Prof. Dr. Armin Lehmann **E-Mail:**lehmann@e-technik.org **Internet:** www.e-technik.org	Frankfurt University of Applied Sciences Forschungsgruppe für Tele- kommunikationsnetze

Mobile Computing

Winter Semester 2019/2020

"IoT Smart-Home: Assisted Training"

By

Harshal Vaze

Matriculation number: **1269879**

Course: **M.Eng in Information Technology**

Under the guidance of

Prof. Dr. Armin Lehmann

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Table Of Contents

1. Abstract:

A person who is fit is capable of living life to its fullest extent. Physical and mental fitness play very important roles in our lives and people who are both, physically and mentally fit are less prone to medical conditions as well. The increasing technologies in fitness are great benefits for everybody and Internet of Things (IoT) moves this to the whole new level. IoT facilitates the user with Real-Time data and it is a close integration of virtual and real worlds in which user interact with Smart-Objects. In this project a Java Application is built comprising of IoT Gateway, IoT Environment and Web service. IoT Gateway handles the requests and responses from IoT devices using CoAP protocol and from Web application using HTTP protocol, different protocols are utilized for communication between different interfaces with different data encoding schemes. Two use-cases are developed to illustrate the working of the application as per project requirements. Technologies used are HTTP and CoAP protocols, JSON and XML encoding schemes, Java Spring-Boot framework, HTML5 and JavaScript.

Keywords: IoT, Smart-Home, Fitness, Java Application

2. Design Architecture:

In this project, twelve emulated Internet of Things (IoT) devices (six sensors and six actuators) are monitored and controlled comfortably and automatically via a Web Service. The IoT devices offer a communication interface based on the Constrained Application Protocol (CoAP). An IoT gateway is created for providing a possible communication between the web service and the CoAP-based IoT devices. The IoT gateway enables the translation of HTTP messages to a corresponding CoAP messages and vice versa. For this purpose, the IoT-Gateway consists of a web server providing the required web service and a CoAP-Component which can communicate with the devices in the IoT-Environment via their individual CoAP-Interface. The web service also enables the realisation of a self-defined and configurable Smart-Home use-case, which uses the functionality of the emulated IoT-Devices. The logical interconnections of the different components can be seen in the following figure:

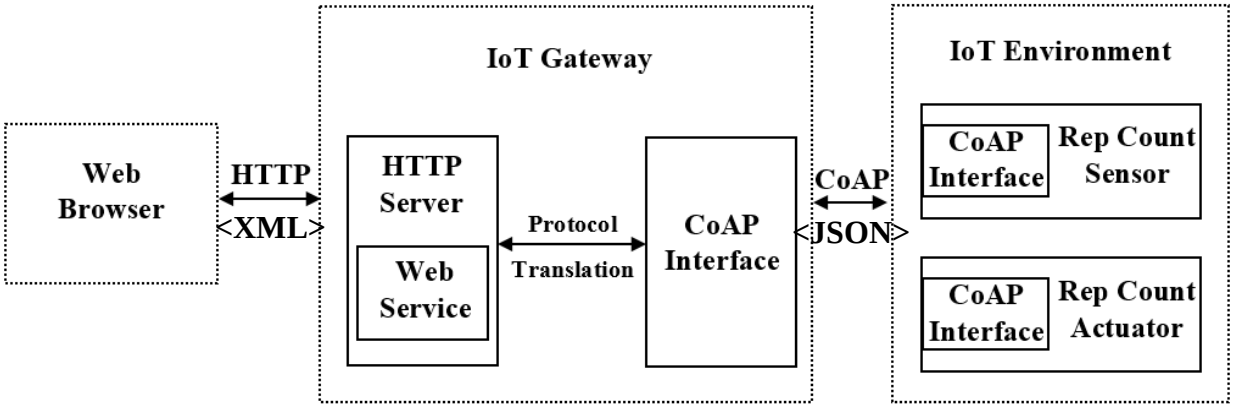


Fig. 2.1 Design Architecture

This project illustrates the use of smart devices in Assisted Training or for Fitness. Two use-case are implemented in this project, one is Smart-Cycling and another is Smart-Weight Training.

2.1 Smart-Cycling:

In this use-case, the user performs the cycling or the spinning exercise in his/her home. The bicycle used by user is facilitated with the various smart sensors. These sensors help the user to track his exercise progress and achieve his goal more accurately. As the user increases the speed of the cycling, the calories consumption and heart rate increases exponentially. In total three smart sensors are deployed in this use-case, namely

* **Speed Count Sensor:** User is incapable to count the speed of the bicycle by his own, hence a smart sensor is used for acquiring the speed with which the user is cycling.
* **Heart Rate Sensor:** Depending on the speed of the cycle, the user's heart rate changes. A smart sensor is used for acquiring the heart rate of the user which changes significantly as the speed of the cycling increases.

* **Calories Count Sensor:** Calories count is the most important aspect in the fitness. Depending on the speed of the cycle, the user's calories consumption also changes. A smart sensor is used for acquiring the calories burnt during the exercise of the user which also changes significantly as the speed of the cycling increases.

2.2 Smart-Weight Training:

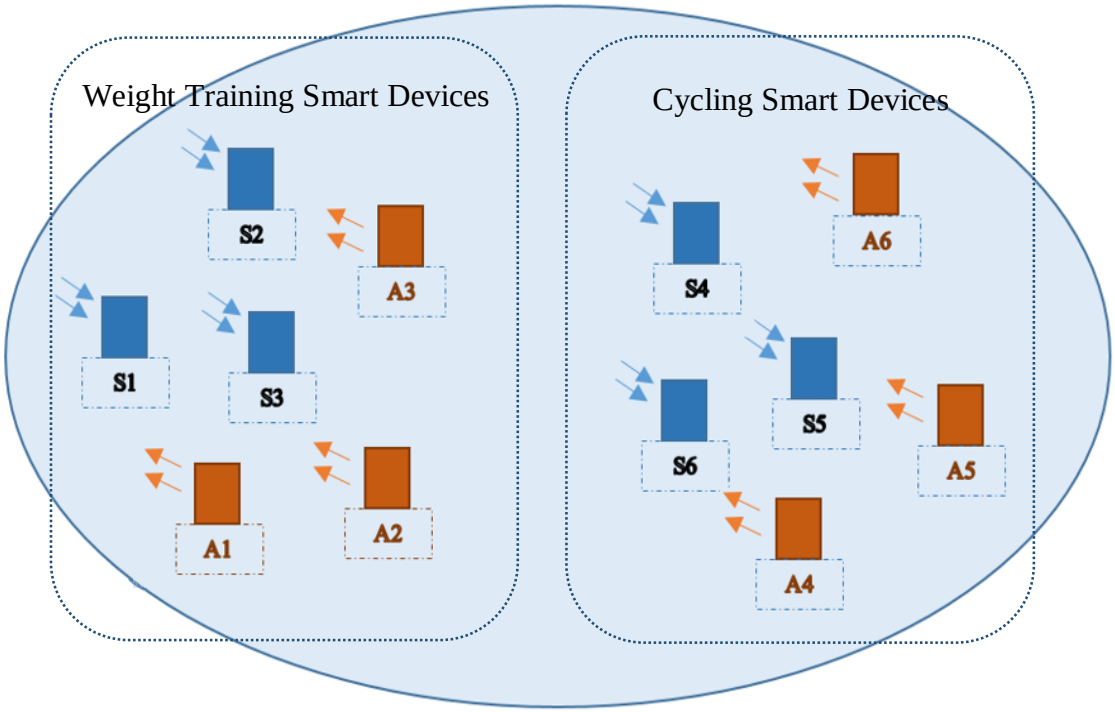
In this use-case, the user performs the weight training exercise (in this case, Bench Press exercise is implemented) in his/her home. The equipment used by user is facilitated with the various smart sensors. These sensors help the user to track his exercise progress and achieve his goal more accurately. As the user performs higher number of repetitions, the calories consumption and heart rate increases exponentially. In total three smart sensors are deployed in this use-case, namely

* **Rep (Repetition) Count Sensor:** Many times user forgets the count of repetitions while performing the exercise. A smart sensor used for acquiring the number of repetitions user performed while executing this exercise.
* **Heart Rate Sensor:** Depending on the number of repetitions, the user's heart rate changes. A smart sensor is used for acquiring the heart rate of the user which changes significantly as the number of repetitions performed by user increases.
* **Calories Count Sensor:** Depending on the number repetitions, the user's calories consumption also changes. A smart sensor is used for acquiring the calories burnt during the exercise of the user which also changes significantly as the number of repetitions performed by user increases.

All these smart devices are attached to the smart cycle and the apparatus used for weight training, the user doesn't need to wear any extra gadgets.

For this project, all these smart devices are assembled in the emulated IoT environment in which user entity is quite difficult to implement, consequently the smart actuators are build. The smart actuators perform two activities here, one being acting as user entity and another is displaying the respective parameters on the display. In total six smart actuators are deployed in this project, three for each use-case.

EMULATED IoT ENVIRONMENT



Devices Description:

S1 - Rep Count Sensor
S2 - Calories Count Sensor
S3 - Heart Rate Sensor
S4 - Speed Count Sensor
S5 - Calories Count Sensor 2*
S6 - Heart Rate Sensor 2*

A1 - Rep Count Actuator
A2 - Calories Count Actuator
A3 - Heart Rate Actuator
A4 - Speed Count Actuator
A5 - Calories Count Actuator 2*
A6 - Heart Rate Actuator 2*

* These smart devices are part of Smart-Cycling use-case.

3. Constrained Application Protocol (CoAP):

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained networks in the Internet of Things. The protocol is designed for machine-to-machine (M2M) applications such as smart-home and building automation. The interaction model of CoAP is similar to the client/server model of HTTP. However, machine-to-machine interactions typically result in a CoAP implementation acting in both client and server roles. A CoAP request is equivalent to that of HTTP and is sent by a client to request an action on a resource on a server. The server then sends a response with a Response Code, this response may include a resource representation.

Unlike HTTP, CoAP deals with these interchanges asynchronously over a datagram-oriented transport such as UDP. This is done logically using a layer of messages that supports optional reliability (with exponential back-off). CoAP defines four types of messages: **Confirmable, Non-confirmable, Acknowledgement, Reset**. Method Codes and Response Codes included in some of these messages make them carry requests or responses. The basic exchanges of the four types of messages are somewhat orthogonal to the request/response interactions; requests can be carried in Confirmable and Non-confirmable messages, and responses can be carried in these as well as piggybacked in Acknowledgement messages.

CoAP makes use of GET, PUT, POST, and DELETE methods in a similar manner to HTTP. Methods beyond the basic four can be added to CoAP in separate specifications. New methods do not necessarily have to use requests and responses in pairs. Even for existing methods, a single request may yield multiple responses, e.g., for a multicast request or with the Observe option. URI support in a server is simplified as the client already parses the URI and splits it into host, port, path, and query components, making use of default values for efficiency. Response Codes relate to a small subset of HTTP status codes with a few CoAP-specific codes added.

3.1 Messaging Model:

The CoAP messaging model is based on the exchange of messages over UDP between endpoints. CoAP uses a short fixed-length binary header (4 bytes) that may be followed by compact binary options and a payload. This message format is shared by requests and responses. Each message contains a Message ID used to detect duplicates and for optional reliability. Reliability is provided by marking a message as Confirmable (CON). A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions, until the recipient sends an Acknowledgement message (ACK) with the same Message ID from the corresponding endpoint (fig.3.1). As CoAP runs over UDP, it also supports the use of multicast IP destination addresses, enabling multicast CoAP requests.

3.2 Request/Response Model:

CoAP request and response semantics are carried in CoAP messages, which include either a Method Code or Response Code. Optional request and response information, such as the URI and payload media type are carried as CoAP options. A Token is used to match responses to requests independently from the underlying messages. The Token is a concept separate from the Message ID (fig.3.1).

A request is carried in a Confirmable (CON) or Non-confirmable (NON) message and, if immediately available, the response to a request carried in a Confirmable message is carried in the resulting Acknowledgement (ACK) message. This is called a piggybacked Response.

3.3 Message Transmission:

CoAP messages are exchanged asynchronously between CoAP endpoints. They are used to transport CoAP requests and responses. As CoAP is bound to unreliable transports such as UDP, CoAP messages may arrive out of order, appear duplicated or go missing without notice. For this reason, CoAP implements a lightweight reliability mechanism, without trying to re-create the full feature set of a transport like TCP. It has the following features:

* Simple stop-and-wait retransmission reliability with exponential back-off for Confirmable messages.
* Duplicate detection for both Confirmable and Non-confirmable messages.

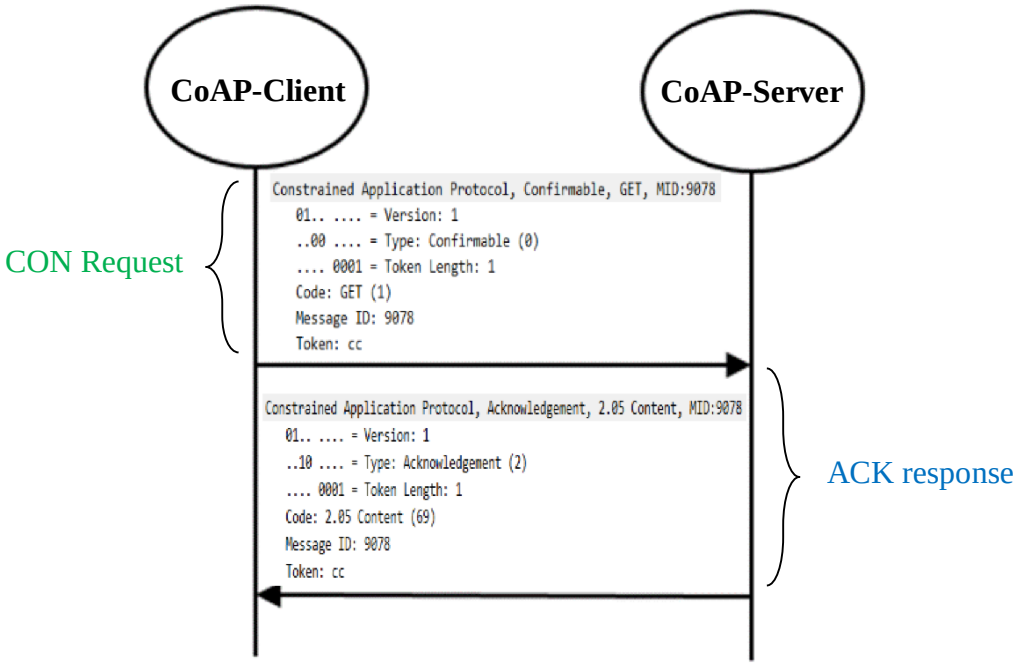


Fig.3.1 : CoAP Message Transmission

3.4 Message Format:

CoAP is based on the exchange of compact messages that, by default, are transported over UDP (i.e., each CoAP message occupies the data section of one UDP datagram). CoAP may also be used over Datagram Transport Layer Security (DTLS). CoAP messages are encoded in a simple binary format. The message format starts with a fixed-size 4-byte header. This is followed by a variable-length Token value, which can be between 0 and 8 bytes long.

Following the Token value comes a sequence of zero or more CoAP Options in Type-Length-Value (TLV) format, optionally followed by a payload that takes up the rest of the datagram.

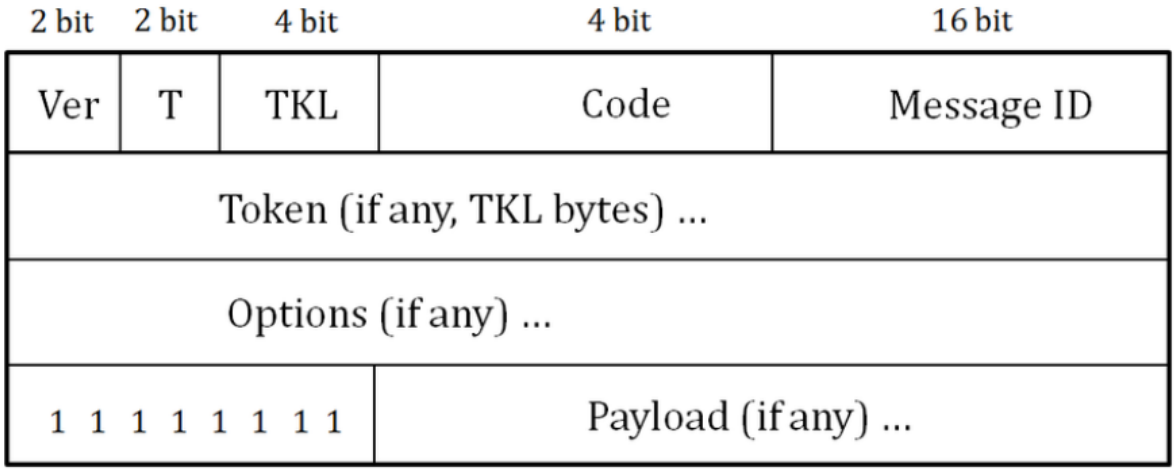


Fig.3.2 : CoAP Message Format

The fields in the header are defined as follows:

- **Version (Ver):** 2-bit unsigned integer. Indicates the CoAP version number. Implementations of this specification MUST set this field to 1 (01 binary). Other values are reserved for future versions. Messages with unknown version numbers MUST be silently ignored.
- **Type (T):** 2-bit unsigned integer. Indicates if this message is of type Confirmable (0), Non-confirmable (1), Acknowledgement (2), or Reset (3).
- **Token Length (TKL):** 4-bit unsigned integer. Indicates the length of the variable-length Token field (0-8 bytes). Lengths 9-15 are reserved, MUST NOT be sent, and MUST be processed as a message format error.
- **Code:** 8-bit unsigned integer, split into a 3-bit class (most significant bits) and a 5-bit detail (least significant bits), documented as "c.dd" where "c" is a digit from 0 to 7 for the 3-bit subfield and "dd" are two digits from 00 to 31 for the 5-bit subfield. The class can indicate a request (0), a success response (2), a client error response (4), or a server error response (5). (All other class values are reserved.) As a special case, Code 0.00 indicates an Empty message. In case of a request, the Code field indicates the Request Method; in case of a response, a Response Code. Possible values are maintained in the CoAP Code Registries.
- **Message ID:** 16-bit unsigned integer in network byte order. Used to detect message duplication and to match messages of type Acknowledgement or Reset to messages of type Confirmable or Non-confirmable.

4. Hypertext Transfer Protocol (HTTP):

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1".

4.1 HTTP Message:

HTTP messages consist of requests from client to server and responses from server to client. Request and Response messages use the generic message format for transferring entities (the payload of the message). Both types of message consist of a start-line, zero or more header fields, an empty line indicating the end of the header fields and possibly a message-body.

Message Headers: HTTP header fields, which include general-header, request-header, response-header and entity-header fields, follow the same generic format. Each header field consists of a name followed by a colon (":") and the field value.

Message Body: The message-body of an HTTP message is used to carry the entity-body associated with the request or response. The message-body differs from the entity-body only when a transfer-coding has been applied, as indicated by the Transfer-Encoding header field.

* The presence of a message-body in a request is signalled by the inclusion of a Content-Length or Transfer-Encoding header field in the request's message-headers.
* For response messages, whether or not a message-body is included with a message is dependent on both the request method and the response status code.

4.2 HTTP Methods:

HTTP uses **GET, HEAD, POST, PUT, DELETE, TRACE** and **CONNECT** methods.

Safe Methods: Implementers should be aware that the software represents the user in their interactions over the Internet and should be careful to allow the user to be aware of any actions they might take which may have an unexpected significance to themselves or others. In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe". This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.

Idempotent Methods: Methods can also have the property of idempotence in that the side-effects of $N > 0$ identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE share this property. Also, the methods OPTIONS and TRACE SHOULD NOT have side effects, and so are inherently idempotent.

5. Principles of REST:

* Resources expose easily understood directory structure URIs.
* Representations transfer JSON or XML to represent data objects and attributes.
* Messages use HTTP methods explicitly (for example, GET, POST, PUT, and DELETE).
* Stateless interactions store no client context on the server between requests. State dependencies limit and restrict scalability. The client holds session state.

5.1 REST Methods:

i. GET:
Retrieve information. GET requests must be safe and idempotent, meaning regardless of how many times it repeats with the same parameters, the results are the same.

ii. PUT:
Store an entity at a URI. PUT can create a new entity or update an existing one. A PUT request is idempotent. Idempotency is the main difference between the expectations of PUT versus a POST request

iii. POST:
Request that the resource at the URI do something with the provided entity. Often POST is used to create a new entity, but it can also be used to update an entity.

iv. DELETE:
Request that a resource be removed; however, the resource does not have to be removed immediately. It could be an asynchronous or long-running request.

6. Application Logic:

* For this project, the implementation is performed on **Java platform** inclined with **REST** architecture (GET, PUT, POST, DELETE). There are three main components of this project **Web Browser**, **IoT Gateway** and **Emulated IoT Environment**. Protocol used to transfer messages between Web Browser and IoT Gateway is **HTTP** whereas protocol used to transfer messages between IoT Gateway and Emulated IoT Environment is **CoAP**. Also messages between Web Browser and IoT Gateway are sent as **XML** format while messages between IoT Gateway and Emulated IoT Environment are sent as **JSON** format.

* Main libraries we are using here are **Java library Californium** and **Jackson.**

* Sensors and Actuators are deployed in an Emulated IoT Environment and IoT Gateway acts as a middleware between Web Browser and Emulated IoT Environment. All the transactions occur through IoT Gateway.

6.1 Sensors:

i. RepCountSensor.java

This class handles the Rep Count Sensor used for Weight Training use-case. It reads sensor data from the RepCount.txt file. The *getRepCount()* is the method used in this class.

ii. CaloriesCountSensor.java

This class handles the Calories Count Sensor used for Weight Training use-case. It reads sensor data from the Calories.txt file. The *getCaloriesCount()* is the method used in this class.

iii. HeartRateSensor.java

This class handles the Heart Rate Sensor used for Weight Training use-case. It reads sensor data from the HeartRate.txt file. The *getHeartRate()* is the method used in this class.

iv. SpeedCountSensor.java

This class handles the Speed Count Sensor used for Cycling use-case. It reads sensor data from the SpeedCount.txt file. The *getSpeedCount()* is the method used in this class.

v. CaloriesountSensor2.java

This class handles the Calories Count Sensor used for Cycling use-case. It reads sensor data from the Calories.txt file. The *getCaloriesCount2()* is the method used in this class.

vi. HeartRateSensor2.java

This class handles the Heart Rate Sensor used for Cycling use-case. It reads sensor data from the HeartRate2.txt file. The *getHeartRate2()* is the method used in this class.

- *handleGET()* is common method for all the Sensor classes.

6.2 Actuators:

i. RepCountActuator.java

This class handles the Rep Count Actuator used for Weight Training use-case. It writes the data in the RepCount.txt file. The *setRepCount()* is the method used in this class.

ii. CaloriesActuator.java

This class handles the Calories Count Actuator used for Weight Training use-case. It writes the data in the Calories.txt file. The *setCaloriesCount()* is the method used in this class.

iii. HeartRateActuator.java

This class handles the Heart Rate Actuator used for Weight Training use-case. It writes the data in the HeartRate.txt file. The *setHeartRate()* is the method used in this class.

iv. SpeedCountActuator.java

This class handles the Speed Count Actuator used for Cycling use-case. It writes the data in the SpeedCount.txt file. The *setSpeedCount()* is the method used in this class.

v. CaloriesActuator2.java

This class handles the Calories Count Actuator used for Cycling use-case. It writes the data in the Calories.txt file. The *setCaloriesCount2()* is the method used in this class.

vi. HeartRateActuator2.java

This class handles the Heart Rate Actuator used for Cycling use-case. It writes the data in the HeartRate2.txt file. The *setHeartRate2()* is the method used in this class.

- *handlePOST()* is common method for all the Actuator classes.

6.3 IoT Gateway:

This class as a middleware between Web Browser and Emulated IoT Environment. It controls the HTTP requests and consequently provides the HTTP responses. On the other hand, it also operates the CoAP responses and makes CoAP requests. Since there are two message formats used in this whole communication, IoT Gateway is responsible to convert JSON data to XML data and vice versa.

- Methods used in this class:

getRepCounts() - gets all the sensors data of Weight Training use-case.

getCyclingCounts() - gets all the sensors data of Cycling use-case.

setRepCounts() - sets the actuators data of Weight Training use-case.

setCyclingCounts() - sets all the actuators data of Cycling use-case.

setNextExercise() - resets all the sensors and actuators data of both the use-case.

6.4 Functionality:

* A simple user friendly HTML graphical user interface (GUI) is designed to handle the java application, the client can control the complete functionality of the application from the web browser.
* All the HTTP requests (*http://localhost:8080*) made from the client are transferred to IoT Gateway (fig.6.4.1), IoT Gateway translates the HTTP protocol to CoAP protocol and then CoAP requests (*coap://localhost:{port number}*) are made from the IoT Gateway to the sensors and actuators in the IoT Environment (fig.6.4.2).
* CoAP responses from these sensors and actuators in the IoT Environment are transmitted to the IoT Gateway (fig.6.4.2), IoT Gateway translates the CoAP protocol back to HTTP protocol and then HTTP responses are transferred from IoT Gateway to the web browser (fig.6.4.1).
* IoT Gateway is also responsible for making some decisions depending on the use-case.

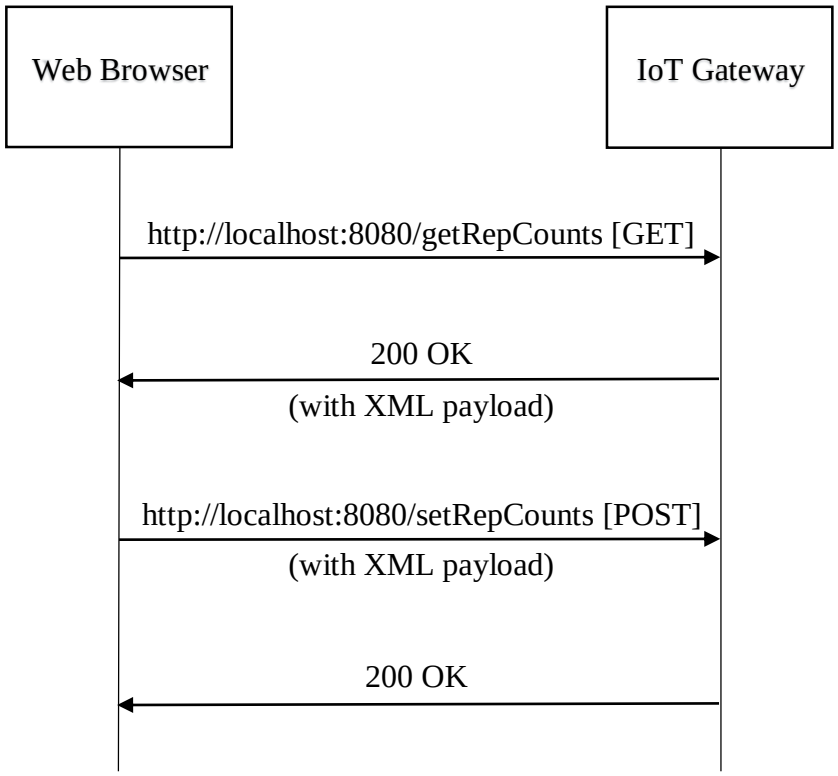


Fig.6.4.1 HTTP Message Sequence Diagram

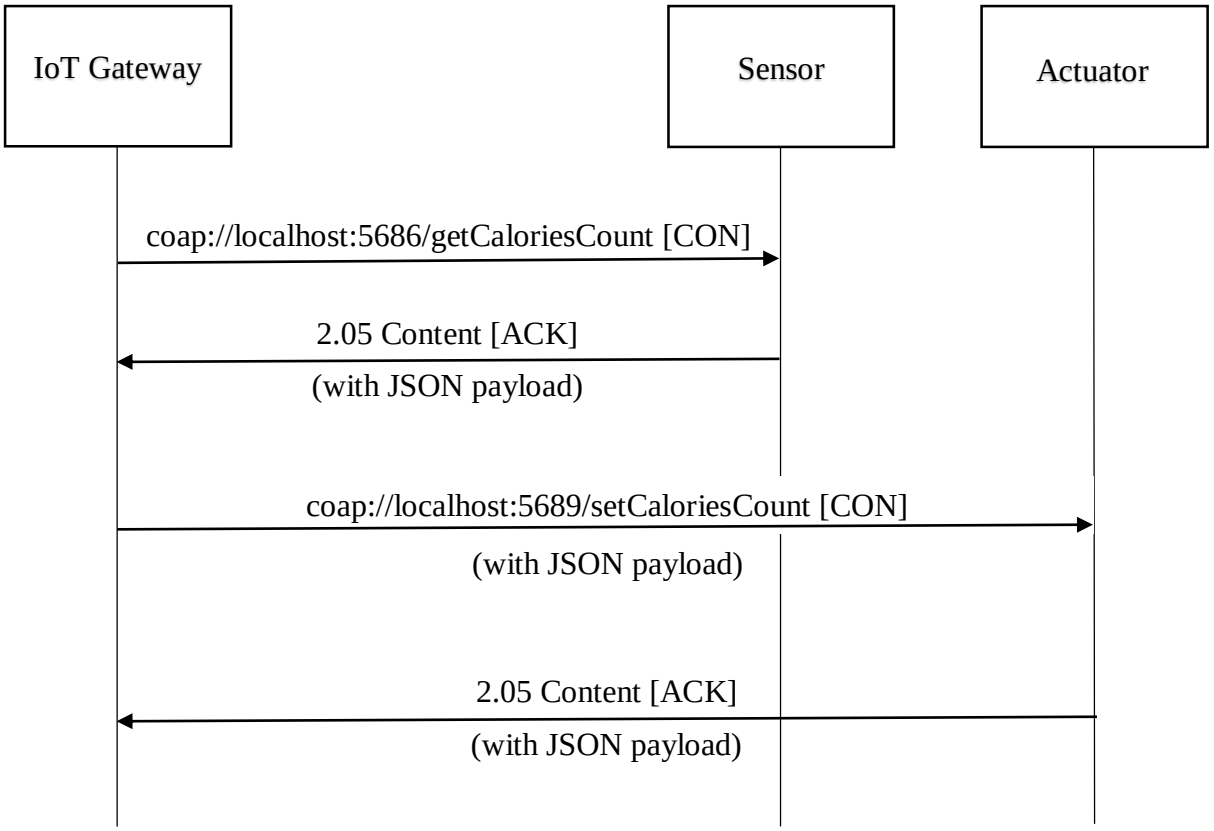


Fig.6.4.2 CoAP Message Sequence Diagram

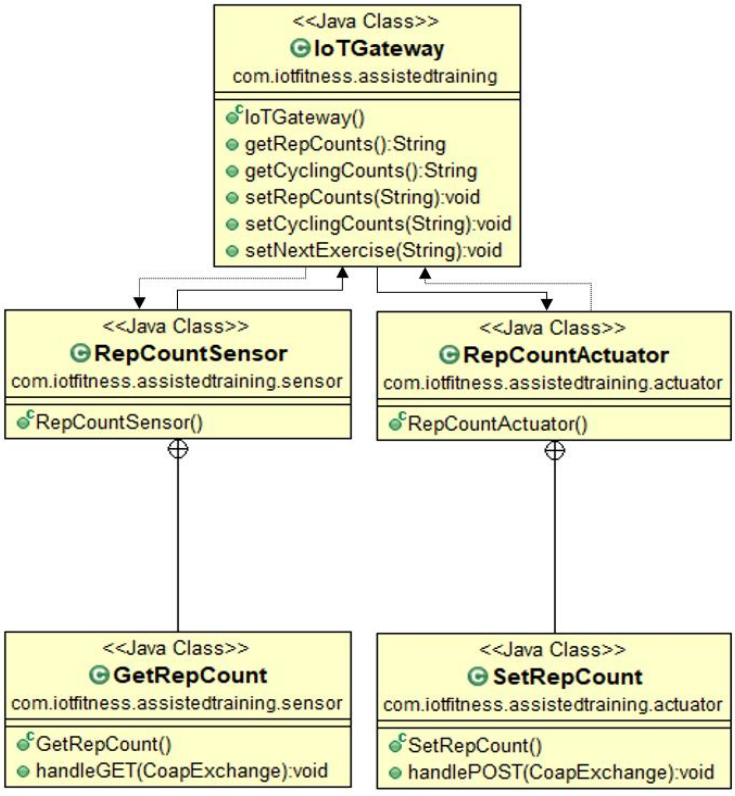


Fig.6.4.3 UML Activity Diagram showing one of the Sensor and Actuator communication with IoT Gateway and describing Methods in the classes

7. Wireshark Captures:

The Wireshark is started on loopback interface to capture the messages between the Web Browser and IoT Gateway transactions (HTTP protocol) and also communications between IoT Gateway and IoT Emulated Environment (CoAP protocol).

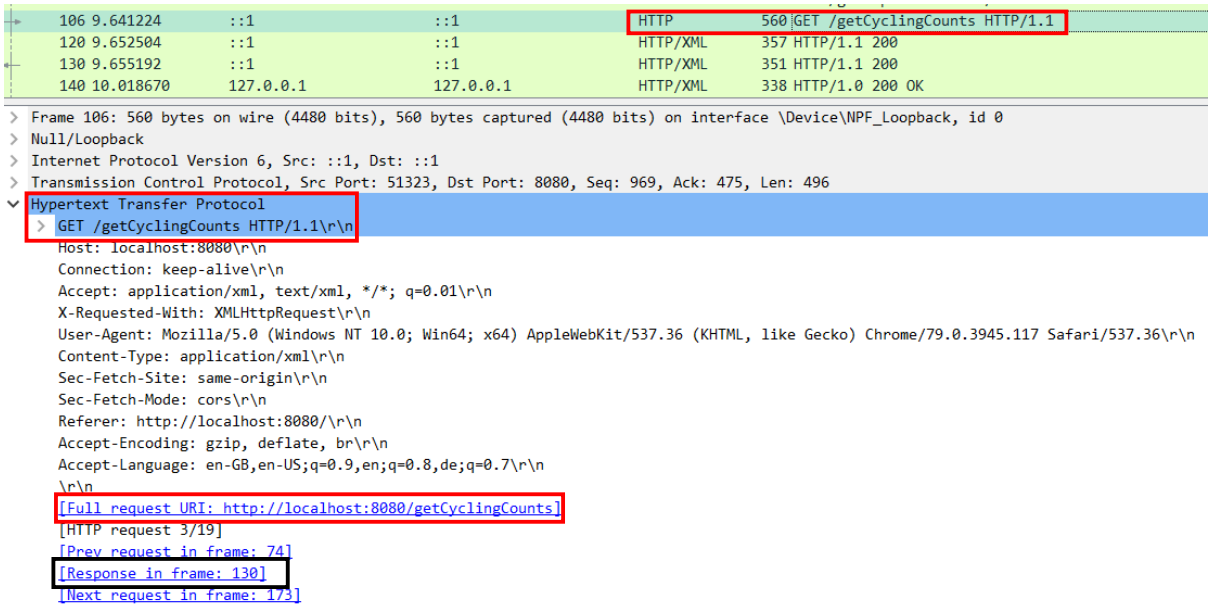


Fig.7.1 HTTP GET Request from Web Browser to IoT Gateway

When client hits the (*http://localhost:8080/getCyclingCounts*) or requests the Cycling Counts data through GUI, web browser sends the HTTP request to the IoT Gateway as shown in fig.7.1.

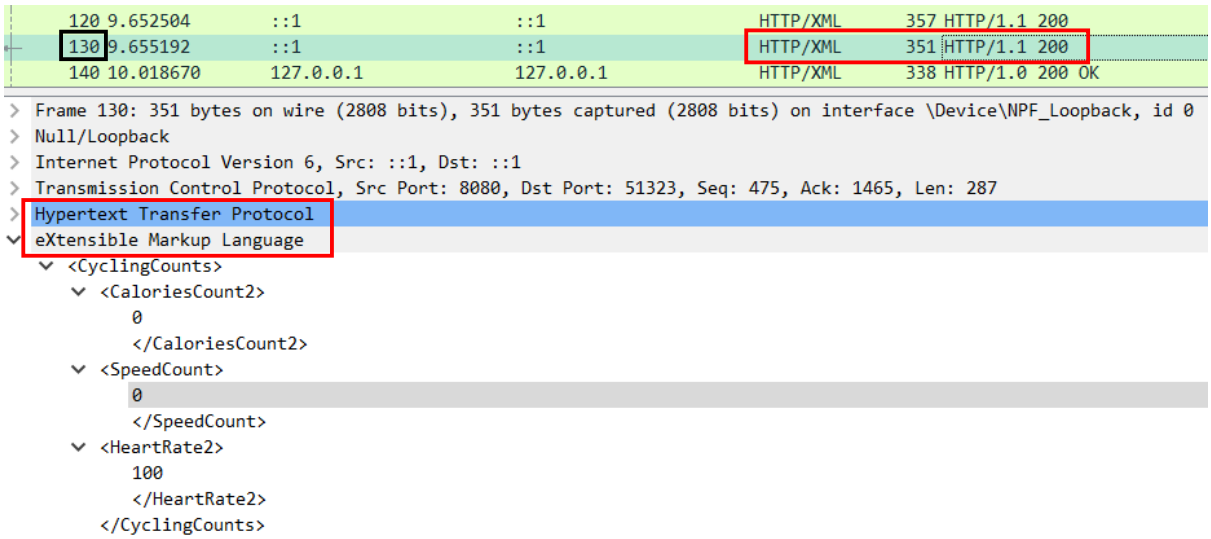


Fig.7.2 HTTP Response from IoT Gateway to Web Browser with XML payload

IoT Gateway send the HTTP response to the web browser with the data in XML encoding scheme as shown in fig.7.2.

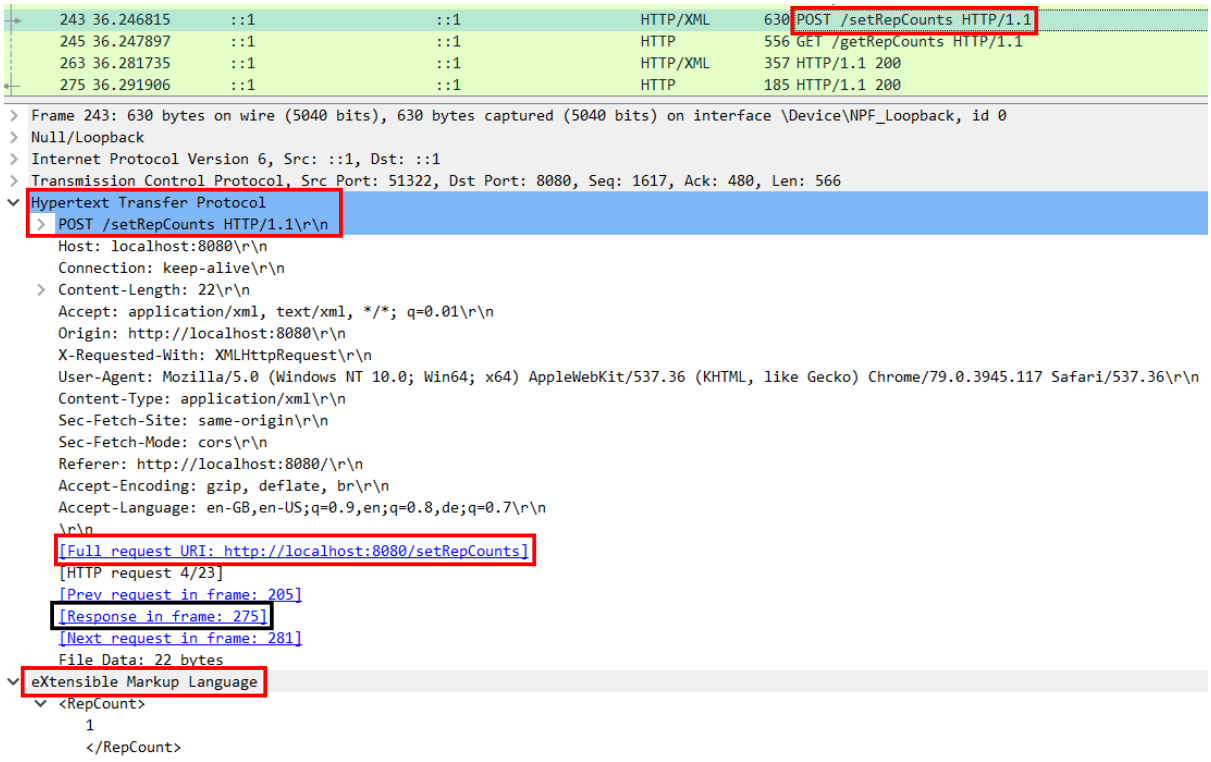


Fig.7.3 HTTP POST Request from Web Browser to IoT Gateway with XML payload

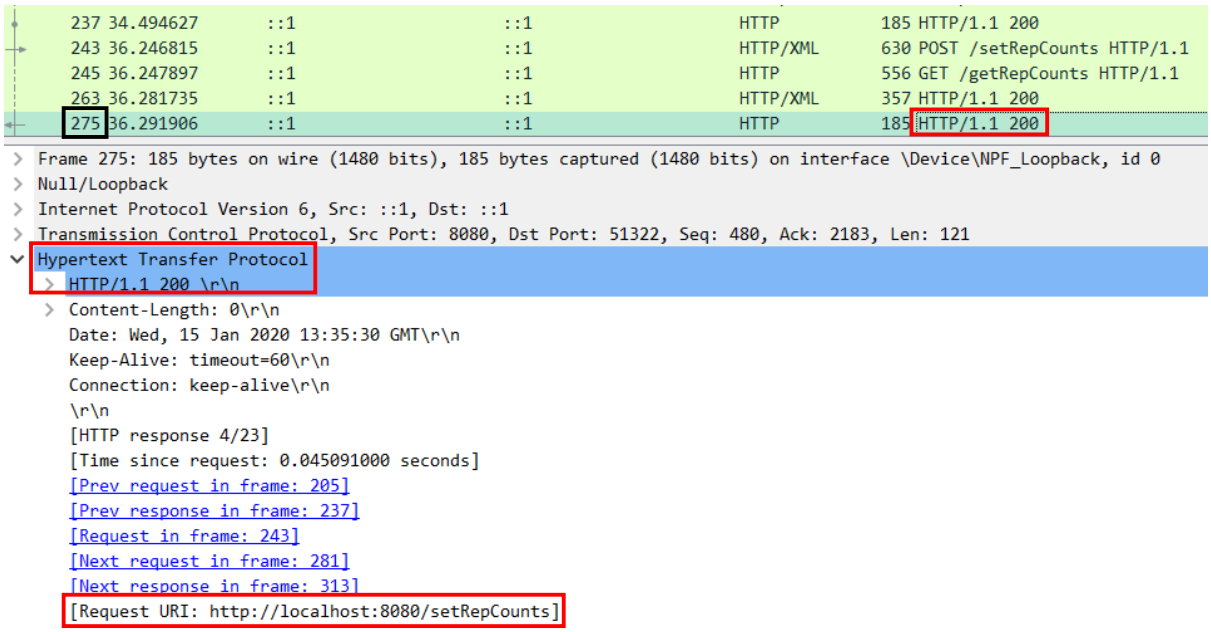


Fig.7.4 HTTP Response from IoT Gateway to Web Browser

The sequence number (Seq) of the HTTP response is same as acknowledgement number (Ack) of HTTP request.

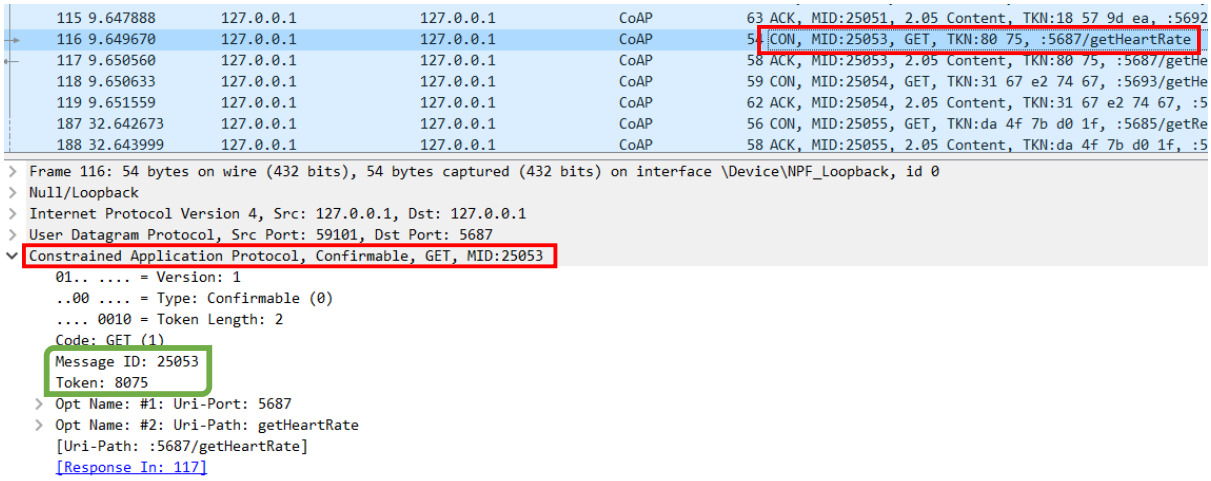


Fig.7.5 CoAP GET Request from IoT Gateway to Sensor

The CoAP requests are made as CON here, which means Confirmable requests and the CoAP responses are sent as the ACK Acknowledgement to the same request.

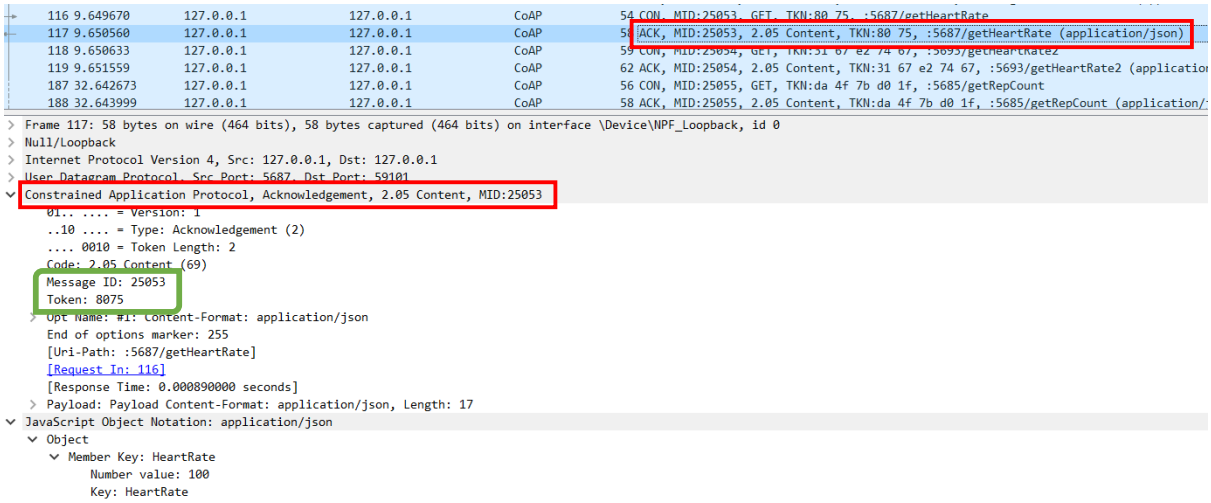


Fig.7.6 CoAP Response from Sensor to IoT Gateway with JSON payload

Note that the Message ID (MID) and Token for CoAP response messages are same as that of CoAP request messages. From fig.7.5 and fig.7.6 we observe that the MID and Token are same for both the packets, hence it is the response to the GET request made by the IoT Gateway to the Sensor.

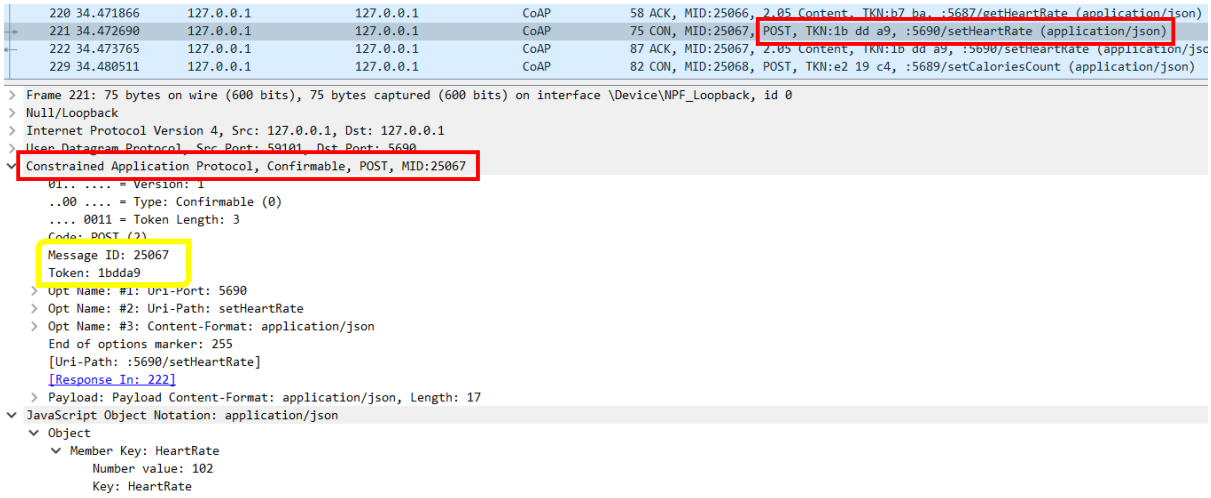


Fig.7.7 CoAP POST Request from IoT Gateway to Actuator with JSON payload

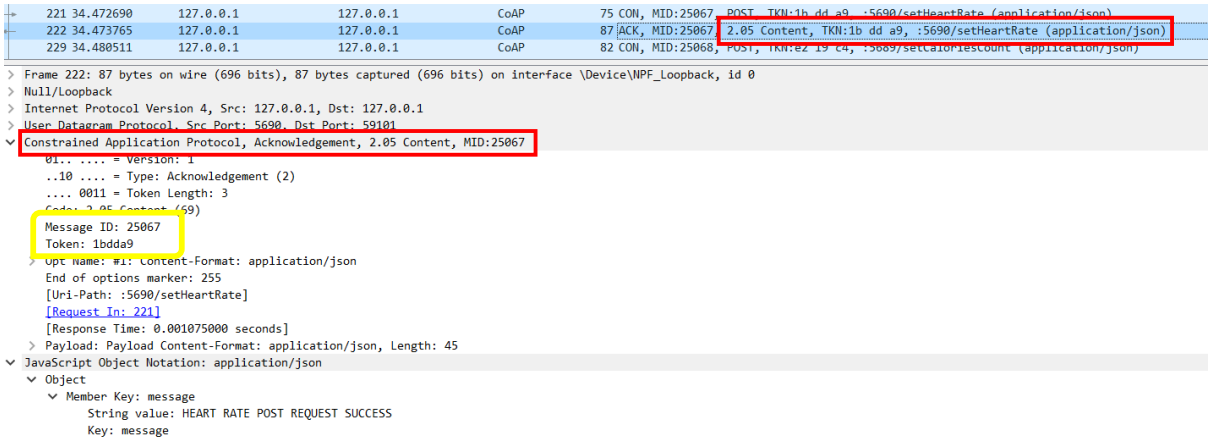


Fig.7.8 CoAP Response from Actuator to IoT Gateway with JSON payload

8. Steps to Run the Application:

The Jar file for the application is generated through the Eclipse.

i. Open cmd in Windows or terminal in Linux.
ii. Change directory to where the **assistedtraining.jar** file is located.
iii. Before running the application make sure that none of the ports used by the project are used by some other application on the machine.
iv. Type: **java -jar assistedtraining.jar** and hit Enter.
v. All the CoapEndpoints first start running and then the Application starts running.
vi. Californium.properties and all the necessary files are generated in the folder containing **assistedtraining.jar** file.
vii. Open the Web Browser and type: http://localhost:8080 and hit Enter.
viii. A user friendly HTML GUI will appear to handle the java application, the user can control the complete functionality of the application from the web browser (fig.8.2).
ix. Start Wireshark and capture on loopback interface.
x. All the data transactions can be observed on console window as shown in fig.8.1.

```
2.05
2.05
{"Content-Format":"application/json"}
{"Content-Format":"application/json"}
{"RepCount":0}
{"SpeedCount":0}
2.05
2.05
{"Content-Format":"application/json"}
{"CaloriesCount":0}
{"Content-Format":"application/json"}
{"CaloriesCount2":0}
2.05
{"Content-Format":"application/json"}
{"HeartRate":100}
2.05
{"Content-Format":"application/json"}
{"HeartRate2":100}
<WeightTrainingCounts><CaloriesCount>0</CaloriesCount><HeartRate>100</HeartRate><RepCount>0</RepCount></WeightTrainingCounts>
<CyclingCounts><CaloriesCount2>0</CaloriesCount2><SpeedCount>0</SpeedCount><HeartRate2>100</HeartRate2></CyclingCounts>
```

Fig.8.1 Console window after hitting http://localhost:8080

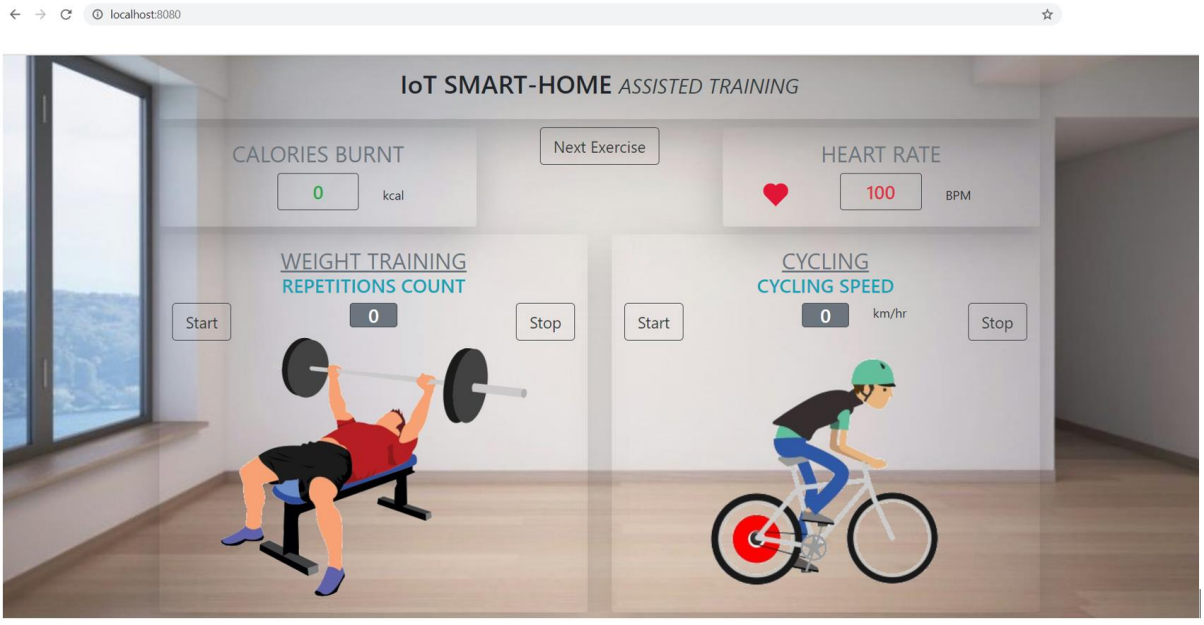


Fig.8.2 GUI on load

9. Appendix:

Port Number	Protocol	Description	URL
5685	CoAP	Rep Count Sensor	coap://localhost:5685/getRepCount
5686	CoAP	Calories Count Sensor	coap://localhost:5686/getCaloriesCount
5687	CoAP	Heart Rate Sensor	coap://localhost:5687/getHeartRate
5688	CoAP	Rep Count Actuator	coap://localhost:5688/setRepCount
5689	CoAP	Calories Count Actuator	coap://localhost:5689/setCaloriesCount
5690	CoAP	Heart Rate Actuator	coap://localhost:5690/setHeartRate
5691	CoAP	Speed Count Sensor	coap://localhost:5691/getSpeedCount
5692	CoAP	Calories Count Sensor Cycling	coap://localhost:5692/getCaloriesCount2
5693	CoAP	Heart Rate Sensor Cycling	coap://localhost:5693/getHeartRate2
5694	CoAP	Speed Count Actuator	coap://localhost:5694/setSpeedCount
5695	CoAP	Calories Count Actuator Cycling	coap://localhost:5695/setCaloriesCount2
5696	CoAP	Heart Rate Actuator Cycling	coap://localhost:5695/setHeartCount2
8080	HTTP	IoT Gateway interface	http://localhost:8080

Interface	Communication Protocol	Encoding Scheme
Web Browser and IoT Gateway	HTTP	XML
IoT Gateway and Emulated IoT Environment (Sensors and Actuators)	CoAP	JSON

10. Acknowledgement:

I am profoundly grateful to Prof. Dr. Lehmann, Mr. Frick and Mr. Shala for their advices and guidance during this course of work. Additionally, I would like to thank the open-source community for enabling a platform for free collaboration provided for CoAP and HTTP libraries. And also I would like to thank all my friends who helped me directly or indirectly during this course of work.

11. References:

i. https://tools.ietf.org/html/rfc7252
ii. https://tools.ietf.org/html/rfc2616
iii. https://www.w3.org/TR/2006/REC-xml11-20060816/
iv. https://tools.ietf.org/html/rfc8259