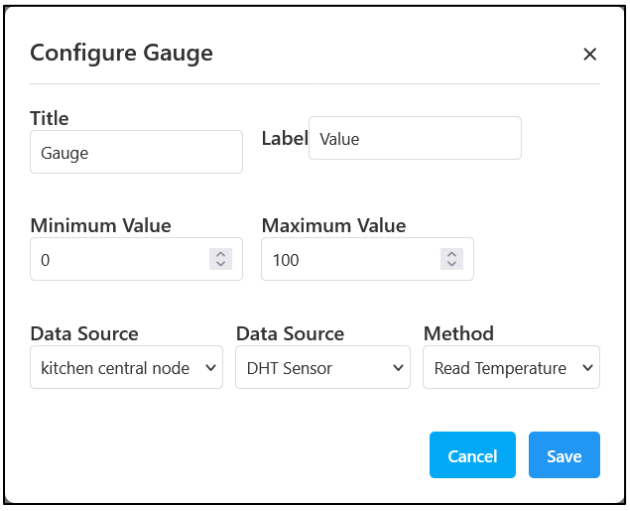


Figure 4.10: card configuration of temperature sensor

Step 2: Submitting the card to backend

After the fields are filled and the IoT device is determined, the user save the card to render on dashboard like follows :

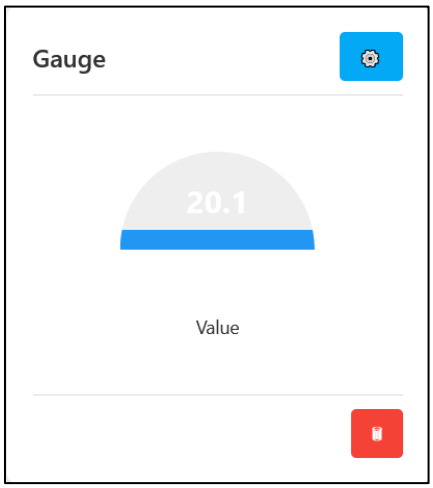


Figure 4.11: temperature sensor display card

Immediately the client (web application) sends periodic request to the back end server in real time socket connection, and these requests are forwarded to the IoT device to get the requested value (dht sensor – temperature), and here is the sample of request and the result below :

```
{
  "peripheral": "dht22",
  "method": "read_temperature",
  "param": "",
  "commandId": "abc123"
}
```

a)request

```
socket.emit("monitor_result",
{
  deviceId: "esp32-01",
  peripheral: "dht22",
  value: 20.1
})
```

b) result

Figure 4.12: monitor card request data and result

4.3.5 Real-Time Control card

Step 1: Add control card UI

similar to monitor card in card creation, but the card type is **control** , and the user choose which type of card input : text, switch button, slider, push button …

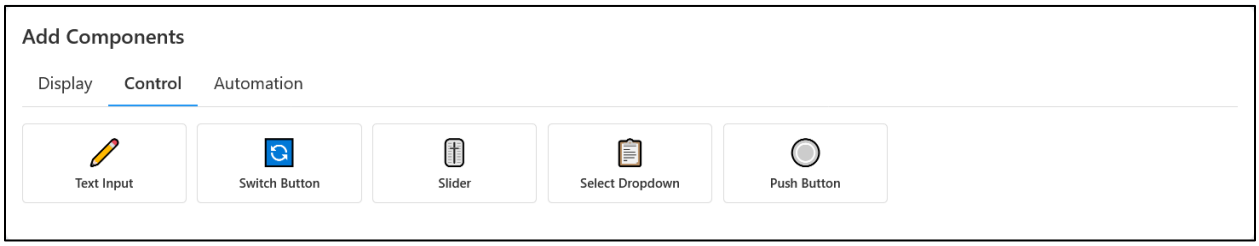


Figure 4.13: Dashboard choose type of control card

For example we need to add push button card, then the configuration model is displayed:

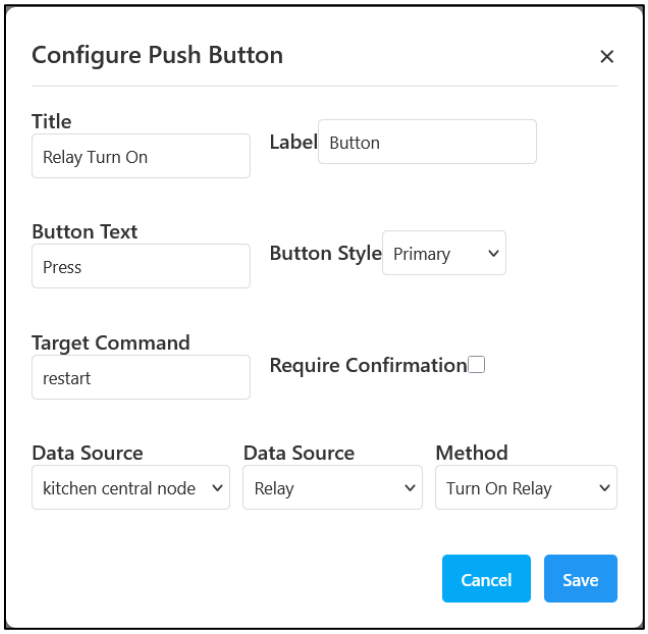


Figure 4.14: card configuration of push button

IoT device is determined in the fields selection:

Microcontroller: kitchen central node, peripheral : relay, method : Turn on Relay

After the card is saved, it is rendered on the **dashboard** with press button to control the relay

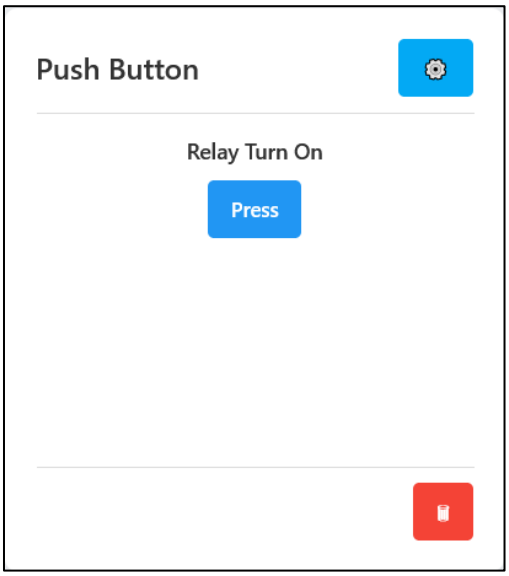


Figure 4.15: Push button card to control a relay

Step 2: send the command

When pressing the button, immediately the client (web application) sends command to back end server then forwarded to the IoT device (esp32 kitchen node – Relay) to turn on the relay:

- The request from client to the server:

```
{
  "peripheral": "relay",
  "method": "turn_on",
  "commandId": "def456"
}
```

- the execution in IoT device, using *Python method wrapper*

```
value = peripherals[msg['peripheral']][msg['method']][msg['param']]
```

which is similar to peripherals.relay.turn_on()

Step 3: receive the acknowledgement

after executing the command (turn on the relay), the esp32 return acknowledgement to the backend server, then returned to the client like the following JSON message :

```
{
  "peripheral": "relay",
  "method": "turn_on",
  "commandId": "def456"
  "value": "true"
}
```

Which we can verify in UI by adding monitoring card to display the relay status :

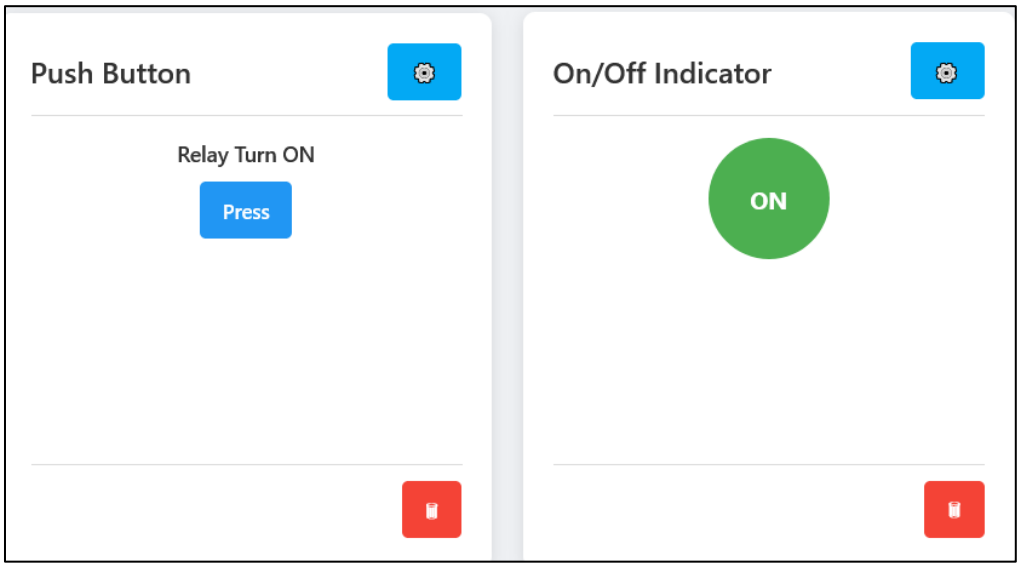


Figure 4.16: UI Feedback Showing Actuator State Updated

4.3.6 Automation Logic Configuration and Execution

The automation feature enables users (customers) to define conditional rules between the IoT devices. And this is done remotely from UI without hard coding the logic on esp32 code. to enable edge processing aims reduce the cloud dependency.

Step 1: Opening the Automation Configuration Interface

The user navigates to the **Devices Page**, clicks on the "**Automation**" tab, and creates a new automation rule.

Step 2: Filling the Automation Form

The user selects the input device, input method, condition type (gt, lt, or eq), threshold, output device and output method.

For example the explained logic in figure 4.17 :

- condition : if the temperature in DHT sensor that is connected to (esp32 kitchen central node) greater than 25
- execution : turn on the internal led of (esp32 living room node)

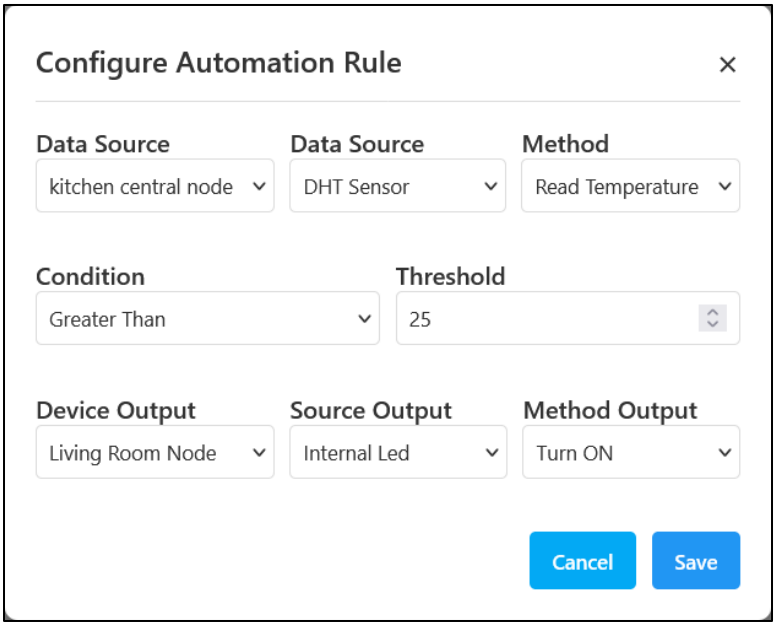


Figure 4.17: Automation Configuration Form

Step 3: Submitting the Automation Rule

After the user press **save,** the automation logic is sent to the backend server and forward it to the input device (esp32 kitchen central node), then the esp32 will save it to later send a trigger to the output IoT device when the condition happens

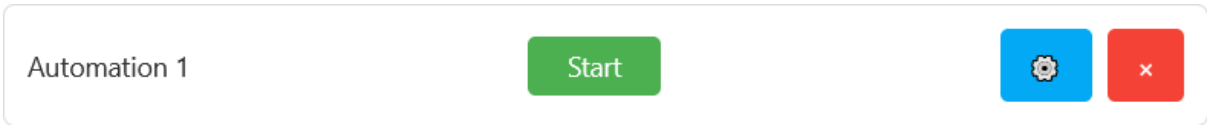
And here is the function that executes the automation logic in periodic loop the input IoT device :

```
async def runAutomation(automation):
    outputMsg = {}
    outputMsg['peripheral'] = automation['source-output']
    outputMsg['method'] = automation['method-output']
    outputMsg['param'] = automation['outputParams']
    outputMsg['commandId'] = 1

    outputDeviceId = automation['outputDeviceId']

    selectedPeripheral = automation['source']
    selectedMethod = automation['method']
    inputParams = automation['inputParams']

    if(automation['returnType'] == 'Number'):

        threshold = automation['threshold']
        if(cmp[automation['condition']][peripherals[selectedPeripheral][selectedMethod][inputP
arams], threshold]):
            await publishMqttAutomation(outputDeviceId, outputMsg)
    elif (automation['returnType'] == 'Boolean'):
        if(cmp['eq'][peripherals[selectedPeripheral][selectedMethod][inputParams],
automation['condition']]):
            await publishMqttAutomation(outputDeviceId, outputMsg)
    print(outputMsg)
```

Or without periodic check, by using **Interrupt** service routing, to run callback function that is specified in peripheral class:

```
if(msg.get('interrupt')): #if peripheral is motion sensor or push button
            peripherals[automation["source"]].set_callback(make_mqtt_cb(automation))
            return
```

Step 4: Visual Feedback in the Dashboard



To control the automation status (start, stop, remove, edit)

Automation Example

Here is simple implemented automation logic for both cases :

- periodic check DHT sensor to send the temperature to OLED to be displayed “dht sensor , 26” for example in the following picture
- interrupt (ISR) trigger from encoder, to write the degree on the servo motor

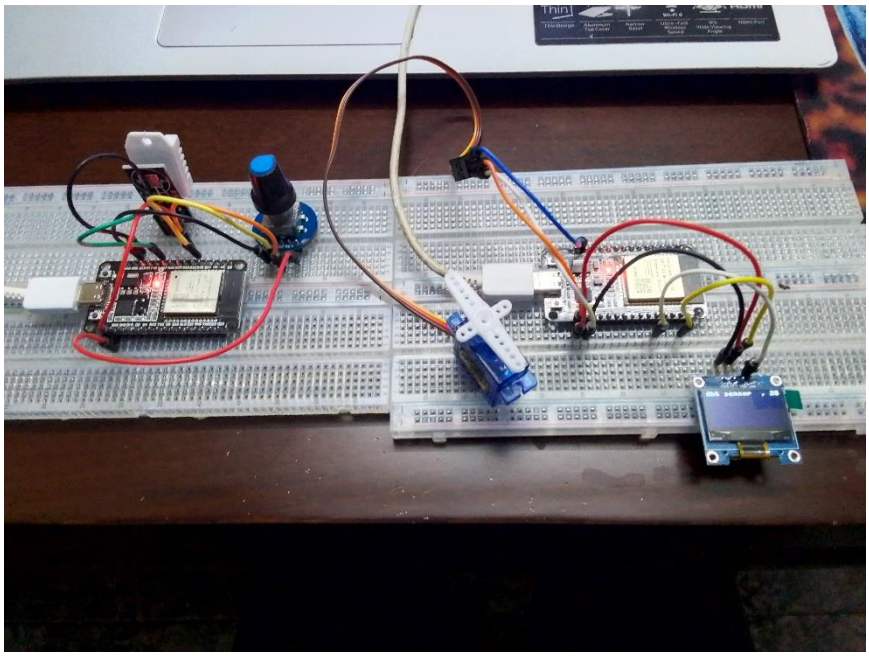


Figure 4.18: simple automation example

And check here to see the video of the example :

Chapter 5

5) Testing

5.1 preface

This chapter will implement the project on a comprehensive scenario , going throw the designed use cases for all types of peripherals included in chapter 2, to check how satisfies with the requirements: friendly user interface, user account management, simple IoT device registration, simple and correct hardware connection guidance, real time control and monitor for all types of peripherals, simple automation logic configuration without hardcoding.

5.2 the scenario

The scenario include 6 esp32 nodes, each of them connection with various type of peripherals which we told about in chapter 2. To show all possible cases implemented on the project smoothly, the test examines the user interface dealing with all use cases, at real time responsive.

The environment is Smart home has esp32s with its peripherals distributed to following rooms :

1. Kitchen
    a. Esp32 oven node : Gas sensor
    b. Esp32 central kitchen node : Motion sensor , Relay (light)
    c. Esp32 window node : Encoder , Servo motor
2. Living room
    a. Esp32 living room node : Oled , Buzzer , Push button
3. Door
    a. Esp32 door node : DHT sensor , Ultra sonic , relay
4. Human body
    a. Human bod node : Mpu6050

5.3 The connection guidance testing

To insure that the intsturctions given from AI is true :

Esp32 oven node

```
## Gas Sensor Connection
1. Connect the gas sensor signal pin to GPIO 13 on the ESP32.
2. Connect the gas sensor VCC to the 3.3V output of the ESP32.
3. Connect the gas sensor GND to any GND pin on the ESP32.
## Fan Connection
1. Connect the fan control pin to GPIO 27 on the ESP32.
2. Connect the fan power supply to an appropriate external power source (fans typically require 5V or 12V depending on the model).
3. Connect the fan GND to both the external power source GND and the ESP32 GND to ensure common ground.
4. If your fan requires PWM control, the GPIO 27 will provide the PWM signal.
```

The instructions are correct because:

1. GPIO 13 on ESP32 can read analog/digital signals, making it valid for the gas sensor output.
2. GPIO 27 is PWM-capable, so it can generate a control signal for the fan.
3. Using 3.3V for the gas sensor and a common ground between ESP32 and the external fan power ensures proper logic levels and circuit stability.

Esp32 central kitchen node

```
## Motion Sensor (PIN 13)
1. Connect the motion sensor signal/output pin to GPIO 13 on the ESP32.
2. Connect the motion sensor VCC/power pin to the 3.3V output on the ESP32.
3. Connect the motion sensor GND pin to any GND pin on the ESP32.

## Light (PIN 23)
1. Connect the light control pin to GPIO 23 on the ESP32.
2. If the light requires power (depending on type), connect its power input to an appropriate power source:
  - For an LED: Connect through a current-limiting resistor (220-330 ohms) to GPIO 23
  - For a relay-controlled light: Connect the relay control pin to GPIO 23
3. Connect the light's GND pin to any GND pin on the ESP32.
```

The instructions are correct because :

1. GPIO 13 supports digital input, making it suitable for receiving the motion sensor's HIGH/LOW output.
2. GPIO 23 can act as a digital output or PWM, so it can directly drive an LED (with resistor) or a relay for light control.
3. Using 3.3V for the sensor and sharing GND between the sensor, ESP32, and light ensures correct power and signal reference.

Esp32 window node

```
## Encoder Connection
1. Connect encoder pin1 to GPIO 25 on the ESP32.
2. Connect encoder pin2 to GPIO 26 on the ESP32.
3. Connect encoder GND to any GND pin on the ESP32.
4. Connect encoder VCC to 3.3V output of the ESP32.

## Servo Motor Connection
1. Connect servo motor pin1 (signal wire) to GPIO 13 on the ESP32.
2. Connect servo motor GND (typically black or brown wire) to any GND pin on the ESP32.
3. Connect servo motor VCC (typically red wire) to 5V output of the ESP32 (servos typically require 5V for proper operation).
4. Note: If your servo requires more current than the ESP32 can provide, consider using an external power supply and connecting only the GND and signal wire to the ESP32.
```

The instructions are correct because:

1. GPIO 25/26 on ESP32 support digital input and are valid pins for reading quadrature encoder signals.
2. GPIO 13 is a PWM-capable pin, so it can drive the servo motor's signal line.
3. GND and VCC lines follow the ESP32's standard power rails (3.3 V for encoder, 5 V for servo), ensuring correct logic and power levels.

Esp32 living room node

```
## Buzzer
1. Connect the positive (longer leg) of the Buzzer to GPIO 12 on the ESP32.
2. Connect the negative (shorter leg) of the Buzzer to any GND pin on the ESP32.

## OLED Display
```

1. Connect the SDA pin of the OLED display to GPIO 21 on the ESP32.
2. Connect the SCL pin of the OLED display to GPIO 22 on the ESP32.
3. Connect the VCC/VDD pin of the OLED display to the 3.3V output of the ESP32.
4. Connect the GND pin of the OLED display to any GND pin on the ESP32.
5. If the OLED has a RESET pin, you can leave it unconnected as most software implementations don't require it.

Push Button
1. Connect one terminal of the push button to GPIO 14 on the ESP32.
2. Connect the other terminal of the push button to GND on the ESP32.
3. Add a 10kΩ pull-up resistor between GPIO 14 and 3.3V to ensure a stable HIGH state when the button is not pressed.

The instructions are correct because:

1. GPIO 12 supports digital/PWM output, making it suitable for driving a buzzer.
2. GPIO 21 (SDA) and GPIO 22 (SCL) are the ESP32's default I²C pins, ensuring proper communication with the OLED display.
3. GPIO 14 with a pull-up resistor provides a clean HIGH/LOW input state, making it reliable for the push button.

Esp32 door node

DHT Sensor Connections
1. Connect DHT sensor data pin to GPIO 4 (already assigned)
2. Connect DHT sensor VCC to 3.3V output of ESP32
3. Connect DHT sensor GND to any GND pin on ESP32

Ultrasonic Sensor Connections
4. Connect ultrasonic sensor TRIG pin to GPIO 5 (already assigned)
5. Connect ultrasonic sensor ECHO pin to GPIO 18 (already assigned)
6. Connect ultrasonic sensor VCC to 5V output of ESP32 (most ultrasonic sensors need 5V for reliable operation)
7. Connect ultrasonic sensor GND to any GND pin on ESP32

Light Connections
8. Connect light control pin to GPIO 12 (already assigned)
9. Connect light GND to any GND pin on ESP32
10. If the light requires power (like an LED), connect it to a current-limiting resistor (220-330 ohm) and then to GPIO 12

Internal LED Connection
11. The internal LED is already connected to GPIO 2 on the ESP32 board (this is built into the ESP32 board and doesn't require external wiring)

General Notes
- Make sure all GND connections can be connected to any of the GND pins on the ESP32 board
- Double-check that all power requirements are met (especially for the ultrasonic sensor)
- Keep wire lengths reasonably short to minimize interference, especially for the DHT sensor's data line

The instructions are correct because:

1. GPIO 4, 5, and 18 are valid digital I/O pins, making them suitable for the DHT sensor (data), and ultrasonic sensor (TRIG/ECHO).
2. GPIO 12 supports digital/PWM output for controlling a light, while GPIO 2 is the onboard LED, already wired internally.
3. Power and ground assignments follow ESP32 standards: 3.3V for the DHT, 5V for the ultrasonic sensor, and common GND for stable operation.

Esp32 human body node (mpu6050)

Accelerometer Connections:
1. Connect SDA (Serial Data Line) of the accelerometer to GPIO 21 on the ESP32.
2. Connect SCL (Serial Clock Line) of the accelerometer to GPIO 22 on the ESP32.
3. Connect VCC of the accelerometer to the 3.3V output pin on the ESP32.
4. Connect GND of the accelerometer to any GND pin on the ESP32.
5. If the accelerometer has an INT (Interrupt) pin, it can be left unconnected unless you need motion detection interrupts in your application.
Notes:
- The I2C bus (SDA/SCL) used for the accelerometer can be shared with other I2C devices if you add more components later.
- Make sure all connections are secure and that no wires are loose or shorting against each other.
- Double-check the VCC requirements of your specific accelerometer - most work with 3.3V but verify in your component's datasheet.

The instructions are correct because:

1. GPIO 21 (SDA) and GPIO 22 (SCL) are the ESP32's default I²C pins, ensuring proper communication with the accelerometer.

2. Powering the accelerometer with 3.3V and connecting GND to the ESP32 GND matches its standard operating requirements.
3. The optional INT pin is not required unless interrupts are needed, so leaving it unconnected is valid and safe.

5.4 The User Interface Testing

The user interface should be simplified with high user experience, and responsive working on cross platform through (pc devices , phone devices …).

Responsive UI

The frontend pages can increase and decrease according to customer screen size

Minimum 360 pixel :

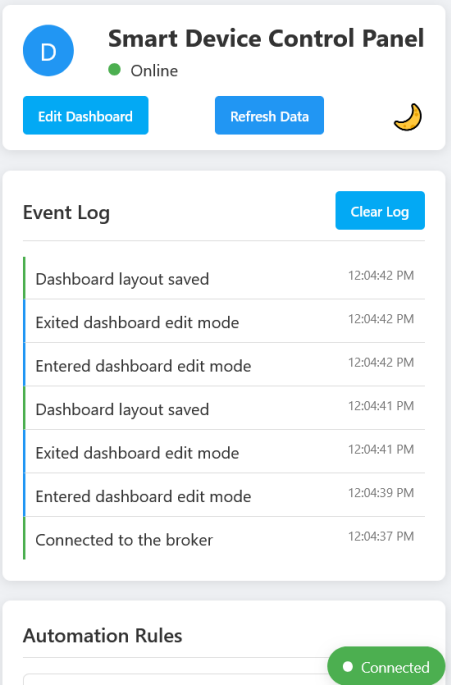


Figure 5.1: responsive UI at 360 pixel

to wide screen interfaces :

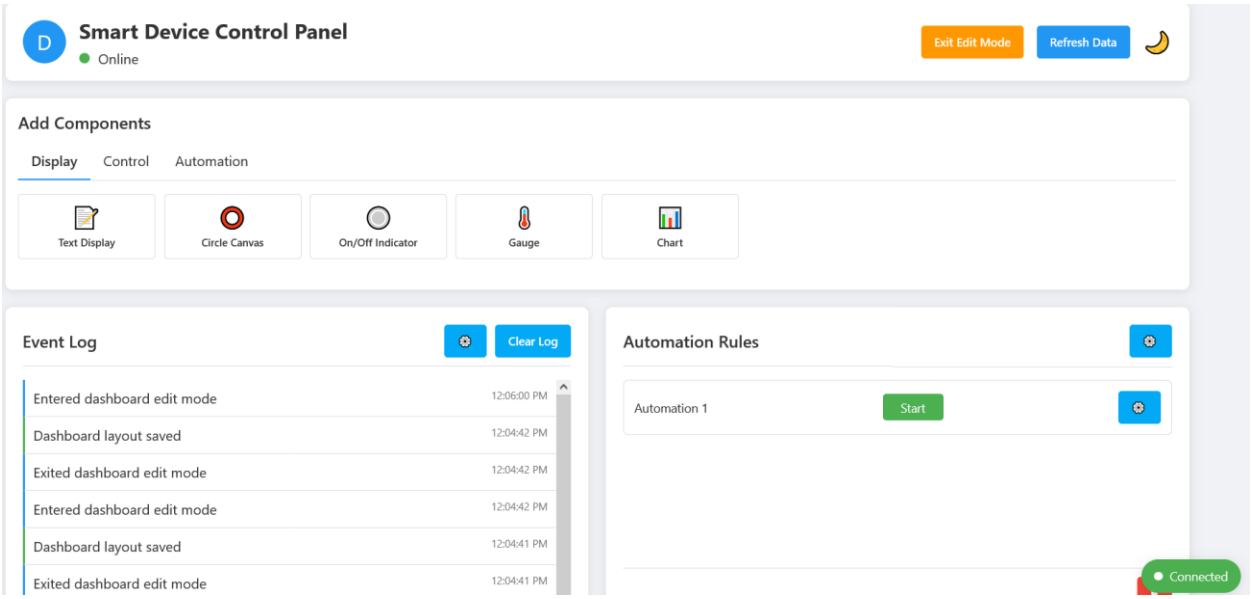


Figure 5.2: responsive website at wide screen

Chapter 6

6) Conclusion and Future Work

The goal of this project was to make IoT smart home systems more accessible and user friendly, especially for individuals have no programming experience, such as electricians and technicians. Some IoT platforms provide easy setup for IoT devices, but do not provide customization, for example a complete IoT Fan and it's application. Our system solves this by using AI code generation, clear wiring instructions, and an intuitive interface into a single system that simplifies the entire process through real time monitoring, controlling and automation.

Through this project, it was possible to design a system where users can:

- Sign up IoT devices and peripherals without writing code.
- Automatically install AI firmware generated into ESP32 microcontrollers.
- Utilize step-by-step guidelines to correctly connect hardware parts.
- Stream data and control actuators in real time with a responsive dashboard.
- Establish simple automation rules to allow devices to operate automatically together.

The result was that multi-sensor devices and automation logic could be set up in a matter of minutes, even by someone without technical knowledge. While the current implementation works in a local environment with some limitations—such as plain text-based wiring commands and limited device memory—it provides a glimpse of how technical limitations can be allevia

For the future, there are several ways that this project can be made better and expanded:

- ***Intelligent Deployment Decisions***

  Instead of relying on a locally installed MQTT broker, future versions could have both cloud brokers for global accessibility and edge brokers that may even be auto-installed on miniature devices like Raspberry Pi. This would make the system more flexible and reliable across different [7]–[10]

- ***More Advanced AI Code Generation***

  Nowadays, AI creates firmware. Future development in this direction can be taken to create libraries automatically, so new sensors and actuators will be covered by the system without specifying predefined libraries. AI would optimize the code generated to better accommodate device[11-12]

- ***Better Connection Guidance***

  Instead of plain text, the system could offer a graphical and animated interface that displays visually how peripherals can be connected. Animations step-by-step would make the installation process easier to comprehend, reduce errors, and provide the user with an improved experience.[13]

- ***User Roles and Collaboration***

  The system currently offers basic account management. It can, in the future, offer different roles (e.g., administrators, technicians, and end-users) and joint access to a device with variable permissions, which would make it more relevant in homes, schools, or businesses.[14-16]

- ***Scalability and Cloud Integration***

  By having the backend and database in the cloud, bigger deployments would be better supported by the system. Users would even be able to register and monitor devices remotely, beyond the local network.[17-19]

Lastly, this project provided a working proof of concept for how AI and modern IoT software can be blended to overcome technical hurdles and bring smart-home tech within reach. Further development could, in theory, turn it into a fully scalable, user-friendly platform that bridges the gap between advanced IoT capability and real-world

7) References

[1] A. Waqar, M. A. Qureshi, and M. Muneer, “Assessment of Challenges to the Adoption of IoT for Small Construction Projects,” *Applied Sciences*, vol. 13, no. 5, p. 3340, Mar. 2023.

[2] “Internet of Things (IoT) Adoption: Challenges and Barriers,” ResearchGate, Mar. 2023.

[3] F. Z. Hannou, “A Survey on IoT Programming Platforms,” *ACM*, 2024.

[4] Zoho Corporation, “Zoho Launches Low-Code IoT Platform with Industry-Specific Solutions,” *Futurum Group*, Oct. 2024

[5] “Skills shortage remains the main barrier to IoT adoption,” Digitalisation World, Mar. 2022.

[6] K. Swathi, T. Uday Sandeep, and A. Roja Ramani, “Performance Analysis of Microcontrollers Used In IoT Technology,” *International Journal of Scientific Research in Science, Engineering and Technology*, vol. 4, no. 4, pp. 1268-1273, Apr. 2018.

[7] T. Tseng, “Cloud-edge MQTT messaging for latency mitigation and fog-enhanced IoT,” *PeerJ Computer Science*, vol. 11, p. e2741, Jan. 2025. [Online]. Available: https://peerj.com/articles/cs-2741/

[8] A. Prazeres, R. R. Barbosa, and J. J. P. C. Rodrigues, “A comparison of MQTT brokers for distributed IoT edge computing,” *Proc. Int. Conf. on Computer Communications and Networks*, 2020. [Online]. Available: https://www.researchgate.net/publication/344273979

[9] C. H. Hong et al., “Navigating the edge-cloud continuum: A state-of-practice,” *arXiv preprint*, arXiv:2506.02003, 2025. [Online]. Available: https://arxiv.org/html/2506.02003v1

[10] T. C. Huang et al., “Decentralized edge-to-cloud load-balancing: Service placement for the Internet of Things,” *arXiv preprint*, arXiv:2005.00270, 2020. [Online]. Available: https://arxiv.org/abs/2005.00270

[11] Y. Bi et al., “A survey on artificial intelligence aided Internet-of-Things,” *Future Internet*, vol. 14, no. 5, p. 152, 2022. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC9029361/

[12] L. Zhang et al., “Gensors: Authoring personalized visual sensors with a hardware-agnostic IoT platform,” *arXiv preprint*, arXiv:2501.15727, 2025. [Online]. Available: https://arxiv.org/html/2501.15727v1

[13] M. Ardito, G. Desolda, and M. Matera, “User-defined semantics for the design of IoT systems,” *Personal and Ubiquitous Computing*, vol. 24, pp. 847–863, 2020. [Online]. Available: https://link.springer.com/article/10.1007/s00779-020-01457-5

[14] R. Singh, R. Rani, and N. Kumar, “Role-based access control (RBAC) enabled secure and efficient data processing framework for IoT networks,” *Proc. Int. Conf. on Computational Intelligence and Security*, 2024. [Online]. Available: https://www.researchgate.net/publication/383295659

[15] R. Ragothaman et al., “Access control for IoT: A survey of existing research,” *Sensors*, vol. 23, no. 2, p. 340, 2023. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC9963042/

[16] P. Anand et al., “Policy-driven access control in large-scale IoT networks: Integrating RBAC with ABAC,” *SSRN preprint*, 2024. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5248797

[17] D. Obelisk et al., “Design and evaluation of a scalable Internet of Things multi-tenant platform,” *Software: Practice and Experience*, vol. 52, no. 10, pp. 2049–2072, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2973

[18] S. R. Abhay, “Scalable IoT solutions with cloud infrastructure,” *PhilArchive*, 2024. [Online]. Available: https://philarchive.org/archive/RONSIS-2

[19] A. Ahmad et al., “Leveraging IoT, cloud, and edge computing with AI,” *Sensors*, vol. 25, no. 6, p. 1763, Mar. 2025. [Online]. Available: https://www.mdpi.com/1424-8220/25/6/1763