International Journal of Advanced Technology and Engineering Exploration, Vol 12(124)
ISSN (Print): 2394-5443 ISSN (Online): 2394-7454
http://dx.doi.org/10.19101/IJATEE.2024.111101934

Research Article

Design and implementation of a REST API-based client-server architecture for multi-sensor IoT monitoring

Giva Andriana Mutiara*, Periyadi, Muhammad Rizqy Alfarisi, Muhammad Aulia Rifqi Zain, Muhammad Ghifar Rijali and Fathurrohman Nur Rochim
Department of Applied Science School, Universitas Telkom, Jawa Barat, Indonesia

Received: 24-October-2024; Revised: 20-March-2025; Accepted: 22-March-2025

Abstract
This paper discusses the design and implementation of a representational state transfer (REST) application programming interface (API)-based client-server architecture for an internet of things (IoT) monitoring application with multiple sensors. The research aims to enable real-time monitoring of an integrated system consisting of more than ten sensors, utilizing ESP-32 and long-range (LoRa) communication. The monitoring system focuses on developing a web-based interface that allows remote access to real-time data changes, without detailing sensor functionality or energy consumption. The system employs a REST API for data transmission. Sensor data is collected by the ESP-32, transmitted over a long-range wide-area network (LoRaWAN), processed by the LoRa receiver, and then relayed via Wi-Fi to the API. The architecture follows an N-layer design, facilitating client-server communication, database synchronization, and data provisioning for the web interface. Performance testing was conducted on two nodes, each containing multiple sensors, across various scenarios to evaluate system efficiency, load capacity, security, and dashboard functionality. Results show that the API response time for data retrieval from the two nodes ranged between 200 and 400 milliseconds. System performance begins to degrade beyond 1,500 users, with an observed error rate of 1.72% for node-1 and 0.9% for node-2. Security tests confirm the system's resistance to various security threats. Dashboard monitoring results indicate that data can be displayed with a latency of less than one second.

Keywords
REST API, Internet of things (IoT), Client-server architecture, Real-time monitoring, LoRa communication, Multi-sensor system.

1.Introduction
The Internet of Things (IoT) is a technological innovation that has advanced significantly in recent years. Multiple domains of human existence undergo significant transformations when integrated with IoT. The IoT facilitates the interconnection of devices to gather and transmit data over the internet. The implementation of IoT significantly enhances accessibility to electronic devices.

IoT technology significantly enhances various aspects of human life. The domestic sector comprises many smart gadgets, including lighting, doors, and appliances, which may be managed by smartphones. Smartphones can be used to monitor cattle health and land conditions in the agriculture [1] and livestock sectors [2].

In the healthcare sector [3], health devices facilitate real-time patient monitoring and expedite responses to health issues. The introduction of the Industry 4.0 idea in the industrial sector, which depends on IoT to enable communication between production systems and machines to increase efficiency.

One of the main aspects of IoT implementation in life lies in the ability of an application to effectively monitor the system [4]. The monitoring process in IoT is closely related to the number of sensors used in the monitoring process. The more sensors used for monitoring, the more parameters that must be displayed in IoT-based monitoring applications.

IoT-based monitoring systems are intrinsically linked to the utilization of networks as a communication medium. The performance advantage of IoT-based systems is due to real-time and continuous data collection [5]. For instance, IoT-based security

*Author for correspondence

systems are used to detect anomalies or monitor specific parameters over designated timeframes, generating essential data to support user decision-making, analysis, and future strategic planning [6]. Consequently, ensuring the continuation of data transmission presents a barrier for numerous researchers developing IoT-based applications [7–9]. Therefore, the continuity of data transmission becomes a challenge, especially with monitoring applications with hardware in remote places where there is no Wi-Fi.

Moreover, website monitoring systems play a crucial role in the monitoring process [10], serving as an interface between the system and the user. The primary purpose of website monitoring is to provide users with data on performance, functionality, speed, and other key metrics to ensure efficient and responsive operation. This helps prevent potential disruptions during user interactions and maintains the integrity of data transmission connections [11]. This is because this monitoring system can allow early detection of equipment damage and errors to prevent substantial issues.

Another aspect that needs to be considered as a challenge in developing monitoring application is the application system architecture. Many applications are built using a monolithic architecture as a foundation for developing easy monitoring applications. However, despite its simplicity in application device construction, this architecture possesses numerous vulnerabilities, particularly concerning adaptability, scalability, flexibility, and the capacity to manage intricate and dynamic systems [12]. Consequently, a monolithic design is unsuitable for a multi-sensor-based monitoring application.

Meanwhile, there are some other specific types of architectures used for IoT-based systems that utilize multiple sensors including n-layered architecture (N-Tier) [13] in conjunction with microservice architecture [14] or event-driven architecture (EDA) [15]. These three architectures exhibit distinct layer separation, facilitate scalability, and are responsive to state alterations. Moreover, edge computing architectures are frequently employed in multisensory-based IoT applications, necessitating data processing near the source at the gateway level to minimize latency, bandwidth consumption, and cloud load. Despite the numerous advantages of the three architectural styles regarding scalability, flexibility, and the capacity to manage intricate and dynamic systems, their complexity renders them suitable for application in multisensory-based systems [16–18].

However, in the case of development of application systems, representational state transfer (REST) application programming interface (API) architecture is frequently employed, facilitating the establishment of network connections between nodes and users on a client-server framework [19]. Sensors linked to edge devices gather data and transmit it to the server over an API. The server functions as a hub that accepts data from several sensors via REST API endpoints, processes the information, stores it in a database, and facilitates access to the processed data for other applications. There is a necessity for mobile, online, or desktop applications that retrieve data from the server over the REST API to present information, perform analysis or manage IoT devices.

This research is motivated by the necessity for smart sensor-based systems that can serve as a basis for developing architectures for extensive monitoring applications that facilitate, help and support the development and the implementation of smart cities. In the future, sensors will take over humans in monitoring processes and alterations that may occur during a specific duration or continuously. Consequently, the deployment of a sensor at an observation site may augment in quantity or capability.

Thus, this study is to present a simple REST API architecture that incorporates a client-server monitoring application design and implementation framework coupled with n-layer architecture. The client-server architecture is employed due to its clarity in the arrangement of communication components within each client and server, the methodology of data transmission between them, and its straightforward implementation in contemporary online applications, mobile applications, and corporate software. The integration of client-server design with n-layer architecture distinctly establishes the separation of frontend and backend, facilitating enhanced scalability.

The application website architecture employs a REST API, serving as an interface to concurrently receive significant sensor data (thousands of entries). This website architecture seeks to provide a solution for creating a streamlined and systematic monitoring system application capable of overseeing multisensory networks linked to IoT technology. According to this fact, this research can serve as a

multisensory system observation architecture that enhances scalability, adaptability, and the ability to handle complex and dynamic systems, while ensuring maintenance efficiency on the monitoring dashboard development side. Thus, this paper does not explain in detail the deployment, accuracy, data preprocessing, or energy consumed by the sensor. This research advances the development of smart city applications necessitating numerous monitoring nodes across several platforms.

The rest of the paper is structured as follows: the literature review is presented in Section 2, followed by the research methodology in Section 3. The results and discussion are covered in Sections 4 and 5, respectively. Finally, conclusions and future work are discussed in Section 6.

2.Literature review

This section provides a comprehensive literature review, analyzing various studies on IoT-based application architectures. Numerous studies explore how sensors collect, process, and transmit data to servers or cloud platforms for further analysis. To ensure efficiency and compliance with established guidelines, application design must follow an appropriate architectural framework.

Various software architecture methodologies used in application development include monolithic architecture [20, 21], microservices architecture [22], service-oriented architecture (SOA) [23], layered (N-Tier) architecture [24, 25], EDA [26], serverless architecture [27, 28], microkernel (plug-in) architecture [29], peer-to-peer (P2P) architecture [30], client-server architecture [31], and component-based architecture [32, 33]. For a monitoring application utilizing IoT technology for various environmental sensors, microservices architecture, layered (N-Tier) architecture, EDA, and client-server architecture are the most appropriate for IoT-based observation applications and sensor integration [34].

The most basic architecture for multi-sensor applications is the client-server model. This architecture positions a server as the central entity for processing and storing data gathered from multiple sensors. This server may be deployed in the cloud or within a local data center. An edge device or gateway is positioned on the client side, functioning as a client that transmits data from the sensors to the server. It may also refer to a mobile or online application utilized for accessing data from the server [35, 36]. These architectures include cloud-based IoT architecture, edge computing architecture, fog computing architecture, gateway-based architecture, and client-server architecture for sensor applications, as illustrated in *Figure 1*.

The illustration shows that the edge position is close to the sensing device, functioning as a client. This edge architecture is typically implemented to minimize latency, improve response speed, and reduce network strain by processing data closer to the source device [37]. In a distributed system, fog architecture follows edge architecture, where data is processed on a router, switch, or a more robust local server than edge gateways, enabling better processing or data aggregation [38]. This fog architecture typically serves as an intermediary layer between the edge and the cloud. Fog architecture may or may not be used, depending on the system's scalability requirements. In cloud architecture, the cloud is typically located far from the data source, serving as a data center for long-term storage [39].

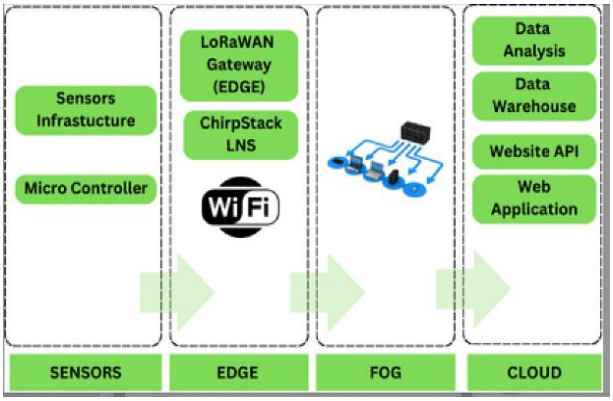


Figure 1 IoT architecture, the hierarchy of edge, fog, and cloud computing

In client-server architecture, processing is allocated between the client and the server; typically, the client manages data presentation as user interface and requests, whereas the server oversees data processing and resource management. Consequently, client-server independence may also be present across many networked devices. This client-server architecture enables the server to simultaneously accommodate several clients through the utilization of various sensors for environmental observation. The server can be scaled by augmenting hardware, such as sensors, or by distributing the load over numerous servers (load balancing) [40]. The client-server architecture for multisensory systems is executed using more intricate models, such as n-tier architecture or microservices architecture. The

establishment of this n-tier paradigm facilitates designers in delineating component duties with greater specificity to accommodate more intricate applications, wherein data gathered from diverse sensors must be processed, analyzed, and managed independently before reaching the client or user [41].

To construct distributed and interoperable web-based applications, a common approach in client-server architecture is the utilization of REST API as the communication channel between the client and server. This enables clients, such mobile applications, web platforms, or IoT devices, to interact with the server through conventional hypertext transfer protocol (HTTP) queries. Conversely, traditional client-server technologies like the simple object access protocol (SOAP) are predominantly utilized at the enterprise level, while graph query language (GraphQL) is occasionally employed in IoT applications, though with restricted bandwidth computing that permits clients to request just particular data [42]. Furthermore, the google remote procedure call (gRPC) framework and websockets provide high-speed client-server communication through binary protocols, enabling continuous bidirectional data transmission between clients and servers; however, the infrastructure is more complicated [43]. Consequently, REST API facilitates numerous IoT applications, as the majority of IoT devices inherently support HTTP as a communication protocol, operate stateless for each request, and can be processed independently without relying on the status of prior requests, thereby significantly enhancing scalability and simplifying communication. Furthermore, the REST API aligns with the functionality of sensors as create, read, update, delete (CRUD) which facilitating basic data manipulation [44]. REST API can be integrated with other protocols like message queueing telemetry transport (MQTT), which offers enhanced communication efficiency for IoT devices with limited bandwidth and power limitations [45].

In *Figure 2*, it can be seen that communication between client and server uses the standard HTTP protocol method with get, post, put and delete commands. This architecture is often used because it is flexible and easy to implement [46].

In *Figure 2*, sensors linked to edge devices or gateways function as clients that gather data and transmit it to the server over REST APIs. The server functions as a central hub that acquires data from various sensors using REST API endpoints, processes the information, stores it in a database, and enables access to the processed data for other applications. Mobile, internet, or desktop applications function as clients by retrieving data from the server over REST APIs to provide information, do analysis, or control IoT devices.

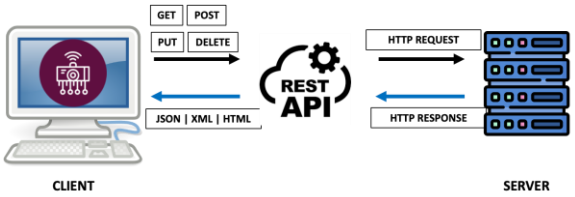


Figure 2 REST API on client-server architecture

In reference to Figure 1, the client-to-edge interaction may manifest as a client activity (e.g., a mobile or web application) transmitting an HTTP request to edge devices to retrieve local information, including device status or sensor data. The edge devices reply through REST APIs with locally processed data. The edge device thereafter interacts with the fog infrastructure to transmit data for further processing, utilizing the REST API for communication with the fog nodes. The fog layer executes supplementary processing, like data aggregation or filtering, and might temporarily store the results prior to transmitting them to the cloud. Subsequently, fog nodes interact with the cloud layer via REST API to transmit processed data for long-term storage or advanced analytics. The cloud layer offers a sophisticated and robust REST API interface for storage, extensive analytics, or services rendered to clients and fog layers. Simultaneously, other clients (e.g., analytical programs or dashboards) can directly access the cloud using REST APIs to retrieve data or execute actions based on cloud analytical outcomes. Consequently, the client-server interaction facilitates a monitoring process for IoT-based multisensory observation applications, connecting several sensors to a web-based application dashboard.

The performance of data transmission over the REST API necessitates an endpoint. This endpoint is publicly accessible, rendering it susceptible to data injection and leaking [47]. Numerous security measures are established inside the client-server architecture. Examples include transport layer security/secure sockets layer (TLS/SSL), authentication, authorization, data encryption, rate limiting and throttling, input validation and sanitization, logging and monitoring, API gateway for enhanced security, session management for REST APIs, and device authentication for sensors [48].

Security measures in the client-server architecture are effective only if fully implemented within the system. Besides focusing on API development, this research also focuses on website development. Where using nodeJS as a backend system and reactJS as a frontend system. In research [49], a nodeJS-REST API-based backend was constructed, demonstrating favorable response time outcomes. The system can accommodate around 3000 queries within a duration of 10 seconds. Subsequently research [50], examines the evolution of reactJS, which is utilized in this study. JavaScript remains extensively utilized in the programming domain. JavaScript can be utilized in multiple frameworks, including node.js, react, react native [51], Vue.js [52], and numerous additional applications that can be developed using JavaScript.

However, to support stateless functions in REST API, security is implemented using the JavaScript Object Notation (JSON) with a web token approach. JSON web token (JWT) is often used in current authentication and authorization frameworks because of its simplicity and capacity for secure transmission via uniform resource locators (URL), HTTP header, or URL parameters. In addition, JWT is independent, highly interoperable, adaptable, and has significant scalability [53]. In addition, JWT is easy to use in REST API and allows decentralized access. The token usage in JWT should have an expiration time limit (e.g., 1 hour). This reduces the risk of misuse of leaked or stolen tokens. Once a client or sensor is successfully authenticated, they receive a token containing their identity and authorization information. This token must be included in every API request.

The security procedure involves obtaining an access token subsequent to logging into an account at the authentication endpoint, as illustrated in *Figure 3*. The authentication procedure involves entering an account using the email and password fields for sign-in, and the username, email, and password fields for sign-up. Upon authentication, an access token is issued for data retrieval from the endpoint [54].

Due to technological advancement, researchers are progressively developing sensor-based automation systems to observe and measure objects or environments, hence supporting modernization and smart city applications. Sensor-based monitoring requires various technologies that support each other to collect, transmit, store, process, and display data efficiently. This monitoring system must be constructed using a technological framework comprising a collection of sensors or actuators designed to detect the parameters under observation. IoT devices, including microcontrollers, central processing unit (CPU), and ESP32, frequently act as primary controllers that manage sensor data and transfer it to the server using wireless communication protocols such as Wi-Fi, long range wide area network (LoRaWAN), and MQTT [55].

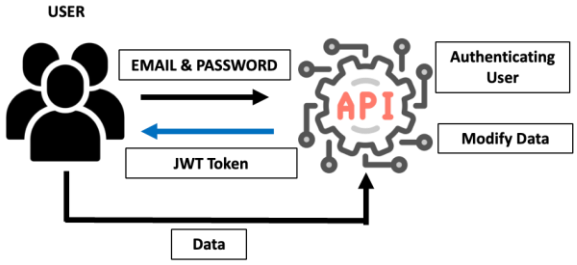


Figure 3 API security with JWT token

In addition, data storage strategies, such my structured query language (MySQL) or cloud services, and analytical data processing techniques, such as artificial intelligence, machine learning, or deep learning also requires building the system [56]. Subsequently, the interface platform, such as the Thingspeak platform or a web-based device interface is implemented to support the system. A proficient application system should have notifications or alarms for early warning detection. Finally, a device is required for the integration of all components, including the REST API and other services.

This research involves the design and development of a REST API-based client-server architecture for IoT applications utilizing multi-sensors within a single node. According to the previous literature review, the monitoring system is executed using a client-server architecture due to its simplicity in establishing an organized workflow from the sensor to the monitoring interface. The Rest API is utilized for its simplicity and statelessness, making it highly ideal for IoT-based applications.

Meanwhile, the sensors deployed to monitor the environment in a specific room comprise ten distinct types, each designed for various measurement and detection functions. These include the MQ2 gas sensor, KY-037 noise sensor, TF Luna Lidar sensor for distance measurement, ADXL-345 sensor for mobility and positional orientation, load cell sensor for weight detection, MPU6050 sensor for gyroscopic and accelerometric parameters, FR-7548 sensor for pressure measurement, thermal camera sensor, and

BME sensor for temperature, humidity, and additional parameters.

This study does not focus on individual sensor applications, as they are considered a collective data source within a multi-sensor environment. Instead, it emphasizes the development of the monitoring dashboard and its connection to the placement of the REST API application within the client-server architecture that links the sensors to the monitoring dashboard.

Thus, this research is presented as a client-server architecture incorporating an N-layered mapping infrastructure to support multi-sensor utilization within a single node. The monitoring solution is web-based, accessible on both mobile and desktop platforms, and employs JWToken for security. This system aims to enhance and contribute to the development of a versatile platform for diverse multi-sensor monitoring applications.

3.Methods

This section provides a detailed explanation of the research and methodologies. This study focuses on the development and implementation of REST APIs for managing IoT applications with multiple sensors, rather than on the sensors themselves. Therefore, the most appropriate methodology for this investigation is software engineering research methodology. The suitable method for this study is design and implementation research, utilizing the relevant prototyping approach.

The initial phases of this prototyping model begin with identifying API requirements, during which the needs of the IoT system intended to connect to the REST API are defined. This is followed by the initial stage of API design, where API endpoints are structured to facilitate communication among sensor devices, servers, and clients. Additionally, during the prototyping stage, the first version of the REST API is developed using the chosen framework, Node.js.

After development, the process moves to the API testing phase, where testing is conducted using Postman to verify that the API functions as intended. Furthermore, multiple scenarios are executed during this phase to evaluate the system's security. Finally, the test outcomes are assessed and analyzed.

3.1API requirement identification

Before developing a REST API for a multi-sensor IoT monitoring application, it is crucial to clearly define the system requirements. This process ensures that the API supports seamless communication between sensors, servers, and clients. The key components interacting with the API include IoT devices such as sensors and gateways, as well as backend servers, web dashboards, and users. The REST API is specifically designed to enable efficient communication between multiple IoT sensors and the server.

This system is deployed in a moving vehicle or instrument by integrating multiple sensors. These include a load cell for weight measurement, an ADXL-345 for assessing system orientation during motion, a KY-037 for noise detection, an MPU6050 for six degrees of freedom orientation detection, and a BME280 for measuring temperature and humidity, among other parameters. The TF-Luna LiDAR sensor measures distance, the MQ-12 detects gas, the force sensing resistor (FSR-7548) gauges tension or pressure during mobility, and thermal camera sensors are employed.

The ten sensors are grouped into a multisensory setup at two distinct points, serving as data input for the REST API application. Since the primary focus of this research is on developing REST API-based applications, the sensors themselves are not discussed in detail. The system utilizes ESP-32 and long range (LoRa) communication technologies, enabling operation even in areas without Wi-Fi connectivity.

In accordance with the explanation provided in the preceding literature review, the sensors are deployed to monitor the surrounding environment, utilizing a Raspberry Pi as an edge data processor, subsequently transmitting the data through ESP and LoRa connections. The user concurrently observes sensor readings remotely via a web-based application. Therefore, it is essential to establish the comprehensive block diagram of the system.

The block diagram of the suggested system is illustrated in *Figure 4*. The diagram demonstrates that each sensor or cluster of sensors is linked to an edge device or gateway, which functions as a client within this framework as part of requirement identification. This client gathers data from the sensors and transmits it to the server for further processing. The illustration depicts a sensor pool, including numerous sensors that integrate to form an observation sensor fusion, using various microcontrollers, including ESP32 and Raspberry Pi, linked to the sensors.

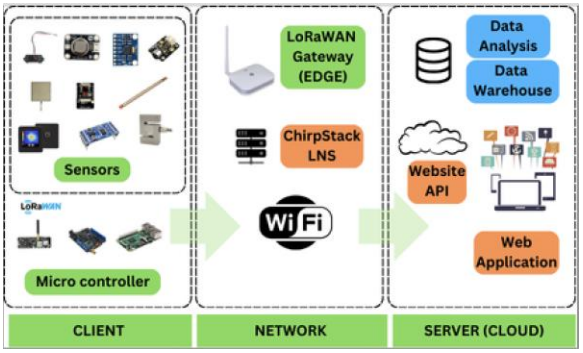


Figure 4 Primary component mapping

The MQTT protocol is frequently employed for network communication between clients and servers because of its lightweight characteristics and interoperability with IoT devices. However, in this research, the ESP32 employs the LoRaWAN protocol for remote transmission to the LoRaWAN Gateway. LoRaWAN transmits sensor data to the chirpstack LoRaWAN network server (LNS), which functions as the communication intermediary between the LoRa gateway and the backend application. Additionally, the Raspberry Pi employs Wi-Fi to communicate data straight to the API website. Consequently, the HTTP protocol is employed in this research. Concurrently, the backend system receives data from the client and executes processing tasks. Database models are constructed to delineate the database, tables, and columns. The API integrates the received data with the database. The data is subsequently stored in the data repository, further processed, and a monitoring report is generated for each monitoring sensor. The subsequent phase for the client-side interface involves the web application or mobile application functioning as a client, enabling real-time access to and retrieval of data from the API website for display purposes.

This client-server approach in IoT has the advantage that the server acts as a control center for all IoT devices, which facilitates the management and supervision of the system or known as centralized control. This approach also supports data integrity where all data is stored and managed in one central location (server), ensuring data consistency and integrity. In addition, on the security side, this approach is easier to manage, because data is collected and processed on a central server, for example through data encryption and user authentication.

3.2Design and the proposed system

The system depicted in *Figure 4* is further developed using an n-tier design illustrated in *Figure 5*. The system shown in *Figure 4* was enhanced utilizing the n-tier architecture presented in *Figure 5* to facilitate the design and organization of systems by layer. The architecture delineates the system into six tiers. The perception layer is the base layer consisting of sensors for data collection and devices that facilitate connectivity from the sensors to the next layer. The network layer comprises long range protocols. Utilizes LoRaWAN to communicate data from the board device to the Gateway through radio waves.

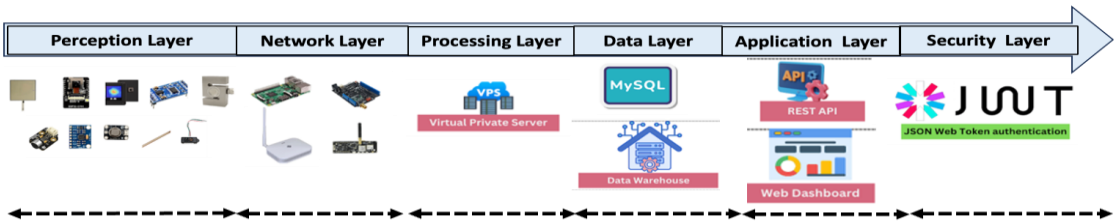


Figure 5 n-tier architecture of the proposed system

The processing layer involves the reception of data from the LoRaWAN network, which is subsequently collected and processed on the virtual private server (VPS), serving as the primary server. The data layer comprises MySQL as the database and a data warehouse for extensive data storage and subsequent data processing. The application layer employs a REST API as a third-party interface, enabling other applications to connect with the database using the REST protocol. The website exemplifies the implementation of the REST API in linking database data to the client for enhanced display and comprehension.

The upper layer is the security layer, which incorporates a JWT authentication token utilized to protect the system's security by acquiring a JSON token throughout the authentication procedure. This research involves inputting the JWT token into postman. The token is utilized to authenticate the logged-in account on the website, enabling modifications to the back-end system. If the user

alters data on the API without this token, the website returns a 400-status code, indicating either the absence of a code or that the entered code is invalid. However, if the code is provided as a JWT token, a GET or POST request returns the specified response.

Furthermore, this research proposes a web system that relies on the performance of APIs and web programming. An observation environment that has more than ten types of sensors installed in a place to monitor the surroundings. All of these sensors send output simultaneously. This research uses LoRa as a transmission medium, the sensor can transmit data even if it is more than two kilometers away without using Wi-Fi [57].

LoRa may transport data from a transmitter to a receiver using radio waves while consuming minimal power. Data transmission to the recipient necessitates time and a clear protocol. LoRa transmitters can typically only be processed by LoRa receivers once at a time. In this system, the data transmission framework is designed to incorporate more than three transmission devices, necessitating configuration at the LoRa receiver by segmenting the queue according to time. This method enables the LoRa receiver to receive signals from a single transmitter without interference. Upon receiving the data, the LoRa receiver transmits it to the API via Wi-Fi using a POST request.

The utilization of over ten sensor types in this system necessitates its capacity to process hundreds of thousands of data points simultaneously. The system may experience a decrease in performance due to the excessive volume of execution requests. The database has the greatest influence on this system; hence it needs support to optimize properly. A viable solution to this issue is the implementation of APIs as intermediaries to substitute the database in system operations, hence preventing a substantial decline in database performance.

The user-side client-based process is illustrated in *Figure 6*. The framework demonstrates that the system is partitioned into two services: data service and user service for database access. Consequently, the efficacy of data transmission is consistently upheld and does not undergo a substantial decline in performance. The illustration demonstrates the data flow inside the system, highlighting the API's role in the data storage process and manual input. Sensors transmit data to the database via the data storage mechanism. The data is recorded in the database named "data sensor" for each sensor identity using the POST method. The data is then transmitted to the website using the GET method, which receives the data in JSON format.

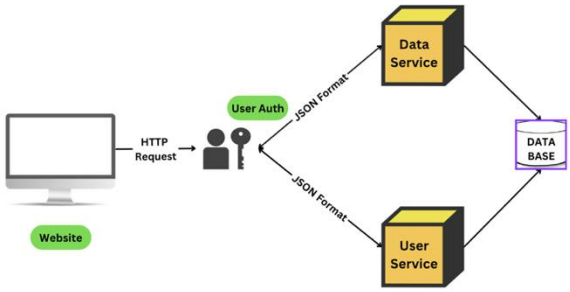


Figure 6 User-side client-based diagram

In contrast to the functionality of the admin, the admin can manually input data that alters the contents of the database via the API and subsequently returns the data modifications to the admin. The administrator can also access the website's content, which comprises data that has been processed and utilized as a sensor graph for monitoring purposes. The POST and GET methods utilize an endpoint for storing sensor data information.

Each sensor possesses a distinct endpoint to maintain an organized delivery address. Each sensor database contains a designated column, necessitating that each sensor declares an endpoint corresponding to its sensor address and thereafter save the destination value in the specified column. Consequently, the data POST system can effectively transmit data.

The sensor data is stored in a MySQL database. MySQL is an open-source and efficient database management system (DBMS) [58]. Tables and columns can be established manually or by migration with NodeJS. The API retrieves data from MySQL and subsequently updates the database. All modifications in the API data are automatically reflected within the API. APIs that interface with the database significantly reduce effort. The communication between the server and the client remains uninterrupted as the API continues to fulfill its primary role as a client-server intermediary.

The sensor_types table in *Figure 7* contains the sensor type of each sensor. This table is connected to the sensors table through the sensor_type_id attribute which indicates that each sensor has a different type. The sensors table contains the identity of each sensor used. Each sensor has one sensor type

(sensor_type_id) which is mentioned in the sensor_types table. The sensor_value table is a representative table for storing data values from each sensor. Each sensor_value table represents one sensor_id connected to the sensors table. The all_sensor table is connected to the sensor_value table to find out which set of sensors at which point sent sensor data. Finally, the users table stores information related to users to be able to access the program.

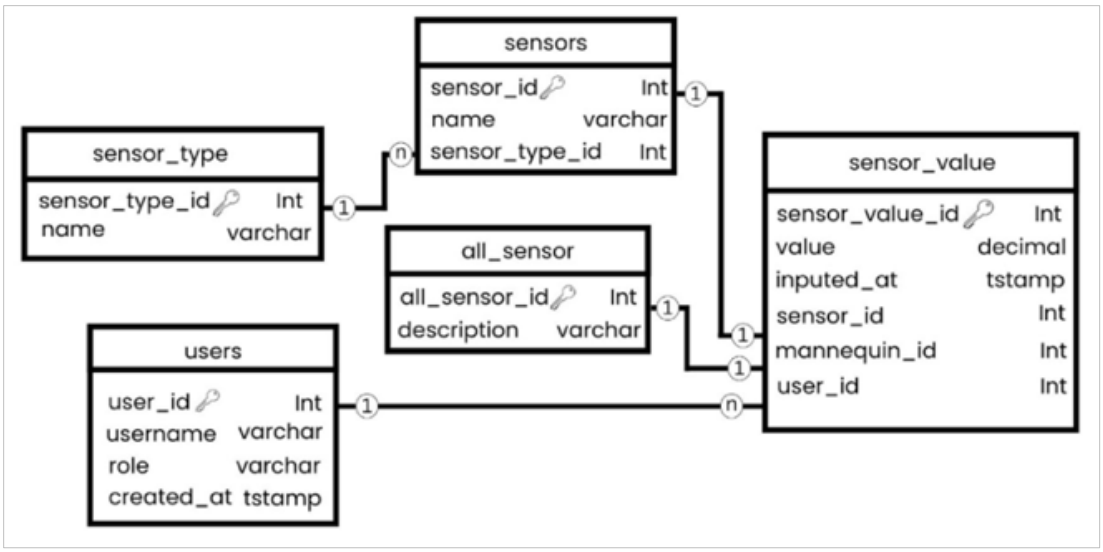


Figure 7 Database system

The next step is to retrieve website data from the API using the GET method for each type of sensor, which is then combined and monitored on the website. *Figure 8* shows the GET method to display one of the sensors, ADXL 345.

Figure 8 illustrates the script for retrieving data from the API and adapting it to the website display. It starts with a function fetchData, which is designed to retrieve data from several resources. The axios library is used to be able to make GET requests asynchronously to a list of URLs that have been provided. This function uses Promise.all which manages the asynchronous nature and ensures all requests are completed before continuing.

Furthermore, define a function that takes prevData to return updated data and then declare newData that contains the returned prevData. The looping sequence starts at the iteration of each item at the index of each newData.data. Then take fetchedData from the response index data. Extract the event_id from fetchedData as newEventId. If the newEventId value differs from the existing item.event_id, the data update is performed using the data from fetchedData. The JSON data is formatted according to the graph template to ensure proper organization and display of the existing data.

```
FUNCTION fetchData
    DECLARE promises AS LIST OF axios.get(url) FOR EACH url IN urls

    Promise.all(promises)
        .then(responses)
            SET adxlData TO FUNCTION(prevData)
                DECLARE newData AS COPY OF prevData

                FOR EACH item, index IN newData.data
                    DECLARE fetchedData AS responses[index].data.data.data

                    IF fetchedData IS NULL
                        PRINT ERROR "No data in response for URL" + urls[index]
                        CONTINUE

                    DECLARE newEventId AS fetchedData[0].event_id

                    IF item.event_id NOT EQUALS newEventId
                        SET item.event_id TO newEventId
                        SET item.series[0].data TO REVERSE(MAP(x_axis, fetchedData))
                        SET item.series[1].data TO REVERSE(MAP(y_axis, fetchedData))
                        SET item.series[2].data TO REVERSE(MAP(z_axis, fetchedData))
                        SET item.label TO REVERSE(MAP(inputed_at, fetchedData))
                    ENDIF

                    newData.data[index] = item
                NEXT item

                RETURN newData
            END FUNCTION
        END THEN
    END Promise.all
END FUNCTION
```

Figure 8 Get ADXL-345 data to website

This project develops a system including of more than ten interconnected sensors for monitoring purposes. The integrated sensors track motion, temperature,

humidity, streaming cameras, friction, lidar, gas, and noise levels, among others. The sensors are subsequently positioned within a compartment, as illustrated in *Figure 9*. A collection of sensors is housed at two distinct observation locations, with each location including 9 sensors for data collection.

(a) (b)

Figure 9 Prototype (a) node-1, (b) node-2

These sensors have undergone pre-calibration using various methods tailored to the specific type of sensor. certain individuals employ Kalman filtering techniques to refine the raw data obtained. Every sensor has been calibrated to the appropriate threshold. All sensors have successfully undergone laboratory testing, confirming their proper functionality.

Simultaneously, the application development is illustrated through a sequence diagram depicted in *Figure 10*. The sequence diagram workflow is categorized into four states: web system, API gateway, data service, and database.

The workflow begins when the user initiates an action to input or retrieve data. The application then sends a POST request to the API gateway at the /auth/signin endpoint. The API gateway forwards this request to the data service, which verifies the username and password against the database.

- If the account does not exist, the data service generates an error response stating **"Account not found."**
- If the account is found, the data service validates the user information:
  - If validation is successful, the data service returns a success response.
  - If validation fails, the data service returns an error response.
  - If a database error occurs, the data service returns a **500 Internal Server Error**.

If all steps are completed successfully, the application displays the sensor data in a graphical format.

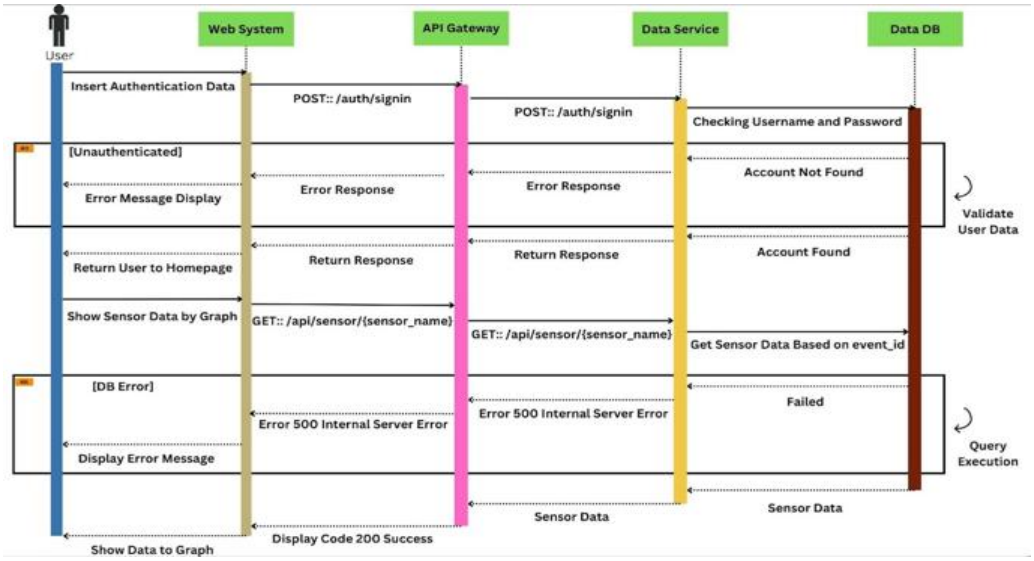


Figure 10 Sequence diagram of application development

3.3Experiment scenario

The subsequent step involves choosing a testing scenario for evaluating the REST API-based application. This aims to ensure that the API works accurately, securely, and in alignment with the expected functionality. Because the client-server architecture design exhibits vulnerabilities, including reliance on the server, potential overload, and network latency issues [59]. API testing may encompass various testing categories, including functionality testing, performance testing, and security testing, among others. The initial test involves conducting manual testing to verify the

functionality of the API endpoints prior to executing automated testing [60]. One of the methods involves utilizing Postman and cURL. It is important to note that sensor testing was not conducted or elaborated upon in the study, as the primary emphasis focused on the design of the REST API. The sensor functions as an input device that supplies data to the monitoring dashboard, having undergone numerous lab tests with the calibration method for each unit, ensuring that it remains operational throughout the testing phase. The sensor is outfitted with multiple protective features, ensuring that its primary functionality remains operational. LoRa has been validated for transmitting information to the gateway, though there is no in-depth discussion provided.

In this situation, postman executes an HTTP request to the API to observe the response and conveniently configure the request parameters. Subsequently, execute a GET and POST data request to one of the endpoints. Upon data acquisition, an HTTP response is generated with a status code of 200, indicating a successful client request. Alongside the 200-code description, postman offers details regarding the duration of data transmission and the data amount in bytes, as well as status codes such as 404 for "Not Found" and 400 for "Bad Request.". *Figures 11* and *12* depict a POST and GET situation executed on a MQ2 type sensor or gas sensor, featuring data comprising sensor_value and sensor_id. In this situation, the operation of all sensors is evaluated.


```
{
    "status": "ok",
    "message": "success input mq data",
    "data": {
        "value": 50.132,
        "inputed_at": "2024-06-25 21:45:01",
        "sensor_id": 101
    }
}
```


Figure 11 POST data using Postman

The subsequent case, designated as the second scenario, pertains to performance testing. This assessment aims to evaluate the response time and stability of the API under specific load conditions. The tool employed in this test is JMeter. This tool is effective in performing load and stress tests on the API to evaluate its capacity to manage numerous simultaneous requests.


```
{
    "status": "ok",
    "message": "success to retrieve all mq data",
    "data": [
        {
            "event_id": 1834,
            "value": "50.132",
            "inputed_at": "2024-06-25T14:45:01.000Z",
            "sensor_id": 101
        },
        {
            "event_id": 1833,
            "value": "50.220",
            "inputed_at": "2024-06-25T10:35:10.000Z",
            "sensor_id": 101
        },
        {
            "event_id": 1832,
            "value": "50.130",
            "inputed_at": "2024-06-24T11:44:54.000Z",
            "sensor_id": 101
        },
```


Figure 12 GET data using postman

In this scenario, the system transmits hundreds or even millions of requests to the API to assess response time and system resilience. The key metrics evaluated are latency and throughput. Latency refers to the average time taken by the API to respond to a request, while throughput measures the number of requests the system can process per second. These parameters are crucial in determining the API's efficiency and ability to handle high traffic loads.

This stress test evaluates the API's resilience under a load exceeding its typical capacity to identify the failure threshold. The key performance metrics—average latency, response time, and throughput—are calculated using the following formulas (Equations 1to 3):

$$\text{Average Latency} = \frac{\text{Sum of latencies of each scenario}}{\text{Number of iterations}} \quad (1)$$

$$\text{Average Response Time} = \frac{\text{Sum of response times of each scenario}}{\text{Number of iterations}} \quad (2)$$

$$\text{Throughput} = \frac{\text{Number of requests}}{\text{Minutes}} \quad (3)$$

The subsequent examination, designated as the third scenario, involves load testing on the API. This test is conducted to verify that the API can manage a substantial volume of concurrent user queries without notable performance decline. It evaluates the API's capacity to handle simultaneous queries from a large user base. Response time is assessed by observing its variation as the volume of requests increases. Additionally, bottlenecks are identified by examining the API's capacity limits and determining the specific points where performance degradation occurs.

The fourth scenario focuses on security testing, which aims to identify vulnerabilities in the API that could be exploited by unauthorized entities. In this test, authentication mechanisms are verified using tokens or API keys to ensure that API endpoints require valid credentials for access. Security testing is conducted using automated tools such as SQL injection testing and security assessment platforms like Burp Suite or structured query language map (SQLMap) to detect potential weaknesses. The final assessment involves API automation testing and monitoring through a dedicated dashboard. The objective of this test is to replicate functional testing using Postman's Collection Runner, which automates test execution to ensure consistent API behavior and reliability.

Each of these scenarios plays a crucial role in evaluating API performance. Response speed is analyzed to measure the time required to process data from IoT sensors, while throughput measurements assess the number of concurrent requests the API can handle. Bandwidth consumption is also evaluated to determine the efficiency of data transmission through the API. Furthermore, API authentication effectiveness is examined using the JWT method to assess its resilience against various cyber threats. These tests collectively ensure the API is optimized for scalability, security, and real-world deployment.

4.Results

4.1Functional performance testing

As explained earlier, the first test involves performing functional testing using Postman. The test results are shown in *Table 1*.

In *Table 1*, it is shown that manual testing using the Postman tool on multi-sensors runs smoothly. There are a total of 9 sensors, utilizing 27 APIs for the GET method and 9 APIs for the POST method. Additionally, the API supports two HTTP methods: GET and POST. The GET method is used to retrieve sensor data from the database and display it on the dashboard, while the POST method is used to input sensor data into the database.

Based on the test table, all testing processes are interconnected and related to each other. The average response time for the GET method is less than 200 milliseconds, while the POST method can take more than 400 milliseconds. This difference occurs because retrieving and displaying larger amounts of data requires more time. Similarly, the larger the payload sent, the longer the server processing time. Additionally, the length of the parameters being sent also affects the API response time, with longer parameters resulting in increased processing time.

Table 1 Client-side test results for postman tools

Sensors	Type	Average GET response time (ms)	Average GET response time (ms)
ID 201 – 204	ADXL	230	445
ID 1001	BME	136	463
ID 601 - 602	KY	145	402
ID 901	Lidar	169	209
ID 801 – 806	Load Cell	127	141
ID 1002	MPU 6050	116	113
ID 101	MQ	108	347
ID 1101 – 1108	FSR	110	111
ID 702	Thermal Camera	174	110

4.2Performance testing

In the second scenario, the test results are shown in *Figure 13*. Performance testing aims to measure the API's response time and stability under a specific load using JMeter. The testing is conducted on the client side. According to *Figure 13*, the effectiveness of the proposed system is assessed on two nodes, each consisting of multiple sensors. The system is evaluated using Apache JMeter with parameters set at 5 iterations and an increasing number of threads from 1 to a maximum of 2,382 users. The test is performed on a laptop with 32 GB of RAM to accommodate 2,382 threads. The table presents the average latency, response time, and system throughput for each case when a thread calls the API to display sensor data. At node-1, the average response time for 100 users is 139.81 ms, increasing up to 13 times at 1,000 users. At 1,500 users, the response time increases significantly by almost 15 times. Meanwhile, at node-2, the average response time for 100 users is 133.81 ms, which is 6 milliseconds faster than at node-1. Similar to node-1, node-2 also experiences an increase in response time at 1,500 users.

A downward trend in response time is observed for both node-1 and node-2 at 1,000 users, followed by an increase at 1,500 users, and another decrease at 2,000 users. The highest increase in response time occurs at 2,382 users. Meanwhile, *Figure 14* presents the throughput test results based on requests per minute for each node. The request per minute values for node-1 and node-2 do not show a significant difference. For most user counts, both nodes exhibit similar request rates. However, a noticeable difference appears between 1,500 and 2,382 users.

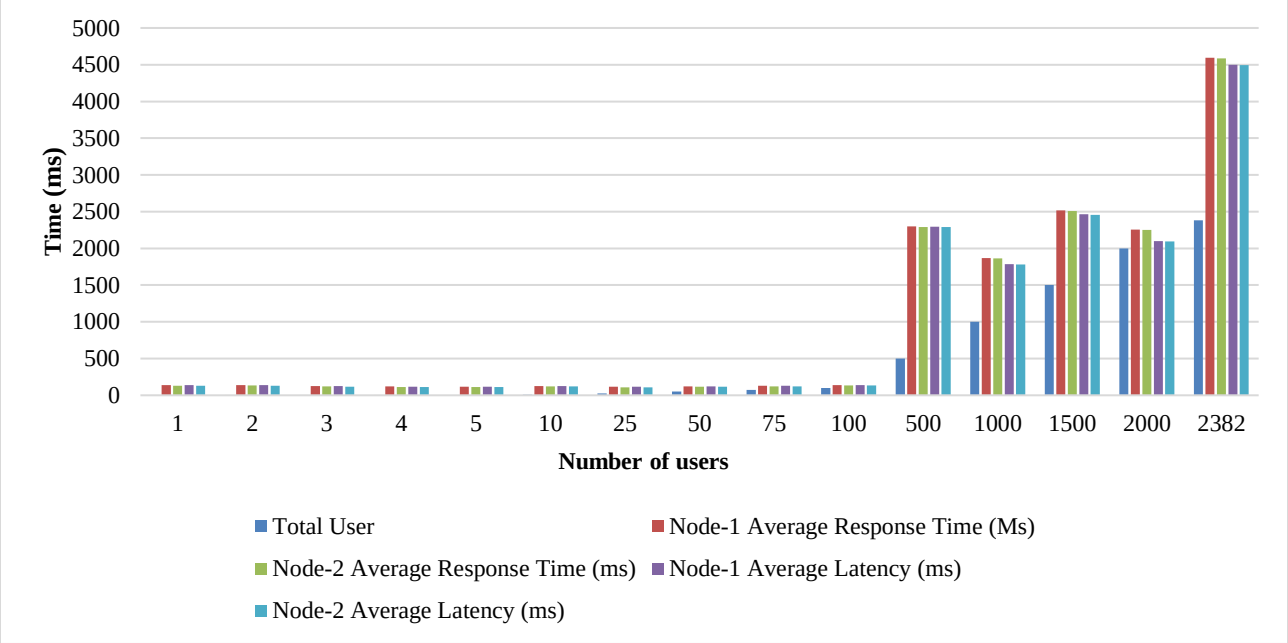


Figure 13 Client-side performance result for response time and latency

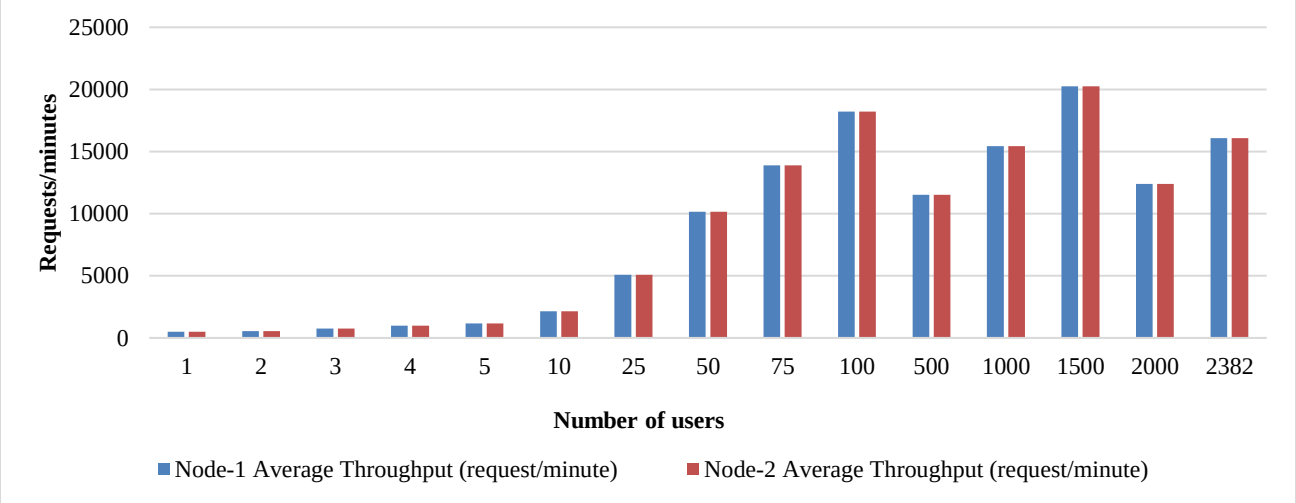


Figure 14 Client-side performance result for throughput

Figure 15 illustrates that the success rate of data transmission to the client remains at 100% when handling more than 1,400 threads. However, at node-1, a decline occurs when reaching 1,500 threads, with the success rate dropping by 1.72% and continuing to decrease until reaching 2,000 threads. Similarly, at node-2, the success rate decreases by 0.9% at 1,500 threads and further drops by 4.87%. This decline is primarily due to the system's RAM limitations, which are unable to keep up with the increasing number of threads.

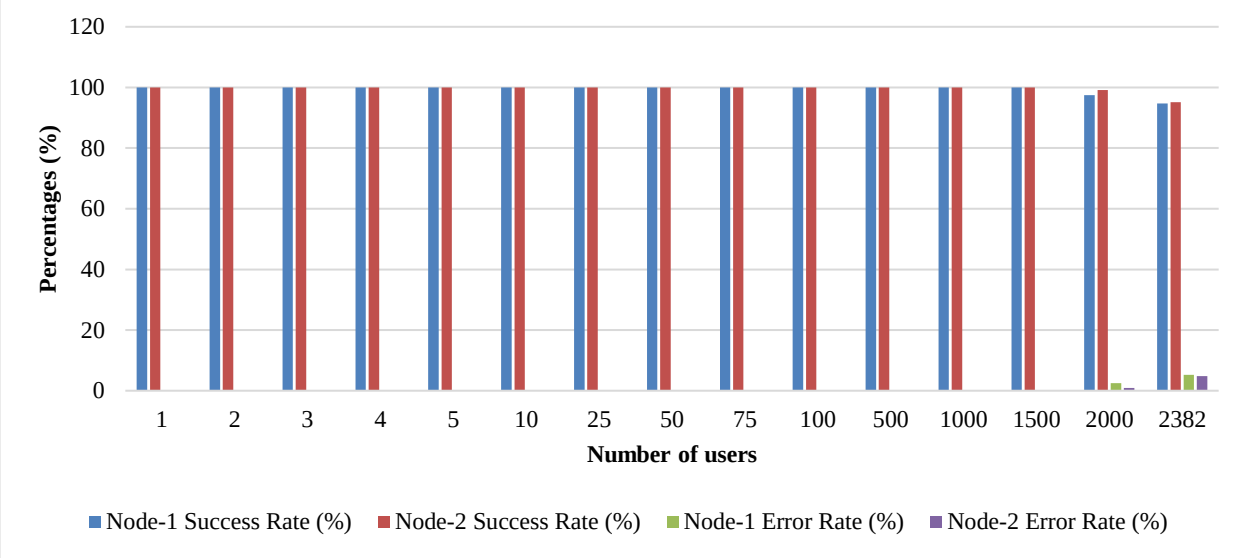


Figure 15 Client-side performance result for success rate

4.3Load testing performance

Simultaneously, in the third test, load testing was performed on the server-side API. This test aimed to verify the API's ability to handle a high volume of simultaneous user requests without significant performance degradation. The test was conducted over a duration of 5 minutes with a maximum of 100 users. The results are illustrated in *Figure 16.*

The graph below presents the outcomes of the 5-minute test, which included a total of 18,585 requests, an average response time of 172 milliseconds, and a 0% error rate. In the graph, the red line represents the error rate, the blue line indicates the response time, and the gray line depicts the number of virtual users tested.

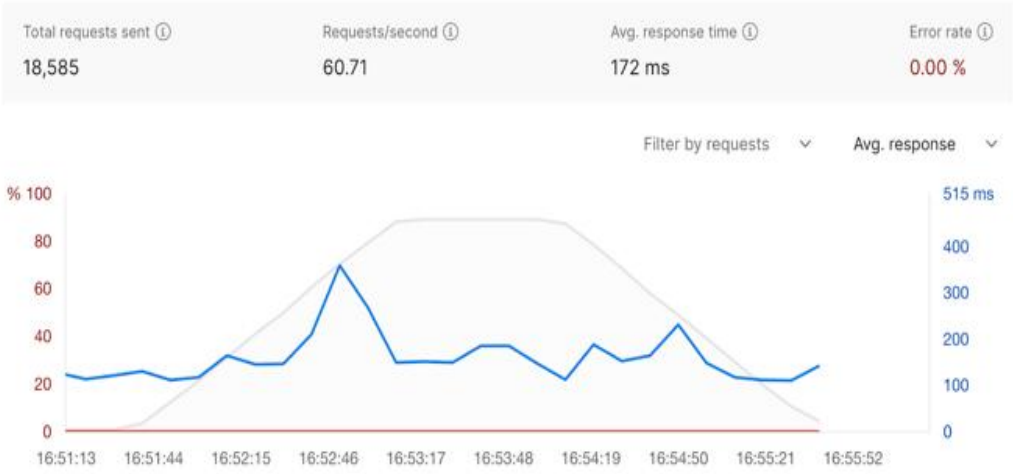


Figure 16 Load testing on the server-side API

Table 2 presents the comprehensive test results for each user concerning the GET and POST methods. *Table 2* shows that a bottleneck occurs during data retrieval using the GET method when the user count reaches 100, leading to a maximum response time approaching 2 seconds. *Figure 17* illustrates the server load under both normal and load testing conditions. *Figure 17* illustrates that under normal settings, the server load transmits 8.61 MB of data, however, during load testing, the load escalates to 26.41 MB. This occurs because, during load testing, the server experiences a higher volume of traffic compared to typical conditions.

Table 2 Server-side response time load test results

GET			POST		
Min	**Avg**	**Max**	**Min**	**Avg**	**Max**
106	128	165	106	116	135
99	114	188	95	108	152
96	114	252	94	110	220
97	265	1300	96	197	1330
94	149	632	95	157	652
97	200	969	94	178	828
92	168	1990	93	120	241

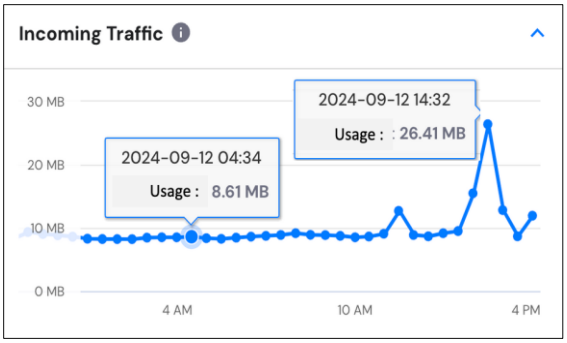


Figure 17 Server load during standard operations and load evaluation

4.4Security performance testing

The fourth assessment involves security testing, which aims to identify vulnerabilities in the API that could be exploited by unauthorized entities. The initial phase of security testing focuses on authentication and authorization processes, as illustrated in the following figure. *Figure 18* demonstrates the procedure for retrieving data from https://api.stas-rg.com/sensor, which fails due to the absence of an authentication token. The authentication process can be performed at https://api.stas-rg.com/auth/signin, where users must enter a valid email and password. Upon successful authentication, a JWT is issued, which remains valid for one hour.

Once the token is obtained, Postman can use it in the Authentication - Bearer Token section to gain authorized access. If incorrect credentials are entered, the system denies access to unauthorized users. However, when the correct email and password are provided, the user can retrieve the previously restricted data by supplying the authentication token. The next phase of testing involves penetration testing using Burp Suite Community Edition 2024.7.5.0. The testing follows the protocols established by Burp Suite, where user data is intercepted during entry into the database. This allows Burp Suite to capture API request inputs, commonly referred to as SQL injection testing.

Figure 19 presents the intercepted input data obtained through Burp Suite, showcasing how the tool analyzes potential vulnerabilities in the API by examining request parameters for possible SQL injection exploits.

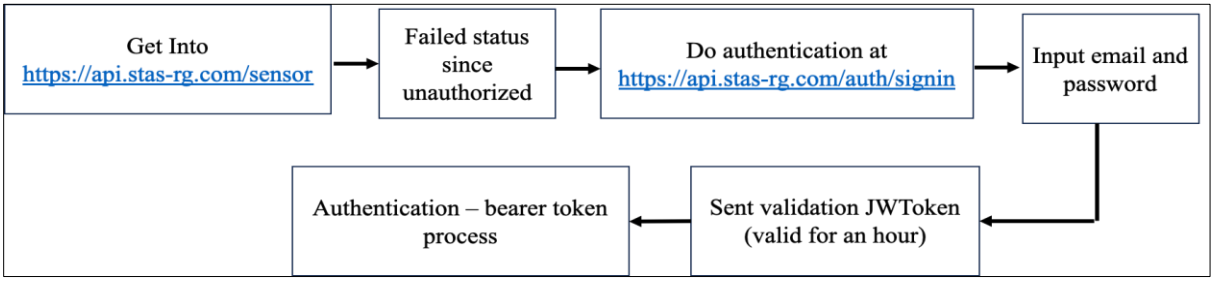


Figure 18 Authentication and authorization process

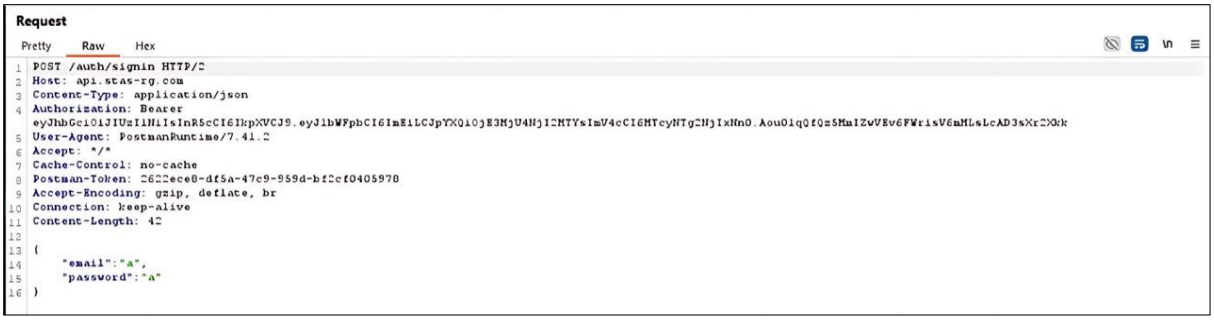


Figure 19 The procedure for submitting input in burp suite

Upon capturing, this text is stored as a request.txt file for further processing using SQLMap. The SQL injection procedure in SQLMap requires the SQLMap package and is executed on the Windows operating system. The command entered in the command prompt is "python sqlmap.py -r (file_name) --dbs", where "python" initiates the execution of all .py files. The sqlmap.py script specifies the filename required to run SQLMap. The -r flag instructs SQLMap to read HTTP requests directly from the specified file. In this case, request.txt represents the target file, while --dbs is used to enumerate all database names on the target server.

The consequences of this injection are illustrated in *Figure 20*. The graphic displays an error indicating that multiple processes were compromised, including MySQL, PostgreSQL, and Microsoft SQL Server. However, it also highlights other processes that remained secure, demonstrating that the website's infrastructure has protective measures in place against SQL injection attacks.

```
[13:47:40] [INFO] testing 'Boolean-based blind - Parameter replace (original value)'
[13:47:40] [INFO] testing 'MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
[13:47:42] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[13:47:43] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'
[13:47:45] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'
[13:47:47] [INFO] testing 'Generic inline queries'
[13:47:47] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
[13:47:49] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'
[13:47:51] [INFO] testing 'Oracle stacked queries (DBMS_PIPE.RECEIVE_MESSAGE - comment)'
[13:47:52] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)'
[13:47:54] [INFO] testing 'PostgreSQL > 8.1 AND time-based blind'
[13:47:56] [INFO] testing 'Microsoft SQL Server/Sybase time-based blind (IF)'
[13:47:57] [INFO] testing 'Oracle AND time-based blind'
it is recommended to perform only basic UNION tests if there is not at least one other (potential) technique found. Do you want to reduce the number of requests? [Y/n] n
[13:48:08] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[13:48:26] [WARNING] (custom) POST parameter 'JSON email' does not seem to be injectable
[13:48:26] [INFO] testing if (custom) POST parameter 'JSON password' is dynamic
[13:48:27] [WARNING] (custom) POST parameter 'JSON password' does not appear to be dynamic
[13:48:27] [INFO] testing for SQL injection on (custom) POST parameter 'JSON password'
[13:48:27] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[13:48:31] [INFO] testing 'Boolean-based blind - Parameter replace (original value)'
[13:48:32] [INFO] testing 'MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
[13:48:34] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[13:48:37] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'
[13:48:38] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'
[13:48:41] [INFO] testing 'Generic inline queries'
[13:48:41] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
[13:48:42] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'
[13:48:44] [INFO] testing 'Oracle stacked queries (DBMS_PIPE.RECEIVE_MESSAGE - comment)'
[13:48:45] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)'
[13:48:47] [INFO] testing 'PostgreSQL > 8.1 AND time-based blind'
[13:48:48] [INFO] testing 'Microsoft SQL Server/Sybase time-based blind (IF)'
[13:48:50] [INFO] testing 'Oracle AND time-based blind'
[13:48:52] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[13:49:12] [WARNING] (custom) POST parameter 'JSON password' does not seem to be injectable
[13:49:12] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk' options if you wish to perform more tests. If you suspect that there is some ki
nd of protection mechanism involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'
[13:49:12] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 244 times

[*] ending @ 13:49:12 /2024-10-14/
```

Figure 20 Injection error result

The cross-site request forgery (CSRF) test is conducted by inputting a form script acquired via Burp Suite, which is then transmitted without website authentication, following the steps illustrated in *Figure 21*. This CSRF test is an attack that forces authenticated users to execute actions without their consent by exploiting browser security protocols that allow automatic requests without user approval.

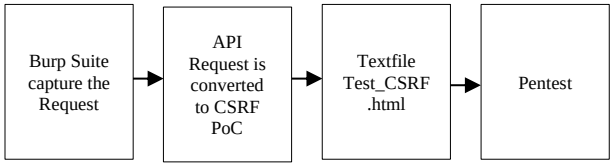


Figure 21 CSRF testing

The graphic illustrates that in the initial phase, Burp Suite intercepts the request submitted by the user, who deliberately enters a genuine username and password. The last phase involves transforming the API request into a CSRF proof of concept, which may be constructed within Burp Suite. However, due to the user's access being limited to the community version, this conversion is executed utilizing an online generator to obtain the form output. The data is subsequently stored and recorded in a text file titled test CSRF.html, after which the penetration testing method is executed.

The outcome of the CSRF test yields a timeout error, since api.stas-rg.com rejects the input due to its

classification as CSRF. Consequently, this method has been demonstrated to be ineffective in breaching the security of the API website, as illustrated in *Figure 22*. Consequently, it can be inferred that the implementation of REST APIs in IoT-based systems demonstrates commendable security. The constructed website features a singular form page, specifically the login page, to mitigate the risk of vulnerabilities such as SQL Injection or XSS in the form and search bar.

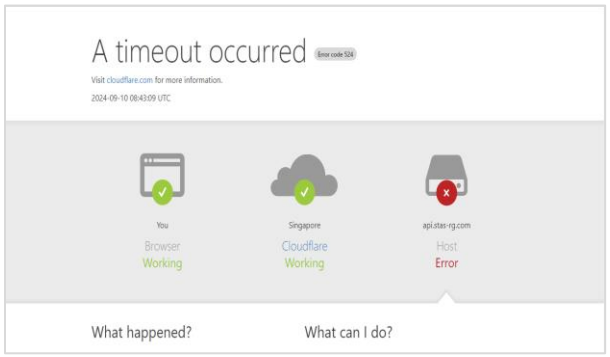


Figure 22 Pentest results of the API website

4.5API Dashboard monitoring

The final assessment focuses on API automation testing and a monitoring dashboard. The primary objective of this test is to replicate functional testing using the postman collection runner (PCR), which facilitates automated API testing.

To perform API automation via the Collection Runner in Postman, the first step is to configure the API collection (pre-request) to automate the sensor type and sensor ID. The sensor_name variable represents the sensor type, while the sensor_ID variable holds the sensor ID. These variables are dynamically incorporated into the API URL to enable flexible and automated testing. The corresponding script is illustrated in *Figure 20.*

In *Figure 23*, the executed script for automation testing systematically evaluates each sensor to ensure that the received response code is 200, confirming the successful input and retrieval of sensor data from the database. The results of the GET method test are shown in *Figure 24*. The figure indicates that the total execution time for all APIs using the GET method is 3.82 seconds, with an average response time of 121 milliseconds. Similarly, the POST method test results are presented in *Figure 25*. The total execution time for all APIs using the POST method is 4.55 seconds, with an average response time of 128 milliseconds.

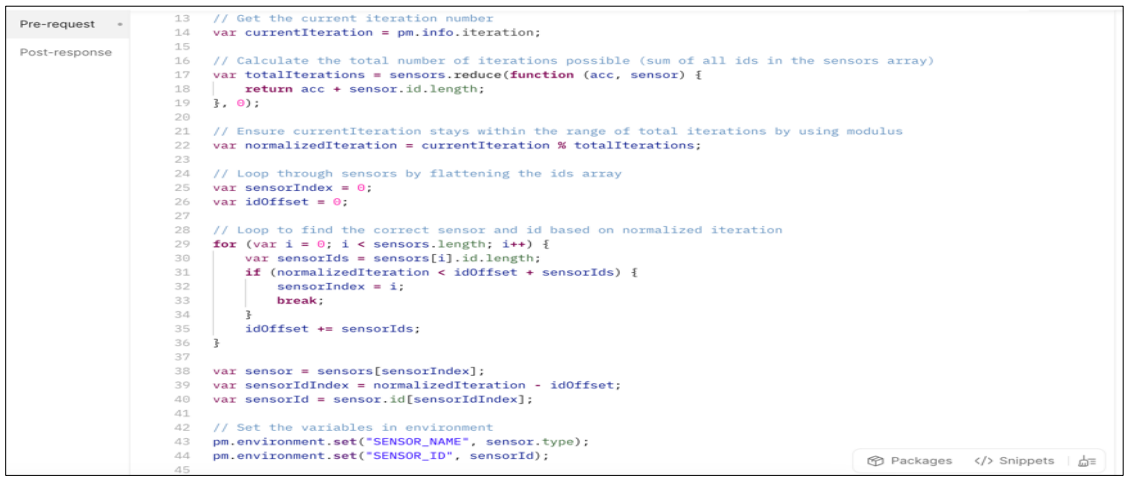


Figure 23 Testing script on PCR

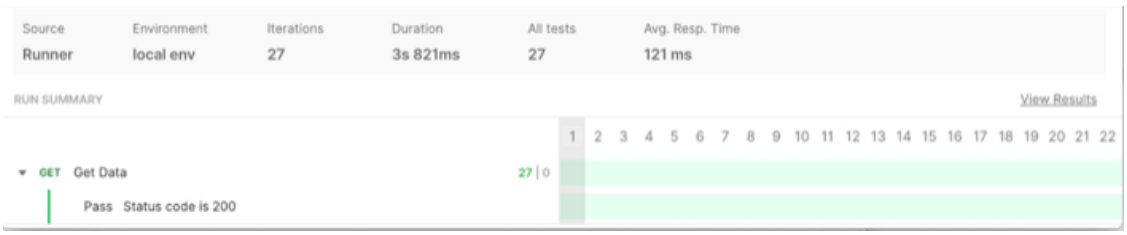


Figure 24 GET method testing

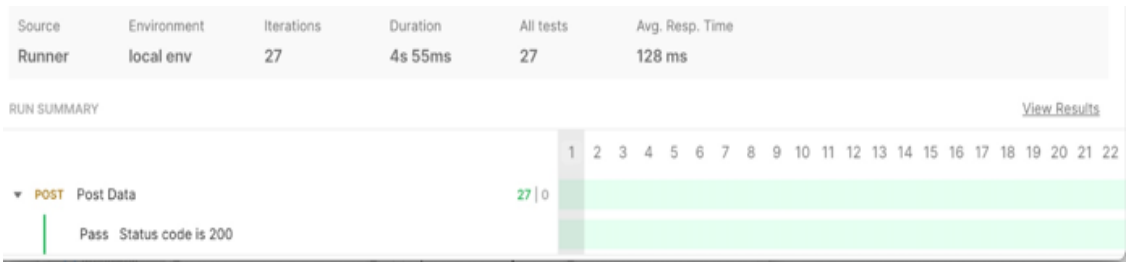
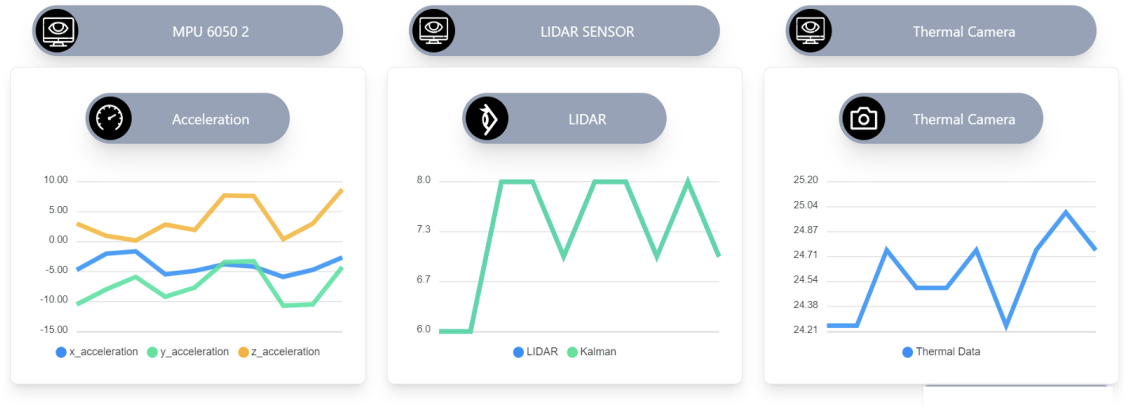
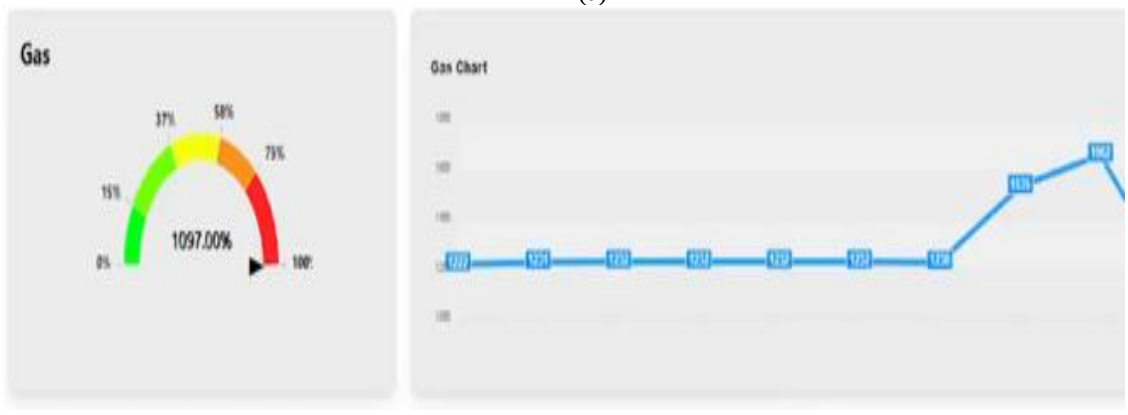


Figure 25 POST method testing

Based on the test results of the POST and GET methods, the display dashboard confirms that the sensors can be effectively observed and monitored, as shown in *Figure 26*. In *Figure 26(a)*, the node-1 display presents the signal transmission graph for the MPU6050 sensor, LiDAR sensor, and thermal camera. Meanwhile, *Figure 26(b)* shows the node-2 dashboard, which displays gas sensor observations along with the corresponding graphs.



(a)



(b)

Figure 26 Dashboard testing: (a) node-1, (b) node-2

5.Discussion

This study explores the development of multiple sensor websites and the integration of the REST API within the system. This website is expected to function as a monitoring tool capable of directly observing conditions in the vicinity of the sensor placements. Through monitoring, the system can display the condition on the dashboard if an abnormal event occurs at the location.

The design and implementation of a REST API for online monitoring is intended to manage the numerous sensors attached to the system. This aims to enhance

contemporary applications by integrating sensors within systems that facilitate smart city initiatives. The system incorporates data inputs from a total of 18 sensors, strategically placed across two distinct locations, with 9 sensors allocated to each site. A total of 27 application programming interfaces were utilized. The packet loss rate in LoRaWAN communication was overlooked, as the focus was on evaluating the API performance during data retrieval from the sensors.

In the initial test, the mean response time for contacting the API via GET and POST varied between 200 and 400 milliseconds. This facilitates access to the monitoring system, which may be observed in real-time on the website. In this testing scenario, it can be confirmed that all sensors are capable of transmitting data accurately via the API. One node was set up in a Wi-Fi zone, while another node was installed in an area utilizing LoRaWAN. Both nodes successfully transmitted their data. Devices situated in monitoring areas lacking wireless fidelity (WiFi) connectivity can relay their data via the LoRaWAN gateway, functioning as edge devices. In contrast to earlier studies, LoRA has the capability to surpass the limitations of Wi-Fi transmission, enabling the monitoring of sensors that are beyond the reach of Wi-Fi networks. In earlier studies, sensor data was saved on a storage device when Wi-Fi was unable to transmit data. In this system, LoRa effectively addresses the limitations of Wi-Fi. Consequently, the data is guaranteed to be transmitted accurately.

In this framework, LoRaWAN may be suggested as a means of communication redundancy for systems that necessitate extensive and broad coverage. By prioritizing Wi-Fi as the primary connection and only resorting to LoRaWAN when the Wi-Fi network is unavailable. Through an automatic failover mechanism, sensors are able to select the most effective communication method according to the prevailing network conditions. As this study emphasizes the performance of the REST API, subsequent investigations could explore further distinct assessments required to evaluate packet loss on LoRaWAN under different conditions (urban, rural, industrial areas). Moreover, the system has the capability to utilize edge computing, allowing for certain processing to occur directly on the sensor prior to data transmission. The system can also incorporate methods to reduce the frequency of data transmission or utilize compression techniques to help conserve LoRaWAN bandwidth.

In the realm of IoT monitoring systems, API response times for GET and POST methods between 200 and 400 ms fall within the semi-real-time category. The acceptable range of 100 to 500 ms remains suitable for IoT systems that do not demand ultra-fast responses. APIs continue to function within acceptable boundaries for IoT monitoring systems, though not entirely in real-time. A recent comparison of REST APIs and MQTT revealed that REST APIs generally exhibit higher latency when contrasted with protocols like MQTT or WebSockets. This is attributed to the request-response nature of REST, in contrast to the more efficient publish-subscribe communication model. The primary elements influencing API response time include server load, the volume of concurrent requests, database optimization, and network latency, particularly in the context of LoRaWAN. Strategies for optimization that can be implemented encompass caching, load balancing, database enhancement, the utilization of message queues, and integration with WebSocket/MQTT for real-time communication.

In the second scenario, system performance testing was executed with a maximum of 2382 users on each node. The system performance deteriorated when 1500 users accessed the website, resulting in 0.9% - 1.72% of users being unable to gain access. The situation emerged from the significant number of visitors utilizing the website, resulting in a substantial load on user authentication (as each user requires verification through a token) and the constrained memory capacity of the testing equipment, which is 32 GB. This led to a disparity between the volume of API requests and the server's ability to manage concurrent requests.

For systems that accommodate numerous users and sensors, substantial memory is crucial to ensure immediate access and facilitate efficient website usage. Consequently, applications necessitating extra observation points demand a broadened memory architecture capable of supporting enhanced services. Furthermore, implementing a load balancer to achieve a more equitable distribution of the load may serve as a viable solution for the upcoming proposal. Furthermore, enhancing the caching mechanism to alleviate the query load on the database and expanding server capacity or transitioning to cloud computing with auto-scaling could serve as viable alternatives for a large smart city application.

In load testing for the third scenario, the response time fluctuates between 90 milliseconds and 1.9 seconds,

with an average duration ranging from 108 to 265 milliseconds. The maximum response time of 1.9 seconds occurs when the system reaches its full capacity with 100 users.

The sorting limitations arise from the requirement for users to retrieve data from the server, which must simultaneously verify the access token assigned to each user. This authentication process places additional strain on the server, as it must validate tokens for every user request. As the number of users increases, the server load rises, leading to longer response times.

Moreover, at the fourth scenario, the security testing incorporated into the application utilizing JWT indicates that the system's security is commendable based on the test findings. This is evidenced by three criteria: authentication and authorization, SQL injection, and CSRF testing, which collectively ensure robust application protection. No attack has successfully breached the security implemented on JWT. Nevertheless, despite JWT offering a commendable degree of security, several vulnerabilities warrant consideration. If the token is inadequately encrypted, there exists a danger of sensitive information exposure. Moreover, prolonged use of JWT without a token refresh mechanism poses problems in the event of token theft. To enhance security and mitigate the danger of misuse, it is advisable to implement supplementary encryption for the token payload and adopt a more robust OAuth 2.0 framework, which encompasses establishing token expiration and utilizing refresh tokens.

The last assessment is the automation system execution test and monitoring dashboard utilizing the PCR, which enables users to execute a series of API requests consecutively within a single collection, with functionalities for repetition, variable incorporation, and automatic response monitoring. This PCR facilitates efficient and repetitive validation of API performance for developers and testers. This method enables the organization of each API test conducted via the PCR into an organized collection, facilitating developers to execute a series of tests with a single click. This feature enables the team to efficiently monitor and evaluate test outcomes to confirm the API's proper functionality.

The findings from the discussion indicate that the developed REST API system can effectively manage up to 2382 users while maintaining satisfactory performance. Nonetheless, for extensive applications, like smart cities equipped with tens of thousands of sensors, additional advancements are required in several areas, including system scalability, AI integration, and database optimization. To be effective on a smart city scale, the system needs to manage tens to hundreds of thousands of sensors distributed across multiple locations, ensuring significantly enhanced user access. This system serves as a foundational element for small-scale applications in smart cities, which can subsequently be aggregated into a node for large-scale utilization.

Limitations

This study encounters scalability challenges, particularly due to a surge in API requests as thousands of sensors simultaneously transmit data, significantly increasing the system’s load. As the number of nodes increases, the system must manage complex connectivity, integrating technologies such as Wi-Fi, LoRaWAN, and 5G.

Additionally, as the volume of incoming data grows, enhancing server capacity becomes essential to accommodate higher processing demands. Storage management also needs optimization, as the integration of additional sensors requires extensive historical data storage.

The system currently lacks load balancing and auto-scaling mechanisms, making it difficult to distribute API traffic efficiently. Furthermore, an in-depth analysis of LoRaWAN packet loss has not yet been conducted, which is crucial for ensuring data transmission reliability. The system's performance declines when user numbers exceed 1,500, primarily due to authentication processing overhead and server memory limitations (32 GB).

A complete list of abbreviations is listed in *Appendix I.*

6.Conclusion and future work

The implemented REST API system has demonstrated the ability to support up to 2,382 users while maintaining commendable performance in the testing scenario. This system enables real-time monitoring of sensor conditions by integrating LoRaWAN and Wi-Fi, making it an effective solution for environments with limited Wi-Fi access. LoRaWAN has proven to be advantageous in overcoming Wi-Fi limitations by transmitting sensor data from remote locations, ensuring broader and more reliable communication.

Security testing confirms that the system has successfully defended against SQL Injection, CSRF, and authentication attacks. However, potential vulnerabilities exist in JWT implementation. To enhance security, it is strongly recommended to:

- Encrypt the JWT payload to prevent unauthorized data exposure.
- Implement OAuth 2.0 for secure authentication and authorization.
- Incorporate refresh tokens to mitigate the risk of token theft and session hijacking.

To improve system scalability and efficiency, the following enhancements are proposed:

- Separating API services into distinct microservices allows independent scaling of each service.
- Distributing API requests across multiple servers ensures efficient resource utilization and maintains performance as user demand grows.
- Upgrading to a more scalable database can improve sensor data management and retrieval efficiency.
- Implementing machine learning to detect anomalies in sensor data allows the system to automatically recognize irregular patterns and trigger timely alerts.
- The system should incorporate a combination of Wi-Fi, LoRaWAN, and 5G to efficiently handle diverse sensor connectivity needs in large-scale deployments.

Acknowledgment

We extend our sincere gratitude to the Directorate General of Vocational Education, Ministry of Education, Research, and Technology of the Republic of Indonesia. We would also like to express our appreciation to the Directorate of Research and Community Service (PPM) at Telkom University and the Applied Science Laboratory of Sustainable Technology (STAS) at Telkom University, Bandung, Indonesia, for their invaluable support in this research endeavor. This research was funded by the Directorate of Research and Community Service (PPM) under the Superior Applied Research Scheme for Higher Education.

Conflicts of interest

The authors have no conflicts of interest to declare.

Data availability

The data utilized in this study were gathered from smart mannequin research test data, which includes military data curated by the Center of Excellence Smart Technology and Applied Science RG at Telkom University. The information is not accessible to the public. Nonetheless, these can be made available by the corresponding author upon a reasonable request.

Author's contribution statement

Giva Andriana Mutiara: Conceptualization, investigation, supervision, writing – original draft, writing – review and editing. **Periyadi:** Data collection, data curation. **Muhammad Rizqy Alfarisi**: Design, investigation on challenges. **Muhammad Ghifar Rijali:** Analysis, data collection. **Muhammad Aulia Rifqi Zain**: Draft manuscript preparation, investigation. **Fathurrohman Nur Rochim**: Coding and testing.

References

[1] Nugraha AK, Mutiara GA, Gunawan T, Hapsari GI. Android-based system monitoring of supporting variables for nursery-plant growth in plantation areas. JOIV: International Journal on Informatics Visualization. 2023; 7(1):51-7.

[2] Pratama FD, Mutiara GA, Meisaroh L. A virtual cage for monitoring system semi-intensive livestock's using wireless sensor network and haversine method. Jurnal Infotel. 2023; 15(2):201-8.

[3] Javaid M, Haleem A, Singh RP. Health informatics to enhance the healthcare industry's culture: an extensive analysis of its features, contributions, applications and limitations. Informatics and Health. 2024; 1(2):123-48.

[4] Nižetić S, Šolić P, Gonzalez-de DL, Patrono L. Internet of things (IoT): opportunities, issues and challenges towards a smart and sustainable future. Journal of Cleaner Production. 2020; 274:1-32.

[5] Pathmudi VR, Khatri N, Kumar S, Abdul-qawy AS, Vyas AK. A systematic review of IoT technologies and their constituents for smart and sustainable agriculture applications. Scientific African. 2023; 19:1-14.

[6] Sasi T, Lashkari AH, Lu R, Xiong P, Iqbal S. A comprehensive survey on IoT attacks: taxonomy, detection mechanisms and challenges. Journal of Information and Intelligence. 2024; 2(6):455-513.

[7] Martens CD, Da SLF, Silva DF, Martens ML. Challenges in the implementation of internet of things projects and actions to overcome them. Technovation. 2022; 118:102427.

[8] Tawalbeh LA, Muheidat F, Tawalbeh M, Quwaider M. IoT privacy and security: challenges and solutions. Applied Sciences. 2020; 10(12):1-17.

[9] Kumar S, Tiwari P, Zymbler M. Internet of things is a revolutionary approach for future technology enhancement: a review. Journal of Big Data. 2019; 6(1):1-21.

[10] Chyrun L, Gozhyj A, Yevseyeva I, Dosyn D, Tyhonov V, Zakharchuk M. Web content monitoring system development. In COLINS 2019 (pp. 126-42).

[11] Rakhmawati NA, Ferlyando V, Samopa F, Astuti HM. A performance evaluation for assessing registered websites. Procedia Computer Science. 2017; 124:714-20.

[12] Tapia F, Mora MÁ, Fuertes W, Aules H, Flores E, Toulkeridis T. From monolithic systems to microservices:a comparative study of performance. Applied Sciences. 2020; 10(17):1-35.

[13] Awais M, Iqbal J. Layered architecture of internet of things-a review. Proceeding book of 2nd international conference on scientific and academic research 2023 (pp.124-31).
[14] Mena M, Criado J, Iribarne L, Corral A, Chbeir R, Manolopoulos Y. Towards high-availability cyber-physical systems using a microservice architecture. Computing. 2023; 105(8):1745-68.
[15] Rescati M, De MM, Paganoni M, Pau D, Schettini R, Baschirotto A. Event-driven cooperative-based internet-of-things (IoT) system. In international conference on IC design & technology 2018 (pp. 193-6). IEEE.
[16] Hamdan S, Ayyash M, Almajali S. Edge-computing architectures for internet of things applications: a survey. Sensors. 2020; 20(22):1-52.
[17] Hossain MD, Sultana T, Akhter S, Hossain MI, Thu NT, Huynh LN, et al. The role of microservice approach in edge computing: opportunities, challenges, and research directions. ICT Express. 2023; 9(6):1162-82.
[18] Rathore N, Rajavat A, Patel M. Investigations of microservices architecture in edge computing environment. In social networking and computational intelligence: proceedings of SCI-2018 2020 (pp. 77-84). Springer Singapore.
[19] Li L, Chou W, Zhou W, Luo M. Design patterns and extensibility of REST API for networking applications. IEEE Transactions on Network and Service Management. 2016; 13(1):154-67.
[20] Thalor M, Allur SR, Bhende VS, Chavan A. Analysis of monolithic and microservices system architectures for an E-commerce web application. International Journal of Intelligent Systems and Applications in Engineering (IJISAE). 2024; 12(4):2400–6.
[21] Zulkarnaini Z, Wahyuningrum I, Octarina A. Monolithic architecture integrated web application school educational management information system. In proceedings of the 7th first international conference on global innovations (FIRST-ESCSI 2023) 2024 (pp. 398-407). Springer Nature.
[22] Saha T. Application development using microservice architecture. Culminating Projects in Computer Science and Information Technology. 2022.
[23] Elhoseny H, Hazem EB. Utilizing service oriented architecture (SOA) in IoT smart applications. Journal of Cybersecurity and Information Management. 2019; 0(1):15-31.
[24] Rashid T, Mustafa S. A review on IoT: layered architecture, security issues and protocols. International Journal of Computer Science and Network Security. 2023; 23(9):100-10.
[25] Abba AAA, Djedouboum AC, Gueroui AM, Thiare O, Mohamadou A, Aliouat Z. A three-tier architecture of large-scale wireless sensor networks for big data collection. Applied Sciences. 2020; 10(15):1-21.
[26] Gupta P, Mokal TP, Shah DD, Satyanarayana KV. Event-driven SOA-based IoT architecture. In international conference on intelligent computing and applications: ICICA 2016 (pp. 247-58). Springer Singapore.
[27] Belhe S, Barshikar S, Kadu S. Serverless computing and its impact on application development in cloud environments. International Journal of Technology Engineering Arts Mathematics Science. 2023; 3(2):22-8.
[28] Ouyang R, Wang J, Xu H, Chen S, Xiong X, Tolba A, et al. A microservice and serverless architecture for secure IoT system. Sensors. 2023; 23(10):1-24.
[29] Karaduman B, Oakes BJ, Eslampanah R, Denil J, Vangheluwe H, Challenger M. An architecture and reference implementation for WSN-based IoT systems. In emerging trends in IoT and integration with data science, cloud computing, and big data analytics 2022 (pp. 80-103). IGI Global Scientific Publishing.
[30] Dixit A, Trivedi A, Godfrey WW. IoT and machine learning based peer to peer framework for employee attendance system using blockchain. In international conference on augmented intelligence and sustainable systems 2022 (pp. 1088-93). IEEE.
[31] O‘g‘li TQ, Anatolevna AD, Ilgizarovna ZR. Exploring client-side and server-side architectures in web development: a comprehensive analysis. Science and Innovation. 2024; 3(17):654-8.
[32] Chen H, Xu L. Software architecture and framework for programmable automation controller: a systematic literature review and a case study. Machines.2016; 4(2):1-14.
[33] Khan SMA. Popular software architecture used in software development. Kindle Publisher; 2023:1-101.
[34] Vila M, Sancho MR, Teniente E. XYZ monitor: IoT monitoring of infrastructures using microservices. In international conference on service-oriented computing 2020 (pp. 472-84). Cham: Springer International Publishing.
[35] Alhajri K, Alghamdi M, Alrashidi M, Balharith T, Tabeidi R. Smart office model based on internet of things. In the international conference on artificial intelligence and computer vision 2021 (pp. 174-83). Cham: Springer International Publishing.
[36] Vanteru MK, Jayabalaji KA, Ilango P, Nautiyal B, Begum AY. Multi-sensor based healthcare monitoring system by LoWPAN-based architecture. Measurement: Sensors. 2023; 28:1-7.
[37] Ramu VB. Edge computing performance amplification. International Journal of Recent Advances in Science and Technology. 2023; 10(3):69-76.
[38] Medina S, Montezanti D, Gómez DL, Garay F, De GA, Naiouf M. Distributed architectures based on edge computing, fog computing and end devices: a conceptual review incorporating resilience aspects. In conference on cloud computing, big data & emerging topics 2023 (pp. 31-44). Cham: Springer Nature Switzerland.
[39] Katal A, Dahiya S, Choudhury T. Energy efficiency in cloud computing data center: a survey on hardware

technologies. Cluster Computing. 2022; 25(1):675-705.
[40] Nguyen T, Nguyen H, Gia TN. Exploring the integration of edge computing and blockchain IoT: Principles, architectures, security, and applications. Journal of Network and Computer Applications. 2024:103884.
[41] Nyabuto MG, Mony V, Mbugua S. Architectural review of client-server models. International Journal of Scientific Research and Engineering Trends. 2024; 10(1):139-43.
[42] Hernández LM, Cadena AH, Vázquez JN, Magaña JÁ, Zea JM. REST (Representational State Transfer) architecture for enterprise web application development. Innovación Y Desarrollo Tecnológico Revista Digital. 2020; 12(3):219-27.
[43] Zaniewski P, Law RR. Comparative review of selected internet communication protocols. Foundations of Computing and Decision Sciences. 2023; 48(1):39-56.
[44] Koteswaramma R. Client-side load balancing and resource monitoring in cloud. International Journal of Engineering Research and Applications. 2012; 2(6):167-71.
[45] Chawngsangpuii R, Das P. Communication protocol in internet of things. International Journal of Innovative Technology and Exploring Engineering. 2020: 9(6): 1737-40.
[46] Ahmad I, Suwarni E, Borman RI, Rossi F, Jusman Y. Implementation of restful API web services architecture in takeaway application development. In 1st international conference on electronic and electrical engineering and intelligent system 2021 (pp. 132-7). IEEE.
[47] Tariq U, Ahmed I, Bashir AK, Shaukat K. A critical cybersecurity analysis and future research directions for the internet of things: a comprehensive review. Sensors. 2023; 23(8):1-46.
[48] Abbasi M, Plaza-hernandez M, Prieto J, Corchado JM. Security in the internet of things application layer: requirements, threats, and solutions. IEEE Access. 2022; 10:97197-216.
[49] William G, Anthony R, Purnama J. Development of NodeJS based backend system with multiple storefronts for batik online store. In proceedings of the 2020 international conference on engineering and information technology for sustainable industry 2020 (pp. 1-6). ACM.
[50] Rawat P, Mahajan AN. ReactJS: a modern web development framework. International Journal of Innovative Science and Research Technology. 2020; 5(11):698-702.
[51] Kaushik V, Gupta K, Gupta D. React native application development. International Journal of Advanced Studies of Scientific Research. 2019; 4(1):461-67.
[52] Quan Y. Design and implementation of e-commerce platform based on Vue.JS and MySQL. In 3rd international conference on computer engineering, information science & application technology 2019 (pp. 449-54). Atlantis Press.
[53] Dalimunthe S, Putra EH, Ridha MA. Restful API security using json web token (JWT) with hmac-sha512 algorithm in session management. IT Journal Research and Development. 2023; 8(1):81-94.
[54] Bucko A, Vishi K, Krasniqi B, Rexha B. Enhancing JWT authentication and authorization in web applications based on user behavior history. Computers. 2023; 12(4):1-18.
[55] Al-ali AR, Gupta R, Zualkernan I, Das SK. Role of IoT technologies in big data management systems: a review and smart grid case study. Pervasive and Mobile Computing. 2024: 101905.
[56] Rane NL, Paramesha M, Choudhary SP, Rane J. Machine learning and deep learning for big data analytics: a review of methods and applications. Partners Universal International Innovation Journal. 2024; 2(3):172-97.
[57] Fadillah WM, Mutiara GA, Periyadi P, Alfarisi MR, Meisaroh L. Vicinity monitoring of military vehicle cabin to improve passenger comfort with fusion sensors and LoRa RFM95W. Journal of Robotics and Control. 2024; 5(5):1216-26.
[58] Koo KY, Battista ND, Brownjohn JM. SHM data management system using MySQL database with MATLAB and web interfaces. In 5th international conference on structural health monitoring of intelligent infrastructure (shmii-5), Cancún, México 2011 (pp. 589-96).
[59] Sharanagowda K. A study on the client server architecture and its usability. IOSR Journal of Computer Engineering. 2022; 24(4):73-6.
[60] Du W, Li J, Wang Y, Chen L, Zhao R, Zhu J, et al. Vulnerability-oriented testing for restful APIS. In 33rd USENIX security symposium (USENIX Security 24) 2024 (pp. 739-55). USENIX Association.

Dr. Giva Andriana Mutiara completed her Ph.D. in Computer Science from Universiti Teknikal Malaysia Melaka in 2022. She is a lecturer in the Diploma in Computer Technology program and serves as an Associate Professor in the Department of Applied Science at Universitas Telkom, Indonesia. Currently, she leads the Center of Excellence for Smart Technology and Applied Science Research Group (RG), with her research focusing on Smart Technology and the Internet of Things (IoT). She holds several intellectual property rights and patents, and her work is widely published in reputable, indexed journals and databases.
Email: givamz@telkomuniversity.ac.id

Periyadi holds a Master's degree in Computer Engineering with a specialization in Computer Systems, Networks, and Security, with a focus on applied sciences. As a dedicated lecturer at Telkom University, he has actively contributed to both academic research and teaching in his area of expertise. His scholarly work is reflected in numerous publications indexed in reputable databases. In addition, he holds several intellectual property rights and patents, demonstrating his commitment to practical innovation and advancing the fields of network and security within applied sciences.
Email: periyadi@telkomuniversity.ac.id

Muhammad Rizqy Alfarisi is a Master's graduate in Computer Enginnering from Bandung Institute of Technology, specialization in Internet of Thing and Computer Science. Currently he is a lecturer at Faculty of Applied Science, Telkom University and one of the researchers at STAS RG laboratory. His work is reflected in various publications indexed in reputable database.
Email: mrizkyalfarisi@telkomuniversity.ac.id

Muhammad Ghifar Rijali is currently a student at Telkom University majoring Multimedia Engineering Technology. He is actively contributed as a researcher of Smart Technology and applied Science RG laboratory, focusing on the development of website and software applications.
Email: muhammadghifarrijali@student.telkomuniversity.ac.id

Muhammad Aulia Rifqi Zain currently a student of Telkom University majoring Multimedia Engineering Technology. He invoved in several research that focus on Internet of Things such in Smart Technology and Applied Science Laboratory. He is focused on embedded system and industrial research solution.
Email: muhammadauliarifqi@student.telkomuniversity.ac.id

Fathurrohman Nur Rochim is a passionate student at Telkom University, majoring in Multimedia Engineering Technology. He actively contributes to projects at the Smart Technology and Applied Science Laboratory, where he is involved in several research initiatives. His primary focus is on front-end design, showcasing both creativity and technical skill in user interface development.
Email: makelartahu@student.telkomuniversity.ac.id

Appendix I

S. No.	Abbreviation	Description
1	API	Application Programming Interface
2	CPU	Central Processing Unit
3	CSRF	Cross-Site Request Forgery
4	CRUD	Create, Read, Update, Delete
5	DBMS	Database Management System
6	EDA	Event-Driven Architecture
7	FSR-7548	Force Sensing Resistor
8	GraphQL	Graph Query Language
9	HTTP	Hypertext Transfer Protocol
10	IOT	Internet of Things
11	JSON	JavaScript Object Notation
12	JWT	JSON Web Token
13	LNS	LoRaWAN Network Server
14	LORA	Long Range
15	LORAWAN	Long Range Wide Area Network
16	MQTT	Message Queueing Telemetry Transport
17	MySQL	My Structured Query Language
18	P2P	Peer-to-Peer
19	PCR	Postman Collection Runner
20	REST	Representational State Transfer
21	SOA	Service-Oriented Architecture
22	SOAP	Simple Object Access Protocol
23	SQLMap	Structured Query Language Map
24	TLS/SSL	Transport Layer Security / Secure Sockets Layer
25	URL	Uniform Resource Locators
26	WiFi	Wireless Fidelity