GATEWAY FOR PATIENT MOVEMENT CAPTURE SYSTEM

A PROJECT REPORT

Submitted by

CB.EN.U4CSE13362 SRUTI BALAN BANAI

in partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND TECHNOLOGY

AMRITA SCHOOL OF ENGINEERING, COIMBATORE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE 641112

JUNE 2017

AMRITA VISHWA VIDYAPEETHAM
AMRITA SCHOOL OF ENGINEERING,

COIMBATORE, 641112

BONAFIDE CERTIFICATE

This is to certify that the project report entitled “**GATEWAY FOR PATIENT MOVEMENT CAPTURE SYSTEM” submitted** by **CB.EN.U4CSE13362 SRUTI BALAN BANAI** in partial fulfilment of the requirements for the award of the Degree **Bachelor of Technology** in **“Computer Science and Engineering”** is a bonafide record of the work carried under our guidance and supervision at Amrita School of Engineering, Coimbatore and Cerner Healthcare.

External Guide
Mr. Bhanu Rana
Software Engineer, CareAware India
Cerner Healthcare India Private Ltd., Bangalore

Internal Guide
Dr.VENKATARAMAN D
Asst. Professor,
Department of CSE

Dr. Latha Parameswaran
Chairperson, Department of CSE

This project report was evaluated by us on ……………………

INTERNAL EXAMINER

EXTERNAL EXAMINER

Cerner

May 17, 2017

TO WHOMSOEVER IT MAY CONCERN

This is to certify that **Sruti Banai (SB051046)** ("Intern" hereafter) has undertaken an internship project from our organization **Cerner Healthcare Solutions Private Limited** ("Company" hereafter) from **January 19, 2017.**

During the internship Intern has undertaken project '**Patient Movement Capture System'** under the guidance of **Bhanu Rana, Software Engineer**, **BLR iBus Dev** successfully.

We wish the intern success in future endeavors.

Yours sincerely,
For Cerner Healthcare Solutions Private Limited,

Vivek Naik
Team Lead, HR Service Center
HRServiceCenter@cerner.com

ACKNOWLEDGEMENT

I would like to express my gratitude to all who have helped me directly or indirectly in my project. To start with, I thank the Almighty for all his blessings showered on me during the tenure of my project.

I express my sincere gratitude to Brahmachari Abhayamrita Chaitanya, Pro Chancellor and Dr.P.Venkat Rangan, Vice Chancellor and Dr.Sasangan Ramanathan, Dean Engineering of Amrita Vishwa Vidyapeetham, for providing me the opportunity to undergo this programme.

I express my sincere thanks to Prof. Dr.Latha Parameswaran, Chairperson, Department of Computer Science and Engineering, for her support and encouragement.

I extend my heartiest gratitude to my guide, Mr. Bhanu Rana (Cerner HealthCare India Private Ltd.) and Dr.VENKATARAMAN D for their valuable guidance. I thank my project coordinators, for their co-operation.

I am also grateful to all other members of faculty for their valuable guidance. Finally, I wish to express my sincere thanks to my parents and my friends who have contributed a lot towards my project work and my mental well-being during this period.

ABSTRACT

WHO defined health as the state of being completely fit in mind, body and health. Health is a major issue in all the countries in the world, which includes both developing and developed countries. Hence healthcare is a necessity in the world. According to several studies, it has been found that the third leading cause of death in U.S. is due to lack of proper healthcare. This being the case in U.S. we should think about the third world nations. For the health provider to provide proper care, he must know about the patient’s status and his/her movements time to time but the provider cannot keep a close watch on all his patients always. Hence we are introducing our Patient movement capture system to help patients get better care from their healthcare providers. The Patient Movement Capture System is a system used to monitor the movements of patients who are bedridden. The main purpose of the system is to monitor the patient using various sensors and alert the nurse or the physician if an abnormality occurs. It uses Internet of Things concept for the communication and exchange of data between the various sensors and the server. Additional functionalities like bedside device integration, checking the vitalities of the patient will also be implemented in the near future. For research purposes, we are concentrating on the movement monitoring module. Using web services, the application can be used in almost all platforms by only This system has been implemented in two platforms web application and android making it easily accessible. One of the important component is to create a gateway for communication between the user application and the sensor devices. This is the most significant modules in the project. This report explains the working and implementation of the gateway between the sensors and the user application.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ABBREVIATION	DEFINITION
EHR	Electronic Health Record
ReST	Representation Stateful Tranfer
IoT	Internet of Things
MOM	Message Oriented Middleware
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
JSON	JavaScript Object Notation
DDS	Demographic Data Server
MSMQ	Message Queuing
XML	Extensible Markup Language
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
UDDI	Universal Description, Discovery and Integration
HTTP	Hyper Text Transfer Protocol
CRUD	Create, Read, Update and Delete

CHAPTER 1

1. ORGANIZATION PROFILE

In this chapter, an introduction about Cerner Corporation and some of its products has been given.

Cerner Corporation

Cerner Corporation (Cerner), incorporated on October 6, 1986, deals with Health Care Information Technology (HCIT). The company offers a range of software, professional services, medical device integration, remote hosting and employer health and wellness services. It designs, develops, markets, installs, hosts and supports healthcare information technology, health care devices, hardware and content solutions for healthcare organizations and consumers. The company also provides range of value-added services, including implementation and training, remote hosting, operational management services, revenue cycle services, support and maintenance, healthcare data analysis, clinical process optimization, transaction processing, employer health centers, employee wellness programs and third-party administrator services for employer-based health plans. Cerner's systems are used by individual consumers, single-doctor practices, hospitals, etc.

Cerner solutions are currently licensed by approximately 9,300 facilities around the world, including more than 2,650 hospitals, 3,750 physician practices, 40,000 physicians, 500 ambulatory facilities, 800 home health facilities, 40 employer sites, and 1,600 retail pharmacies. The Company solutions are offered on the unified Cerner Millennium architecture, a person-centric computing framework, which combines clinical, financial and management information systems. This architecture allows providers to access an individual's Electronic Health Record (EHR) at the point of care, and it organizes and proactively delivers information to meet the specific needs of physicians, nurses, laboratory technicians, pharmacists, front and back-office professionals and consumers. The Company also offers the Millennium+ architecture, which leverages the cloud and enables greater mobility. It also created the Healthe Intent platform, a cloud-based platform that enables a new generation of solutions to leverage the increasing amount of data being captured as the healthcare industry is digitized. On the Healthe Intent platform, the Company is building EHR-agnostic solutions based on statistical algorithms that are intended to help providers predict and improve outcomes, control costs and improve quality.

Cerner Millennium

Cerner Millennium is a partnership of Cerner and HP. The architecture of Millennium allows caregivers and supporting providers, the ability to view results, problems, diagnosis, medications, and other pertinent information in real-time as well as share clinical and management data across multiple disciplines and facilities. This architecture has been referred to as Health Network Architecture (HNA), providing twelve major system applications operating by this means, fitting into four interrelated groups.

Millennium+

In 2012, Cerner launched Millennium+, which uses the Cerner Cloud to provide a user experience that is 'fast, smart and easy', enabling caregivers to have personalized, intuitive and moment relevant clinical work flows via desktop, tablet or smartphone with minimal orientation to begin usage.

One of the solutions that was launched as part of the Millennium+ platform was PowerChart+Touch™. PowerChart+Touch as a mobile solution allow physicians to complete workflows directly from their mobile devices and was created specifically for the iPad.

Key Benefits of Cerner Products

- Improve coordination and identification of patients
- Positively impact cash flow
- Access records at any time from any location
- Optimize workflow efficiency and performance

CHAPTER 2

INTRODUCTION

2.1 Overview

The main motivation of our project is to provide a 24/7 healthcare service for patients from their healthcare provider. A provider cannot always manually monitor a patient and there may be some situations where the patient wakes up from a long-time coma and wants immediate medical attention. Moreover, a physician or nurse cannot always be next to a patient to find out if he is back from coma. So, a patient movement capture system will prove to be a useful tool in all these situations. The system will monitor the patients around the clock and notify the healthcare provider if a situation like above mentioned occurs.

For this project, communication between the sensors and the user application is very important. This is achieved by using messaging systems and ReSTful web services.

2.2 Problem Statement

To develop the gateway and provide web services which will facilitate any third-party application to achieve communication with IoT devices that capture Patient movement through a motion detection sensor and process the data for publishing the status of the patient to the healthcare provider(Doctor/Nurse).

CHAPTER 3

MESSAGE ORIENTED MIDDLEWARE

A Message oriented middleware is a middleware that allows different application in an organization to synchronously or asynchronously send and receive messages between them. This allows applications running on different platforms interact and exchange data. Even if there is a new platform coming up and the data transfer can happen with the old applications without any major change to the data model or the applications. The main advantage of using this approach is decoupling of the components or modules i.e. the data processing component will be in a separate application and the data creation will be in a separate application, but as long as the data model is same, changes made on one application doesn't affect the other.

A MOM system consists of two or more clients and a MOM provider. The client will send the message to the MOM provider through API's provided by it. The message will be stored in a queue and afterwards, it will be accessed by the destination client. The messages transferred should be in the form of XML or JSON. There are also certain protocols used like AMQP, DDS, MSMQ to be followed for the exchange of messages. The MOM middleware used here is RabbitMQ which is an open source implementation of AMPQ protocol.

3.1 Advanced Message Queuing Protocol

AMQP (Advanced Message Queueing Protocol) is an openly published wire specification for asynchronous messaging. Every byte of transmitted data is specified. This characteristic allows libraries to be written in many languages, and to run on multiple operating systems and CPU architectures, which makes for a truly interoperable, cross-platform messaging standard.

3.2 RabbitMQ

RabbitMQ is an open source implementation of Message Broker which uses the AMQP protocol for data exchange between publisher and subscriber. In RabbitMQ, a queue can be created and the applications can connect to the queue to exchange messages. The message can be a JSON object describing about a data or an XML, the queue manager stores the message in a queue until the receiver connects to the queue and receives the message.

The architecture of the message queue as shown in Fig. 3.1 is very simple. There is a producer who produces the data, a consumer who consumes the data and the message broker, RabbitMQ in between them who passes the data. The RabbitMQ consists of two components - queues and exchange. The queues are where the messages are stored. As there will be many queues in RabbitMQ, the Exchange is responsible for routing of the messages to the appropriate queues.

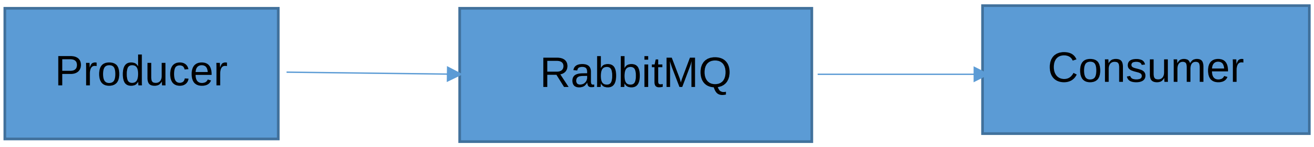


Fig 3.1 Message System Architecture.

3.2.1 AMQP Concepts

RabbitMQ uses AMQP protocol and the key concepts are explained below.

3.2.1.1 Queue

A queue is a buffer that stores messages, for clients to retrieve. A queue is bound to one or more exchanges, optionally with a routing key. A queue may have one or many consumers.

3.2.1.2 Message

A message is what's transported between the publisher and the consumer, it's essentially a byte array with some headers on top.

3.2.1.3 Exchange

Receives messages from producers and pushes them to queues depending on rules defined by the exchange type. In order to receive messages, a queue needs to be bound to at least one exchange.

3.2.1.4 Binding

A binding is a link between a queue and an exchange. A binding is what determinants to which queues a message arriving to an exchange should be routed, it may be to zero or many. How a message is routed depends on which kind of exchange the message was destined for and the "routing key".

3.2.1.5 The Default exchange

The default exchange has no name, but there's a default binding which says that the message will arrive at a queue with the same name as the "routing key".

3.2.1.6 Routing key

As part of the header of every message there's a key called the "routing key" which can be used to route the message.

3.2.2 Exchange types

Exchanges take a message and route it into zero or more queues. The way the message is routed depends on the exchange type and rules called bindings.

3.2.2.1 Direct

A client sends a message to a queue for a particular recipient. The default exchange is a direct exchange with no name (empty string) pre-declared by the broker.

3.2.2.2 Fan-out

A fanout exchange copies the message and routes the message to all queues that are bound to it, the routing key is ignored.

3.2.2.3 Topic

A message is sent to a number of queues based on some rules.

3.2.2.4 Headers

Message headers are inspected and the message is routed based on those headers.

3.3 Workflow in RabbitMQ.

The workflow is illustrated in the fig 3.2 and described below in steps.

Step 1: The producer produces the message and publishes it to the exchange.

Step 2: The exchange then separates the messages and places them in the specific message queues depending on their message attributes.

Step 3: Bindings are created to exchange the messages in queues. The queue will store the message until the consumer subscribes for the message.

Step 4: The consumer will subscribe the queue for the messages.

Step 5: The consumer receives the message.

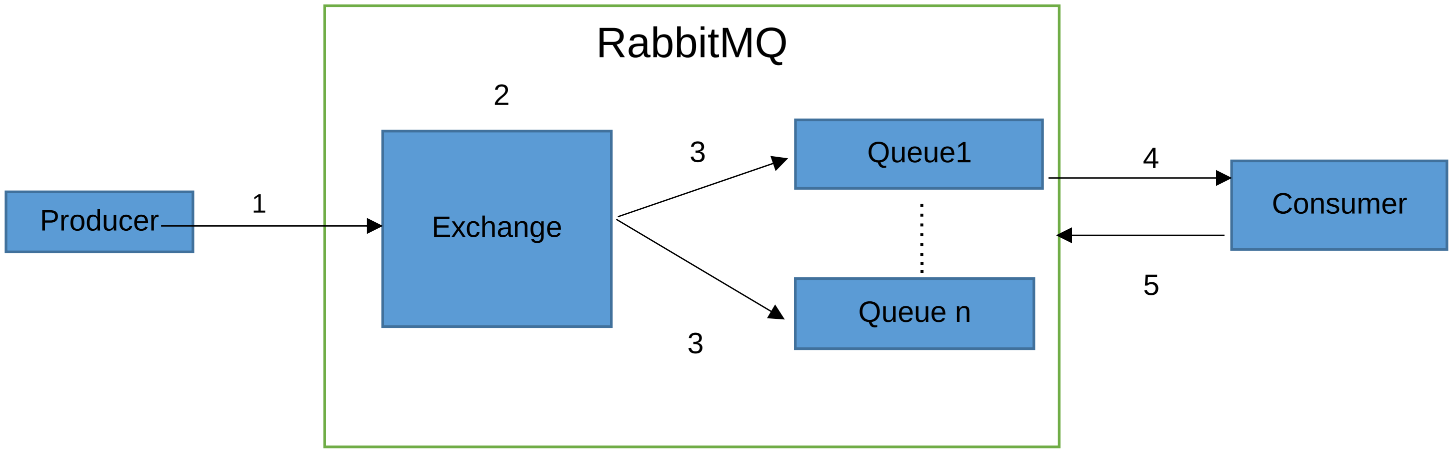
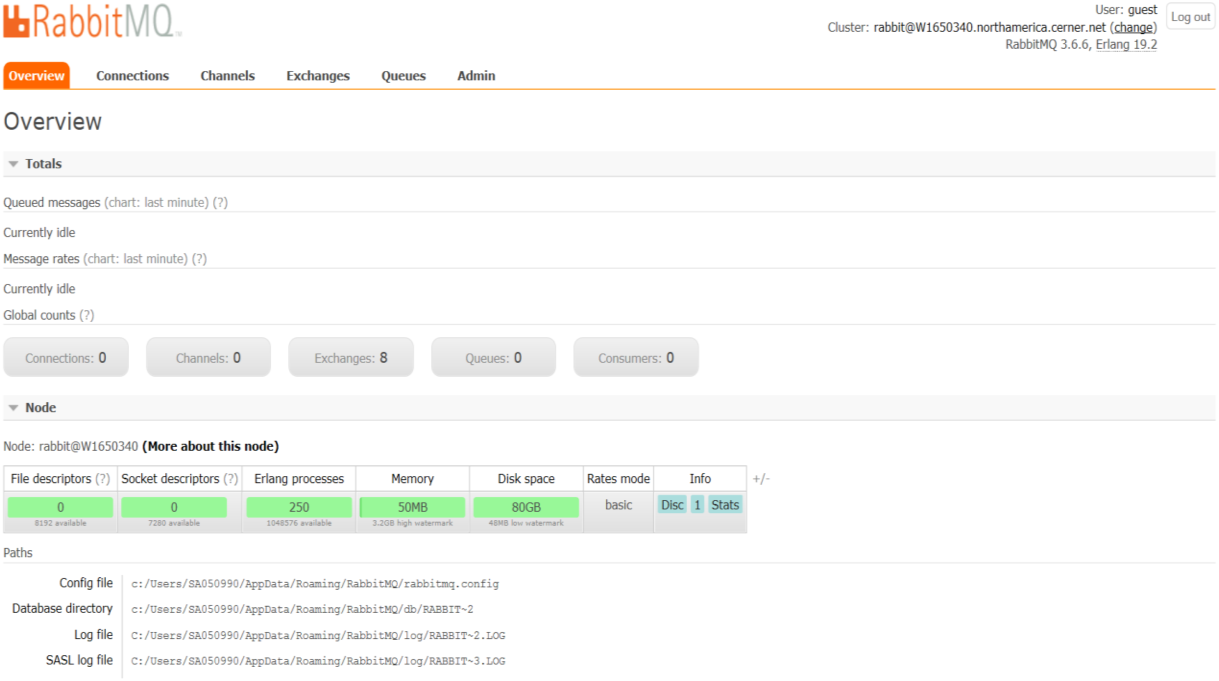


Fig 3.2 RabbitMQ Workflow

A sample program for demonstrating the working of RabbitMQ. The RabbitMQ has a server running on the machine. The homepage of the server looks like Fig 3.3.



User: guest
Cluster: rabbit@W1650340.northamerica.cerner.net (change)
RabbitMQ 3.6.6, Erlang 19.2

Overview

Totals

Queued messages (chart: last minute) (?)

Currently idle

Message rates (chart: last minute) (?)

Currently idle

Global counts (?)

Connections: 0 | Channels: 0 | Exchanges: 8 | Queues: 0 | Consumers: 0

Node

Node: rabbit@W1650340 **(More about this node)**

File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk space	Rates mode	Info
0 8192 available	0 7280 available	250 1048576 available	50MB 3.2GB high watermark	80GB 48MB low watermark	basic	Disc 1 Stats

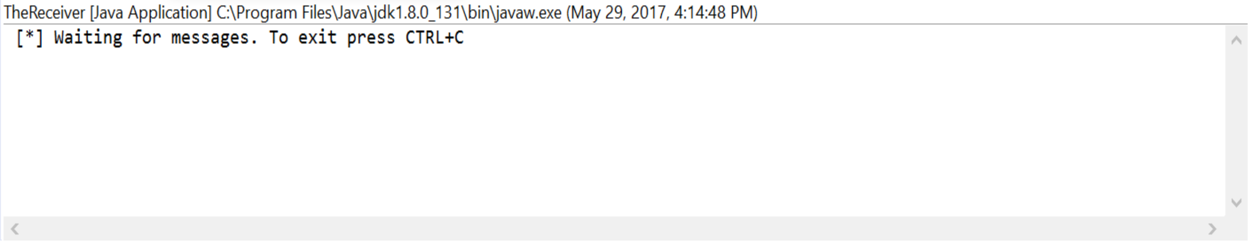
Paths

Config file	c:/Users/SA050990/AppData/Roaming/RabbitMQ/rabbitmq.config
Database directory	c:/Users/SA050990/AppData/Roaming/RabbitMQ/db/RABBIT~2
Log file	C:/Users/SA050990/AppData/Roaming/RabbitMQ/log/RABBIT~2.LOG
SASL log file	C:/Users/SA050990/AppData/Roaming/RabbitMQ/log/RABBIT~3.LOG

Fig 3.3

If the above page is visible, you can be sure that the RabbitMQ sever is running.

Next in eclipse, when the consumer program is run, the receiver will wait for any message from the server.

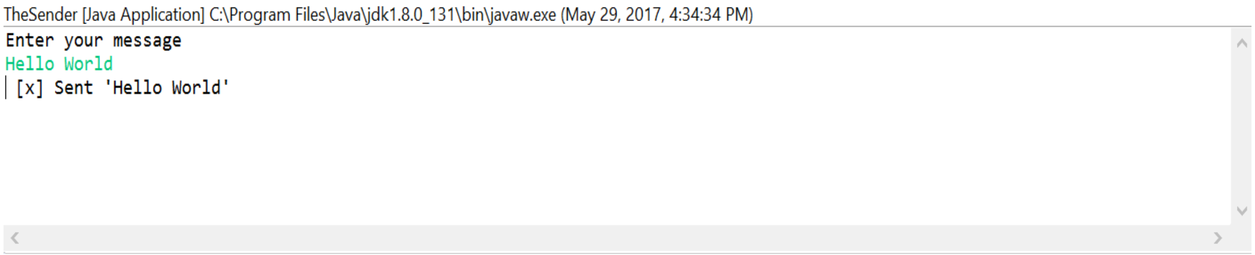


TheReceiver [Java Application] C:\Program Files\Java\jdk1.8.0_131\bin\javaw.exe (May 29, 2017, 4:14:48 PM)

```
 [*] Waiting for messages. To exit press CTRL+C
```

Fig 3.4

In the next step, we will start the producer program. This program will get an input from the user and publishes it to the server.

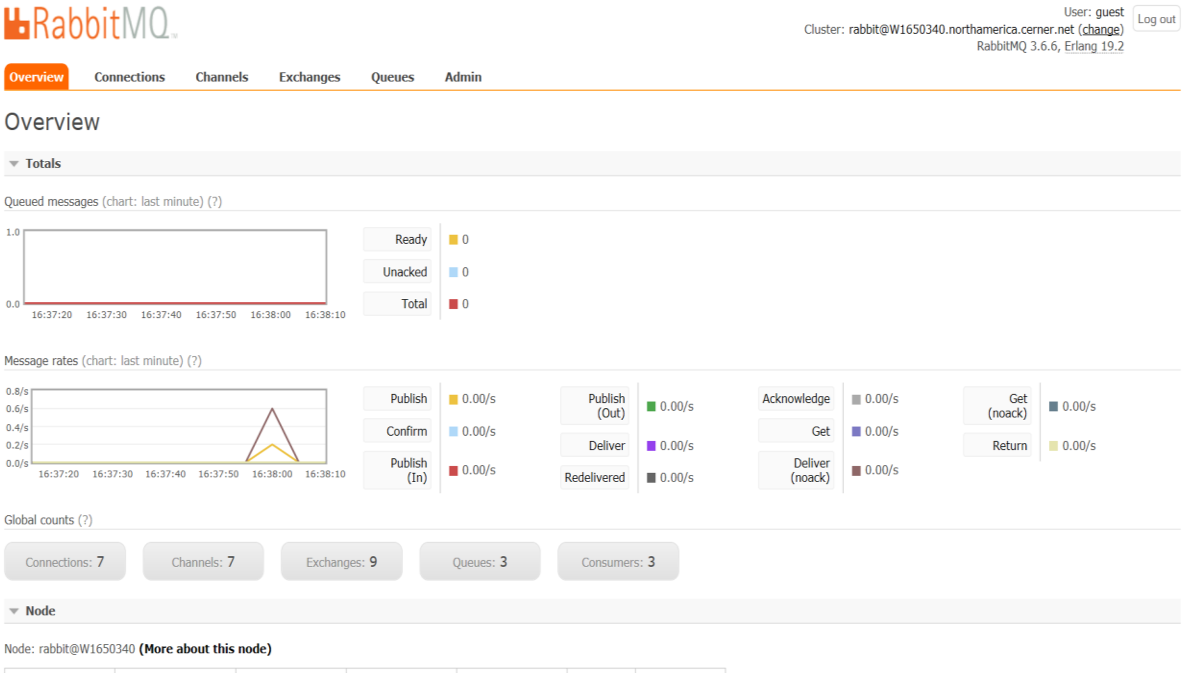


TheSender [Java Application] C:\Program Files\Java\jdk1.8.0_131\bin\javaw.exe (May 29, 2017, 4:34:34 PM)

```
Enter your message
Hello World
 [x] Sent 'Hello World'
```

Fig 3.5

The message has been send to the receiver and the received messages rate can be seen in the RabbitMQ management site.



User: guest
Cluster: rabbit@W1650340.northamerica.cerner.net (change)
RabbitMQ 3.6.6, Erlang 19.2

Overview

Totals

Queued messages (chart: last minute) (?)

Ready	0
Unacked	0
Total	0

Message rates (chart: last minute) (?)

Publish	0.00/s	Publish (Out)	0.00/s	Acknowledge	0.00/s	Get (noack)	0.00/s
Confirm	0.00/s	Deliver	0.00/s	Get	0.00/s	Return	0.00/s
Publish (In)	0.00/s	Redelivered	0.00/s	Deliver (noack)	0.00/s		

Global counts (?)

Connections: 7 Channels: 7 Exchanges: 9 Queues: 3 Consumers: 3

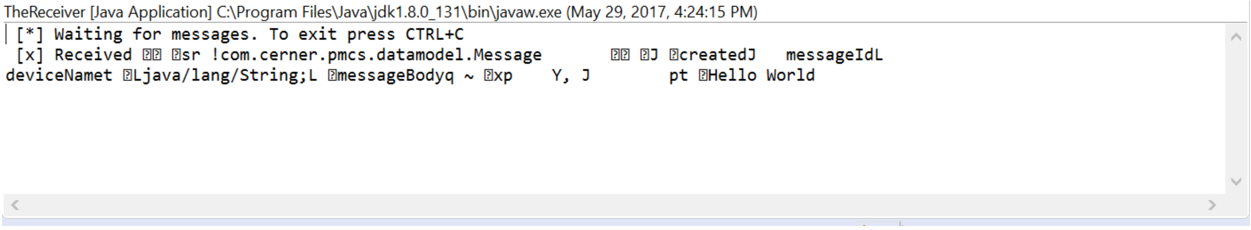
Node

Node: rabbit@W1650340 **(More about this node)**

Fig 3.6

The spike in the second graph indicates the transfer of messages and if there are any messages to be sent it will be displayed in the queued messages graph.

The consumer program gets the message and displays it in the output as seen in fig 3.7



TheReceiver [Java Application] C:\Program Files\Java\jdk1.8.0_131\bin\javaw.exe (May 29, 2017, 4:24:15 PM)

```
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received ⍰⍰ ⍰sr !com.cerner.pmcs.datamodel.Message       ⍰⍰ ⍰J ⍰createdJ   messageIdL
deviceNamet ⍰Ljava/lang/String;L ⍰messageBodyq ~ ⍰xp    Y, J        pt ⍰Hello World
```

Fig 3.7

The RabbitMQ is used in this project for the transfer of messages between the IoT device and the gateway. The sensor information will be processed into useful information and sent to the RabbitMQ message broker. This information will be stored in the queue and can be accessed by the user application using web services.

CHAPTER 4

WEB SERVICES

Web services is the service provided by a machine to another machine through World Wide Web(WWW). We have a wide collection of protocols and standards used. Applications from different platforms can communicate between each other. Hence it is platform independent. It differs from Messaging systems in a way like we can use all the basic HTTP calls and work on them and not have to access them with API.

There are mainly three web service components

1. SOAP
2. WSDL
3. UDDI

SOAP

SOAP stands for Simple Object Access Protocol. It is a XML Based protocol for accessing web services.

WSDL

WSDL is an acronym for Web Services Description Language. It is an XML document which contains information about the web service such as method name, parameters and usage of it. It is the interface between the web service and the application.

UDDI

UDDI is an acronym for Universal Description Discovery and Integration. It is also a XML based framework which describes, discovers and integrates web services.

There are two types of web services, SOAP and ReSTful. We are using ReSTful in our system.

4.1 ReSTful Web Services.

ReSTful stands for Representational State Transfer. ReSTful is an architectural design for Web services used for transferring resources using HTTP to a wide range of clients in different platforms. The ReST has taken over the other designs like the SOAP and WSDL. ReSTful is fast and doesn’t have strict specification like SOAP and they can be used in any programming language. It supports different data formats like plain test, JSON, XML. A web service is just a web page meant for a computer to request and process.

More precisely, a Web service is a Web page that’s meant to be consumed by an autonomous program as opposed to a Web browser or similar UI tool Web Services require an architectural style to make sense of them, because there’s no smart human being on the client end to keep track. The pre-Web techniques of computer interaction don't scale on the Internet. They were designed for small scales and single trust domains.

ReST uses HTTP protocols are used to transfer of data. HTTP stands for Hyper Text Transfer Protocol. It is an application layer protocol which gets request from clients, contacts the server, fetches the data from the server and replies to the client with the resource. There are several HTTP methods through which the client can communicate with the server. The methods are listed in the table Table 4.1 below.

Method	**Usage**
GET	Fetch all or any one resource
POST	Create a resource
PUT	Update a resource
DELETE	Delete a resource

Table 4.1

The ReST services are stateless because the server doesn't store the state of the clients. Each client must pass its state each time it needs to communicate with the server. This makes it possible for the server to process all the requests independently from each client. But because of this the client must pass on more information to the server every time, which may increase the latency.

In this project, the restful services have been used for managing device information and sending messages. Every time a new sensor is attached, it needs to be registered into the server and it can be modified or deleted. We can also get a list of all the devices registered. The services also handle sending and receiving messages.

There are two RESTful Web Services that are implemented.

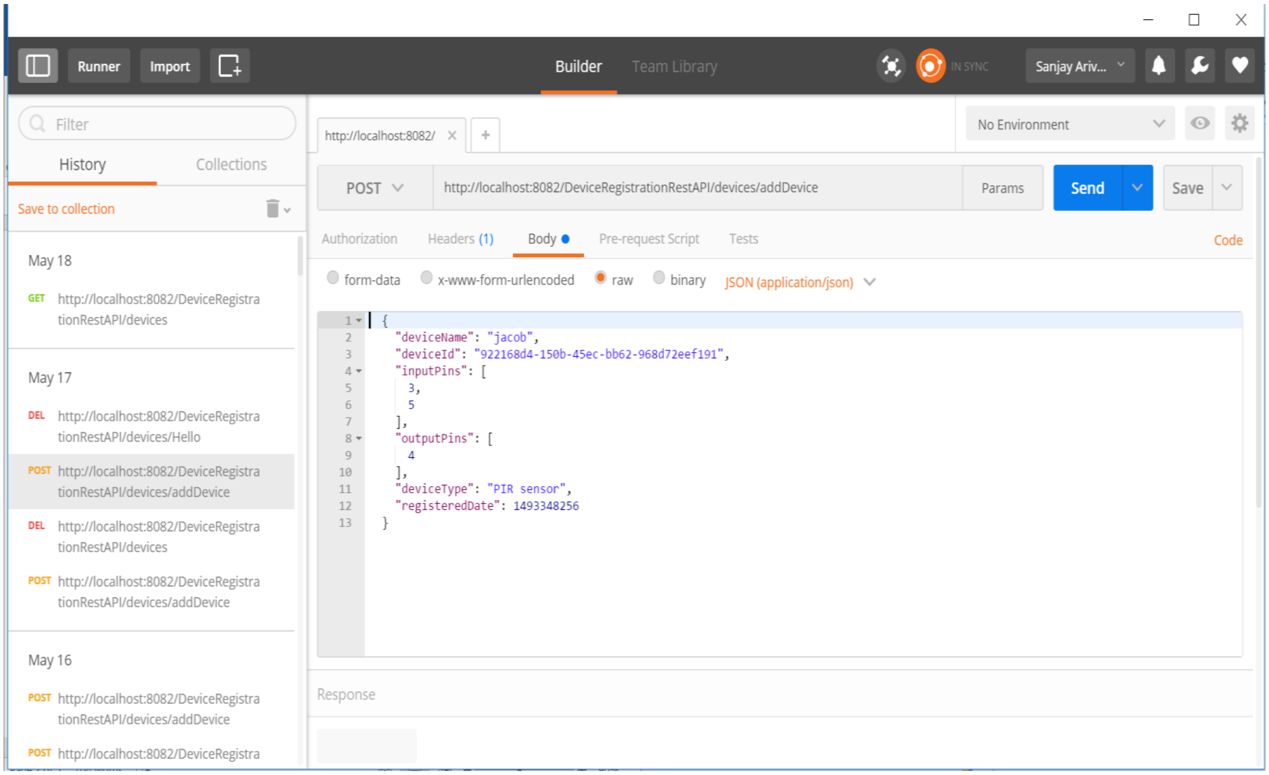
1. Device Registration
2. Messenger

The web services have been deployed in a maven project and it has a java class that has the methods implemented methods to handle HTTP requests from the other applications. Once the war file for the project has been deployed into the server. The following HTTP methods have been handled,

1. POST
2. GET
3. DELETE
4. PUT

4.1.1 POST

The post method is used to register the sensors. The parameters specifying the device configuration will be send in the body of the POST request like in Fig 4.1.



May 18

GET http://localhost:8082/DeviceRegistrationRestAPI/devices

May 17

DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/Hello

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

DEL http://localhost:8082/DeviceRegistrationRestAPI/devices

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

May 16

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

POST http://localhost:8082/DeviceRegistra

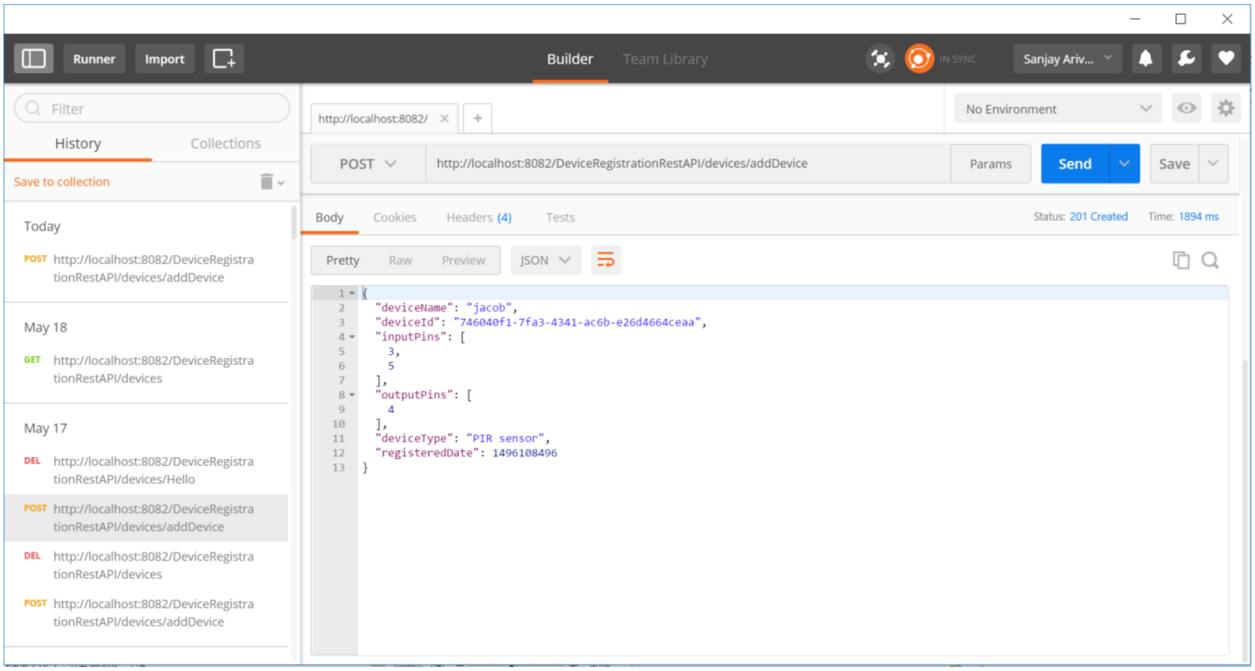
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

JSON (application/json)

```
{
  "deviceName": "jacob",
  "deviceId": "922168d4-150b-45ec-bb62-968d72eef191",
  "inputPins": [
    3,
    5
  ],
  "outputPins": [
    4
  ],
  "deviceType": "PIR sensor",
  "registeredDate": 1493348256
}
```

Fig 4.1

When the device is successfully registered, the server returns a 201(created) HTTP status code as shown in fig 4.2.



Today

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

May 18

GET http://localhost:8082/DeviceRegistrationRestAPI/devices

May 17

DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/Hello

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

DEL http://localhost:8082/DeviceRegistrationRestAPI/devices

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

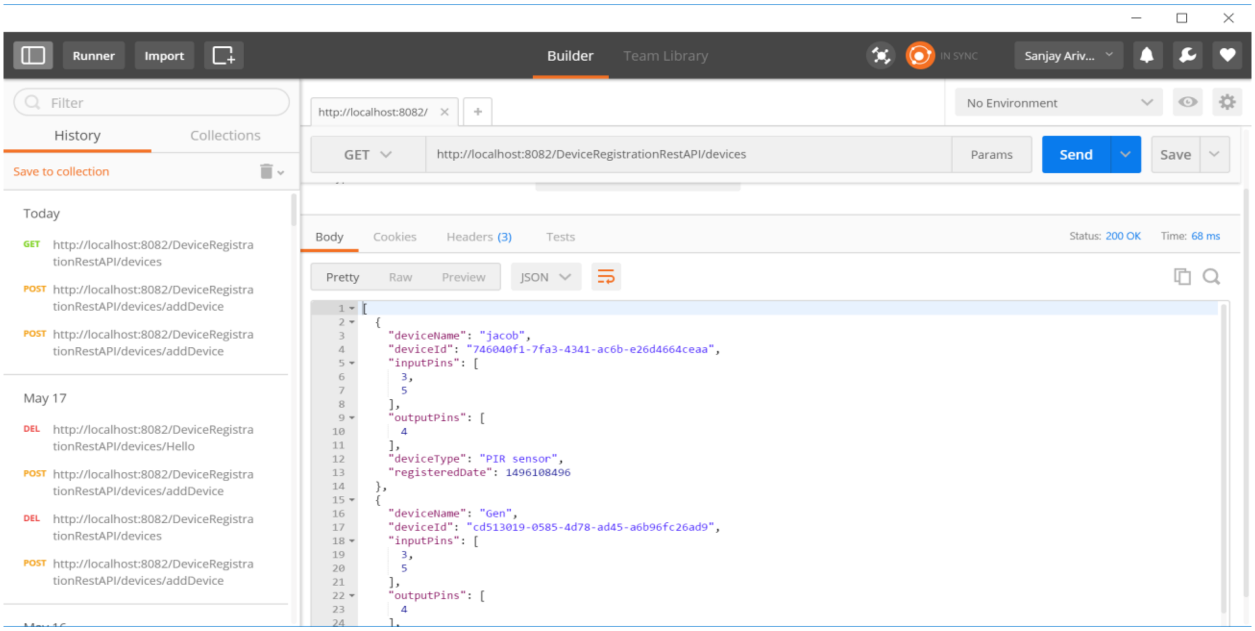
Status: 201 Created Time: 1894 ms

```
{
  "deviceName": "jacob",
  "deviceId": "746040f1-7fa3-4341-ac6b-e26d4664ceaa",
  "inputPins": [
    3,
    5
  ],
  "outputPins": [
    4
  ],
  "deviceType": "PIR sensor",
  "registeredDate": 1496108496
}
```

Fig 4.2.

4.1.2 GET

The get method is used to receive all the devices’ information. The fig 4.3 displays all devices.



Today

GET http://localhost:8082/DeviceRegistrationRestAPI/devices

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

May 17

DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/Hello

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

DEL http://localhost:8082/DeviceRegistrationRestAPI/devices

POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

GET http://localhost:8082/DeviceRegistrationRestAPI/devices

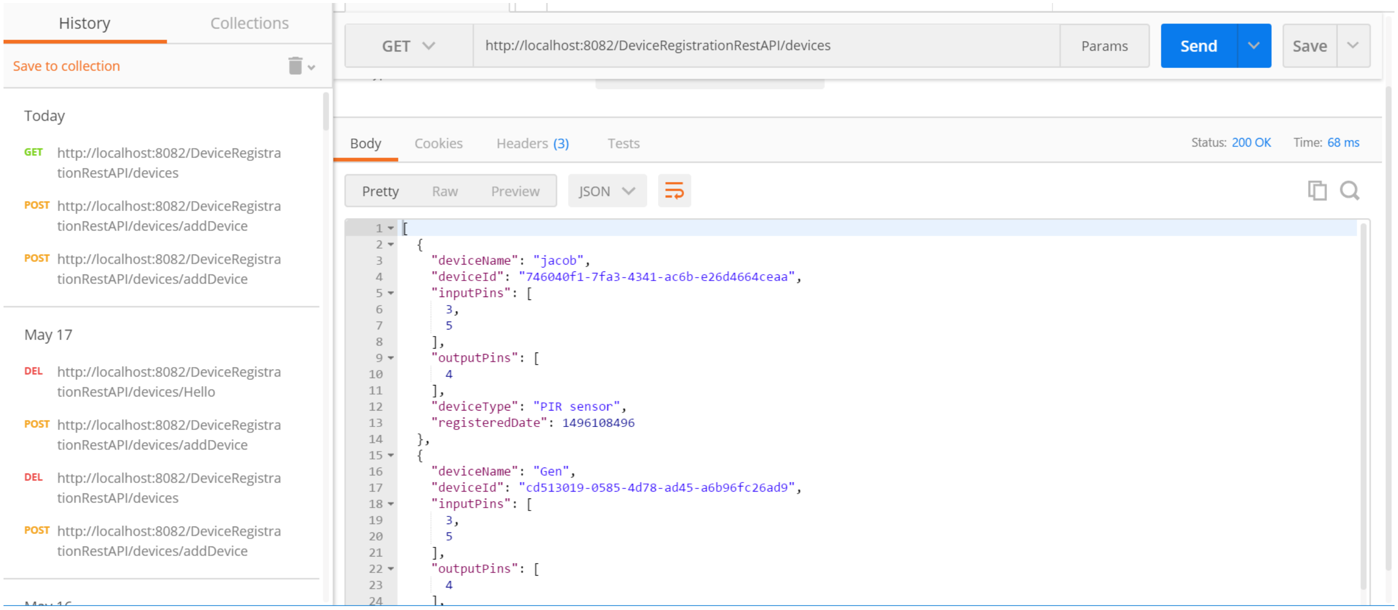
Status: 200 OK Time: 68 ms

```
[
  {
    "deviceName": "jacob",
    "deviceId": "746040f1-7fa3-4341-ac6b-e26d4664ceaa",
    "inputPins": [
      3,
      5
    ],
    "outputPins": [
      4
    ],
    "deviceType": "PIR sensor",
    "registeredDate": 1496108496
  },
  {
    "deviceName": "Gen",
    "deviceId": "cd513019-0585-4d78-ad45-a6b96fc26ad9",
    "inputPins": [
      3,
      5
    ],
    "outputPins": [
      4
    ],
```

Fig 4.3

4.1.3 DELETE

The DELETE method is either used for deleting a single device or all devices. The Devices list before deleting the device “Jacob” is shown in Fig 4.4.



GET http://localhost:8082/DeviceRegistrationRestAPI/devices

Status: 200 OK Time: 68 ms

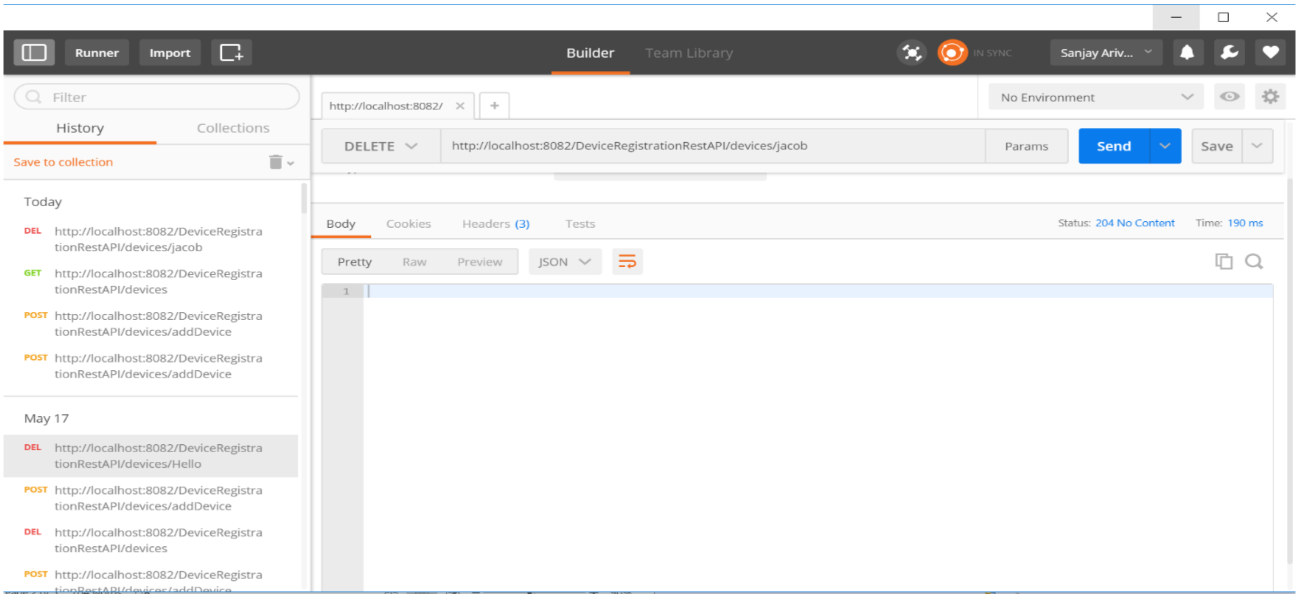
```
[
  {
    "deviceName": "jacob",
    "deviceId": "746040f1-7fa3-4341-ac6b-e26d4664ceaa",
    "inputPins": [
      3,
      5
    ],
    "outputPins": [
      4
    ],
    "deviceType": "PIR sensor",
    "registeredDate": 1496108496
  },
  {
    "deviceName": "Gen",
    "deviceId": "cd513019-0585-4d78-ad45-a6b96fc26ad9",
    "inputPins": [
      3,
      5
    ],
    "outputPins": [
      4
    ],
```

History
Collections
Save to collection
Today
GET http://localhost:8082/DeviceRegistrationRestAPI/devices
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
May 17
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/Hello
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
Params
Send
Save
Body
Cookies
Headers (3)
Tests
Pretty
Raw
Preview
JSON

Fig 4.4

The sensor is deleted with the URL http://localhost:8082/DeviceRegistrationRestAPI/devices/jacob.Now the sensor "Jacob" is deleted which is shown in fig 4.5.

The Rest APIs provide option to delete a single device or delete all devices at once.



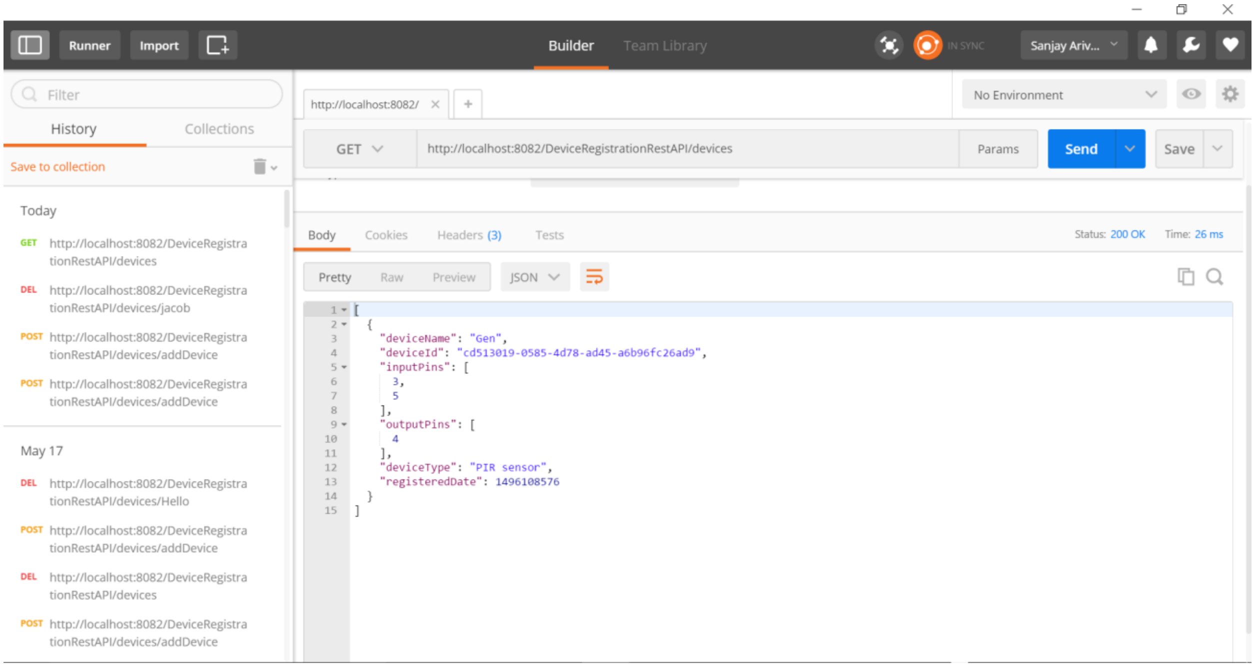
DELETE http://localhost:8082/DeviceRegistrationRestAPI/devices/jacob

Status: 204 No Content Time: 190 ms

Runner
Import
Builder
Team Library
IN SYNC
Sanjay Ariv...
Filter
History
Collections
Save to collection
Today
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/jacob
GET http://localhost:8082/DeviceRegistrationRestAPI/devices
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
May 17
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/Hello
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
http://localhost:8082/
No Environment
Params
Send
Save
Body
Cookies
Headers (3)
Tests
Pretty
Raw
Preview
JSON

Fig 4.5.

The below figure fig 4.6 confirms that the sensor "jacob" has been deleted.



Today

GET http://localhost:8082/DeviceRegistrationRestAPI/devices
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/jacob
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

May 17

DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/Hello
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

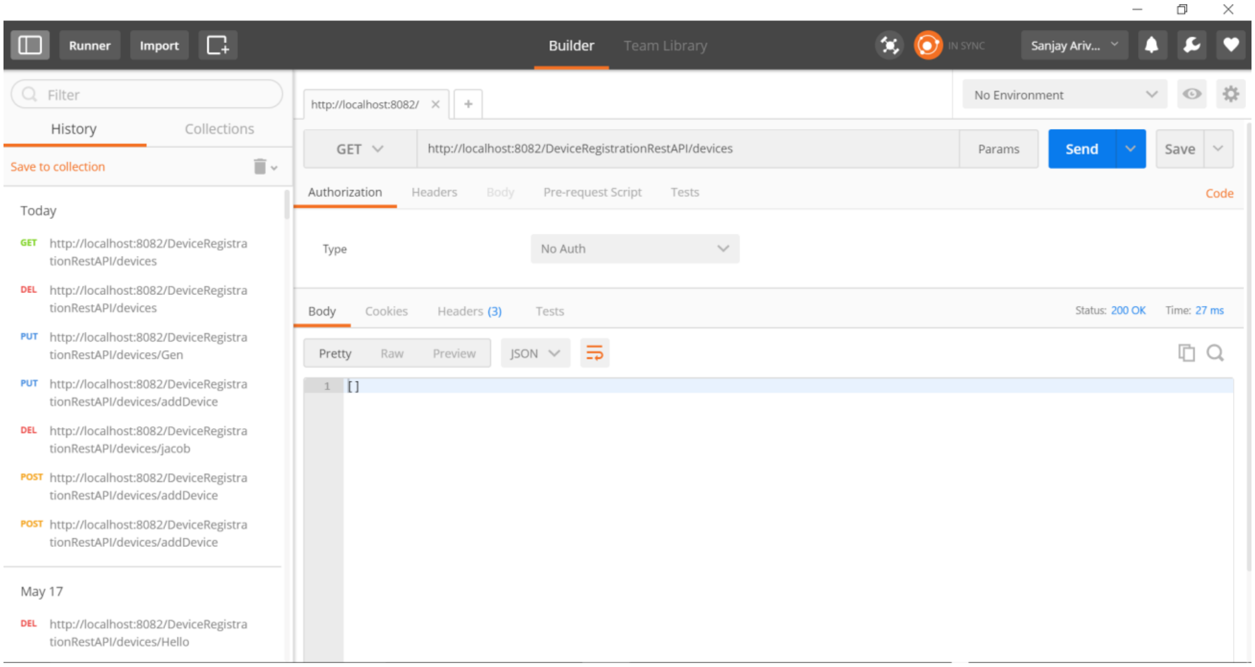
GET http://localhost:8082/DeviceRegistrationRestAPI/devices

Status: 200 OK Time: 26 ms

```
[
  {
    "deviceName": "Gen",
    "deviceId": "cd513019-0585-4d78-ad45-a6b96fc26ad9",
    "inputPins": [
      3,
      5
    ],
    "outputPins": [
      4
    ],
    "deviceType": "PIR sensor",
    "registeredDate": 1496108576
  }
]
```

Fig 4.6

All the devices can be deleted all at once with the same delete but with a different URL, http://localhost:8082/DeviceRegistrationRestAPI/devices as shown in Fig 4.7.



Today

GET http://localhost:8082/DeviceRegistrationRestAPI/devices
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices
PUT http://localhost:8082/DeviceRegistrationRestAPI/devices/Gen
PUT http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/jacob
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice

May 17

DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/Hello

GET http://localhost:8082/DeviceRegistrationRestAPI/devices

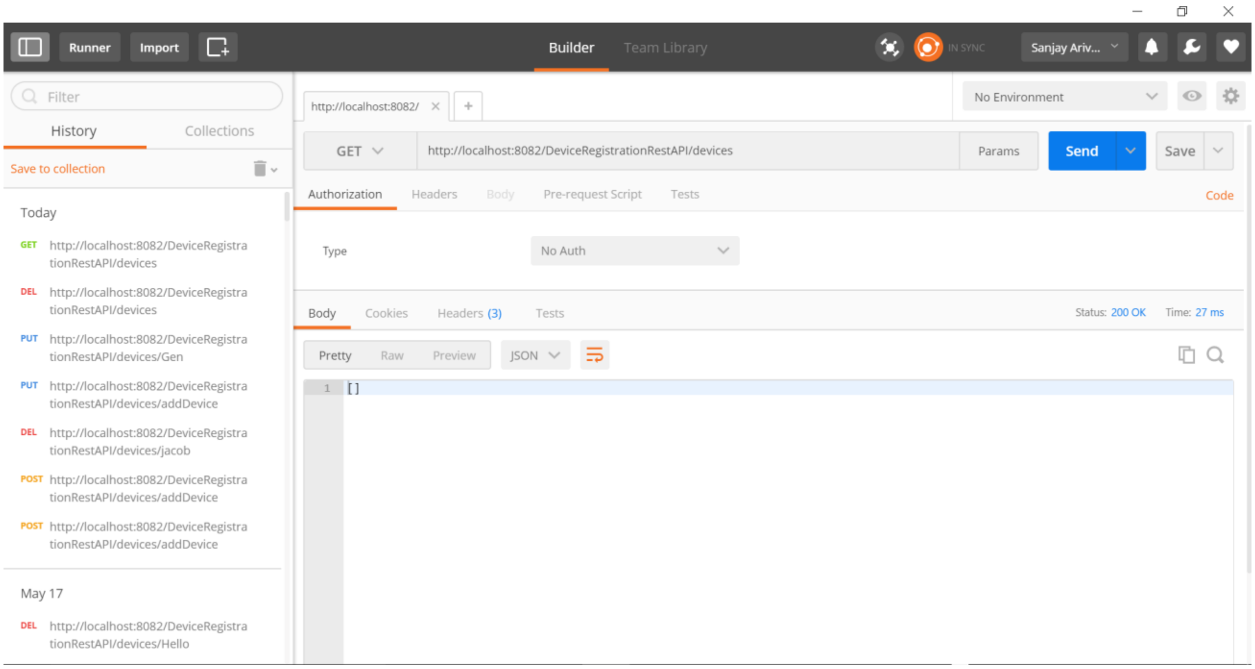
Type: No Auth

Status: 200 OK Time: 27 ms

```
[]
```

Fig 4.7

The previous DELETE method has deleted all the sensors. As there are no sensors left in the server, a GET method will now return an empty square brackets as in Fig.4.8.


Runner
Import
Builder
Team Library
IN SYNC
Sanjay Ariv...
Filter
History
Collections
Save to collection
Today
GET http://localhost:8082/DeviceRegistrationRestAPI/devices
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices
PUT http://localhost:8082/DeviceRegistrationRestAPI/devices/Gen
PUT http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/jacob
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
POST http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
May 17
DEL http://localhost:8082/DeviceRegistrationRestAPI/devices/Hello
http://localhost:8082/
No Environment
Params
Send
Save
Authorization
Headers
Body
Pre-request Script
Tests
Code
Body
Cookies
Headers (3)
Tests
Pretty
Raw
Preview
JSON

GET http://localhost:8082/DeviceRegistrationRestAPI/devices

Type: No Auth

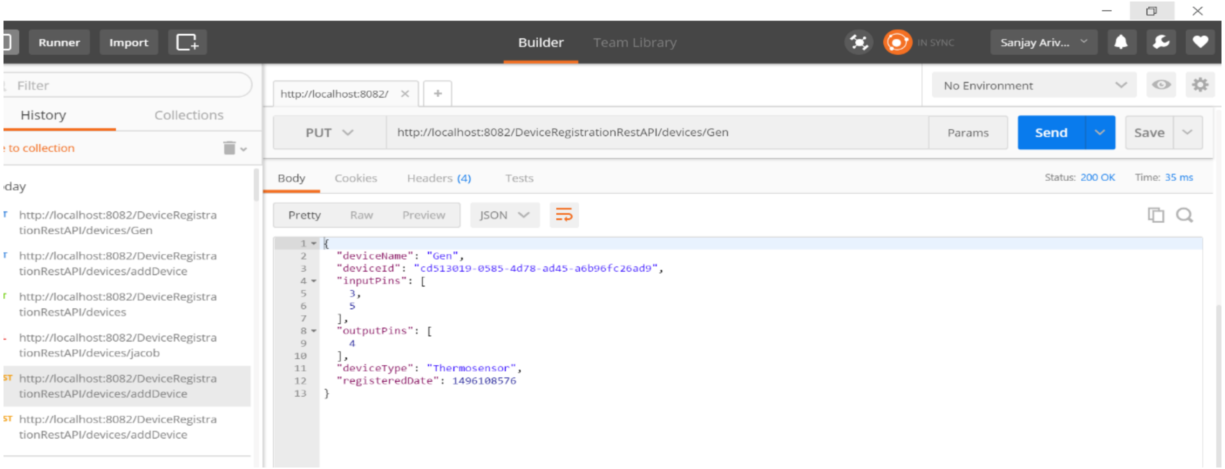
Status: 200 OK Time: 27 ms

```
[]
```

Fig 4.8

4.1.4 PUT

The put method is used for updating the previously existing sensor. The device information that needs to be updated will be in the body of the request as in fig 4.9.


Runner
Import
Builder
Team Library
IN SYNC
Sanjay Ariv...
Filter
History
Collections
to collection
day
http://localhost:8082/DeviceRegistrationRestAPI/devices/Gen
http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
http://localhost:8082/DeviceRegistrationRestAPI/devices
http://localhost:8082/DeviceRegistrationRestAPI/devices/jacob
http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
http://localhost:8082/DeviceRegistrationRestAPI/devices/addDevice
http://localhost:8082/
No Environment
Params
Send
Save
Body
Cookies
Headers (4)
Tests
Pretty
Raw
Preview
JSON

PUT http://localhost:8082/DeviceRegistrationRestAPI/devices/Gen

Status: 200 OK Time: 35 ms

```
{
  "deviceName": "Gen",
  "deviceId": "cd513019-0585-4d78-ad45-a6b96fc26ad9",
  "inputPins": [
    3,
    5
  ],
  "outputPins": [
    4
  ],
  "deviceType": "Thermosensor",
  "registeredDate": 1496108576
}
```

Fig 4.9

CHAPTER 5

DATA MODEL

5.1 Device class

The data model generalizes the sensors as devices. This way we can add any type of devices like actuators in the future. The device is modelled as shown in the fig 5.1.

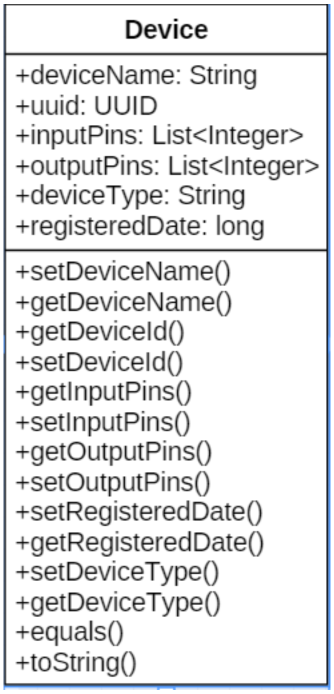


Fig 5.1

The Device class consists of name, Id, type, date, input pins and output pins. Inputs pins and output pins specify the pin that is connected to the IoT device. This way, the IoT device can be configured so that it can receive the input and send output accordingly. Builder pattern is used for creating the device object.

5.2 Message Class

The Message class defines the model of how the message needs to be composed. The Device Name is associated with the message. It acts like a foreign key and we know from which device the message is from.

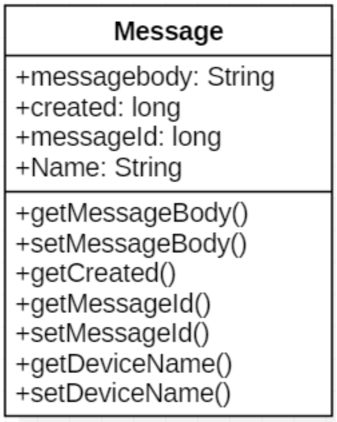


Fig 5.2

The Message class consists of message Id, message body, created date and Device Name.

5.3 RabbitMQ classes

The below classes were used to create connections and channels that allows to communicate using RabbitMQ.

1. Connection Factory
2. Connection
3. Channel Factory
4. Channel

The design pattern used here was factory pattern which allowed to create object based on the child class.

CHAPTER 6

SYSTEM DESIGN

The web services can be deployed in any platform and in any language. A web application and an android application have been created to test it.

6.1 ARCHITECTURE

The system architecture consists of the sensors, a raspberry pi (or any IoT device), the message gateway and user application.

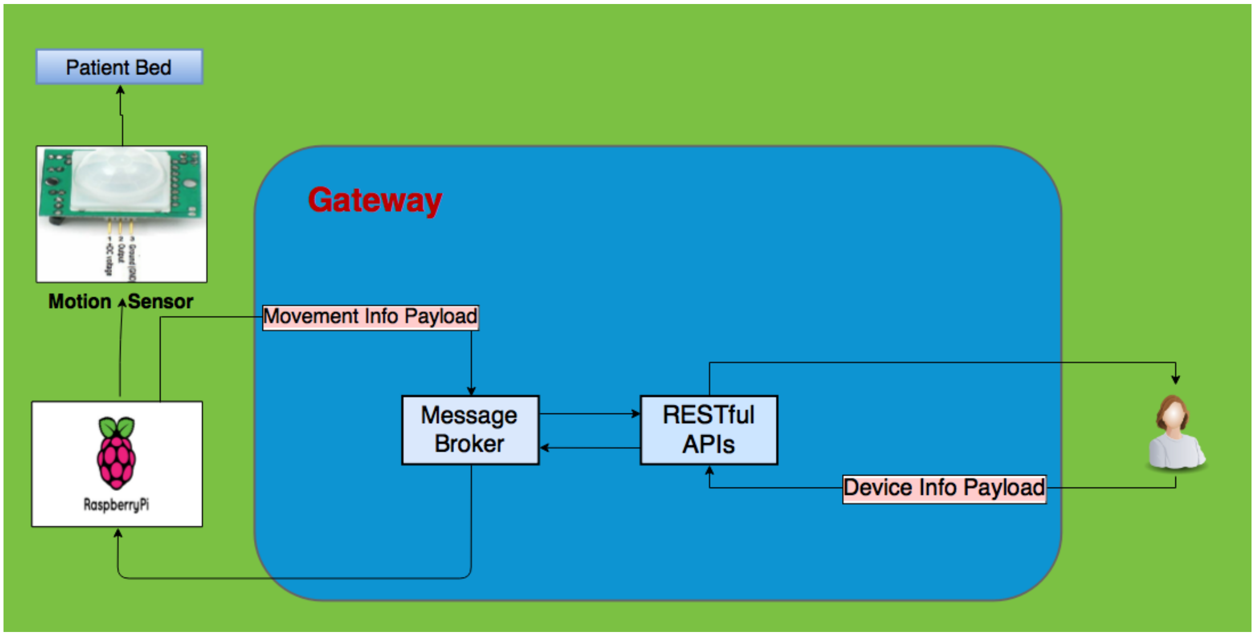
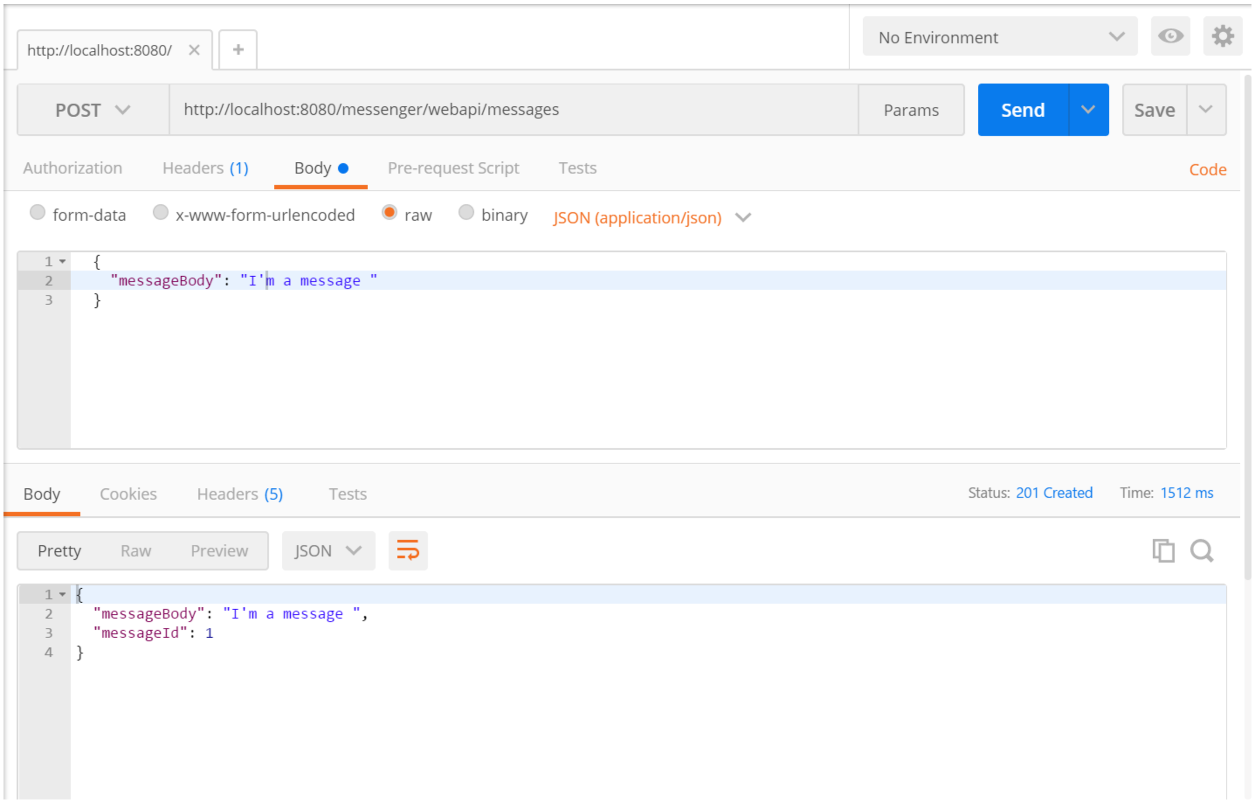


Fig 6.1

The movement of the patient is captured in the sensor. The raspberry pi will get the raw information from the sensors and then it is processed to collect useful information. Then the data is send to the Message broker, which stores it in a queue. The user application can access the information through ReSTful API's.

The messages are send using the POST method in the web services. A sample exchange is shown in the next steps.



POST http://localhost:8080/messenger/webapi/messages

● raw JSON (application/json)

```
{
  "messageBody": "I'm a message "
}
```

Body JSON

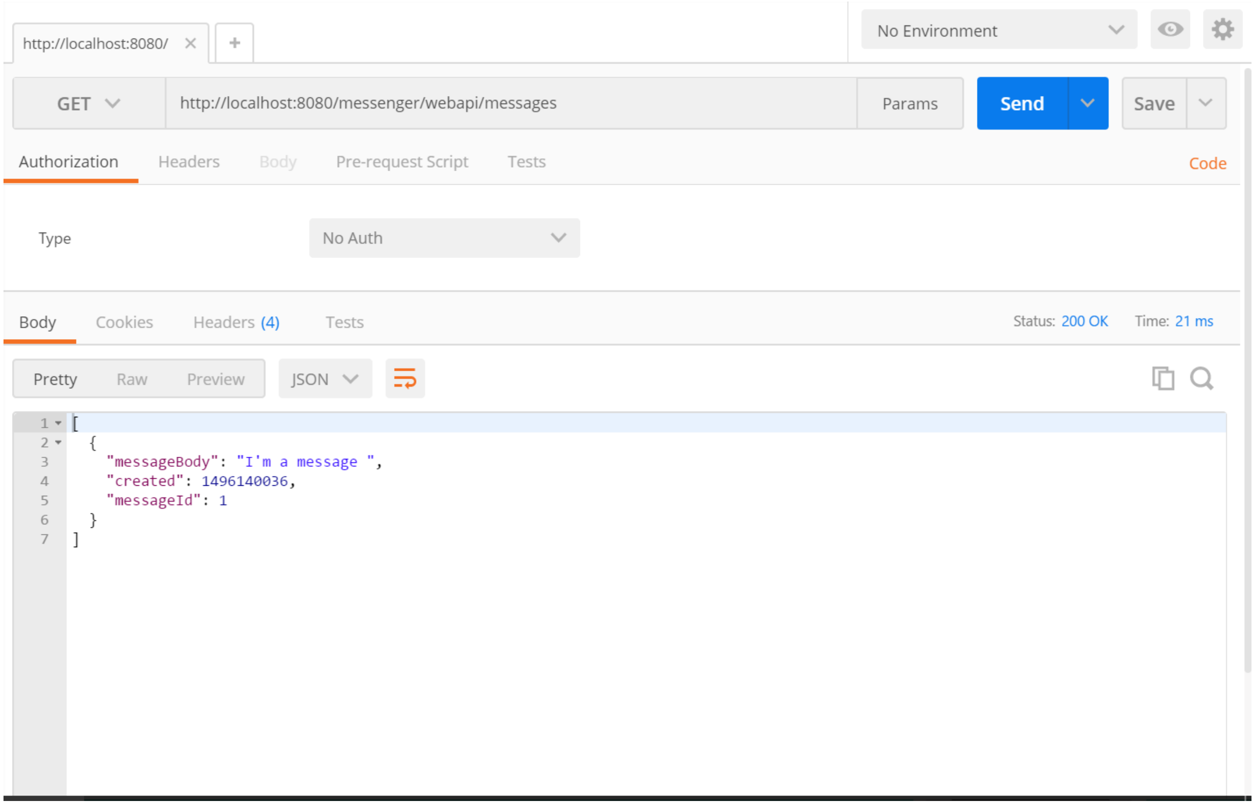
Status: 201 Created Time: 1512 ms

```
{
  "messageBody": "I'm a message ",
  "messageId": 1
}
```

Fig 6.2

The post method body contains the message that needs to be send as shown in fig 6.2.

The message is received and viewed using the GET method as shown in fig 6.3.



GET http://localhost:8080/messenger/webapi/messages

Type: No Auth

Status: 200 OK Time: 21 ms

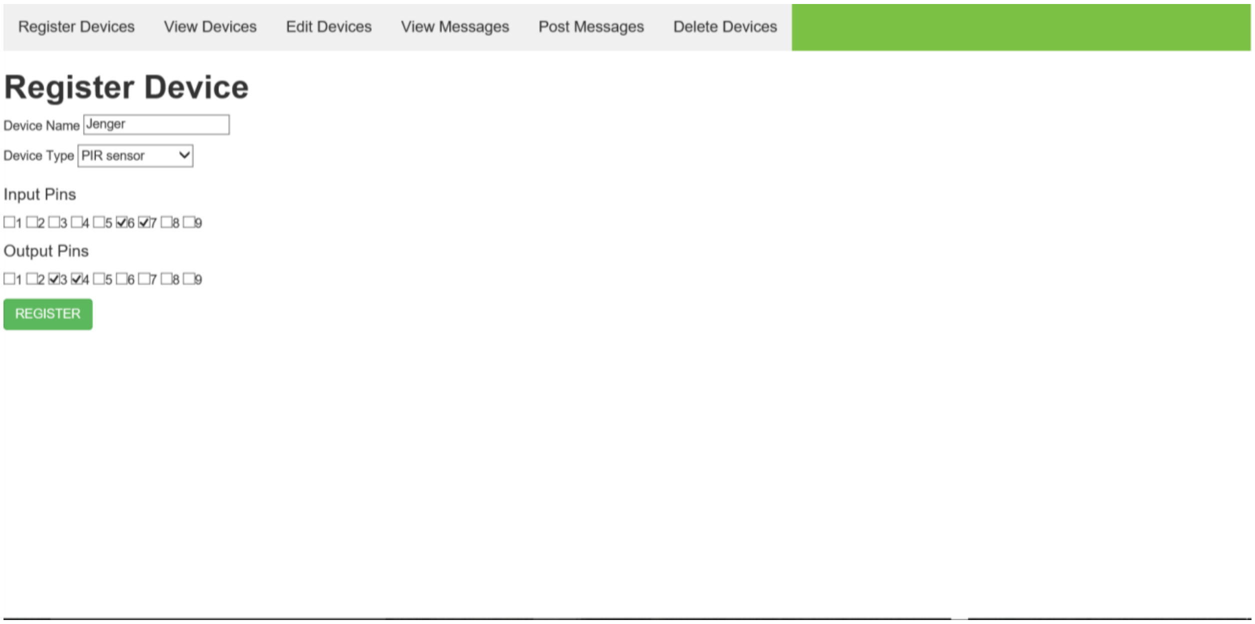
```
[
  {
    "messageBody": "I'm a message ",
    "created": 1496140036,
    "messageId": 1
  }
]
```

Fig 6.3

6.2 WEB APPLICATION

The web application demonstrates the basic CRUD functionalities and the device registrations along with sending messages with a device name and viewing it in the page. The following pages are developed in application.

6.2.1 Registration Front End



Register Device

Device Name Jenger

Device Type PIR sensor

Input Pins

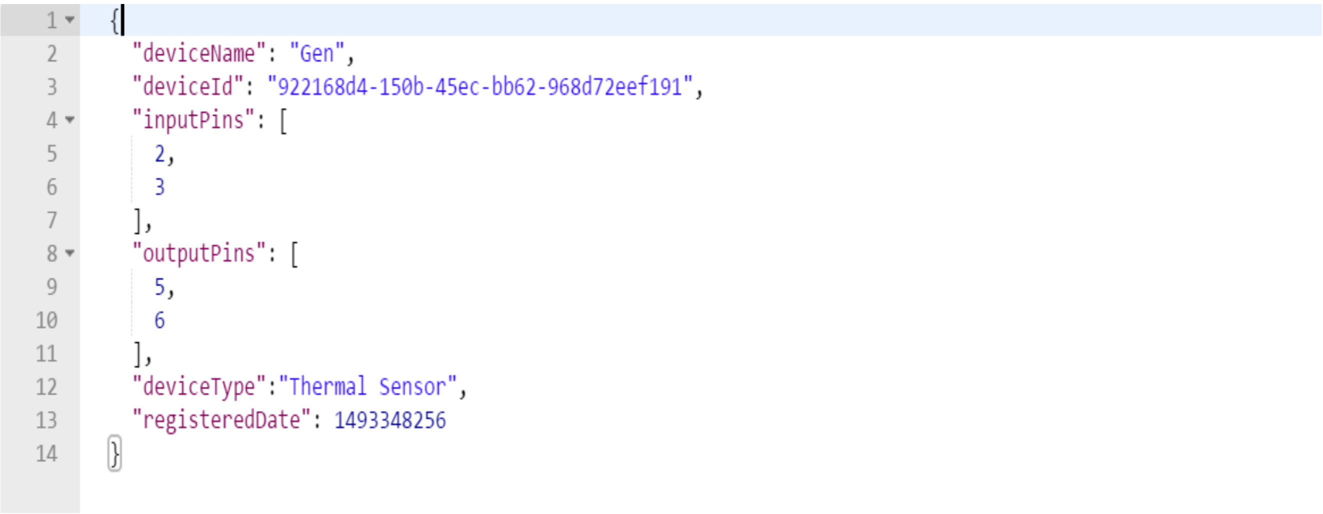
□1 □2 □3 □4 □5 ☑6 ☑7 □8 □9

Output Pins

□1 □2 ☑3 ☑4 □5 □6 □7 □8 □9

Fig 6.4

The device name can be added to the edit box and the device type can be selected from the given selection. The sensors will have input/output pins. Specific pins will be used for getting input and output. The pins that are connected are selected here and then the register button is selected. The information is converted to a JSON object like below,



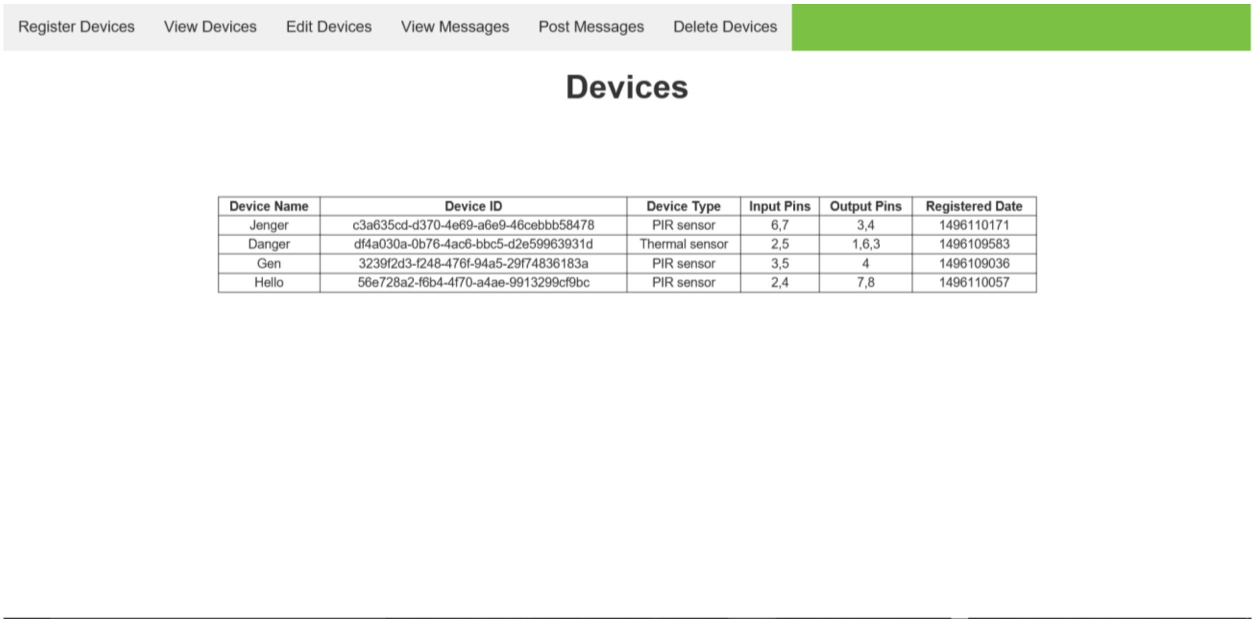
```
{
  "deviceName": "Gen",
  "deviceId": "922168d4-150b-45ec-bb62-968d72eef191",
  "inputPins": [
    2,
    3
  ],
  "outputPins": [
    5,
    6
  ],
  "deviceType":"Thermal Sensor",
  "registeredDate": 1493348256
}
```

Fig 6.5

Then they are send to the server using ajax send method. After it is processed and stored in a local system.

6.2.2 Device list Front End

The device information is got from the server while the page is loaded and the device information is got as an array of JSON objects from the server. Then it is loaded in to a table in the page.



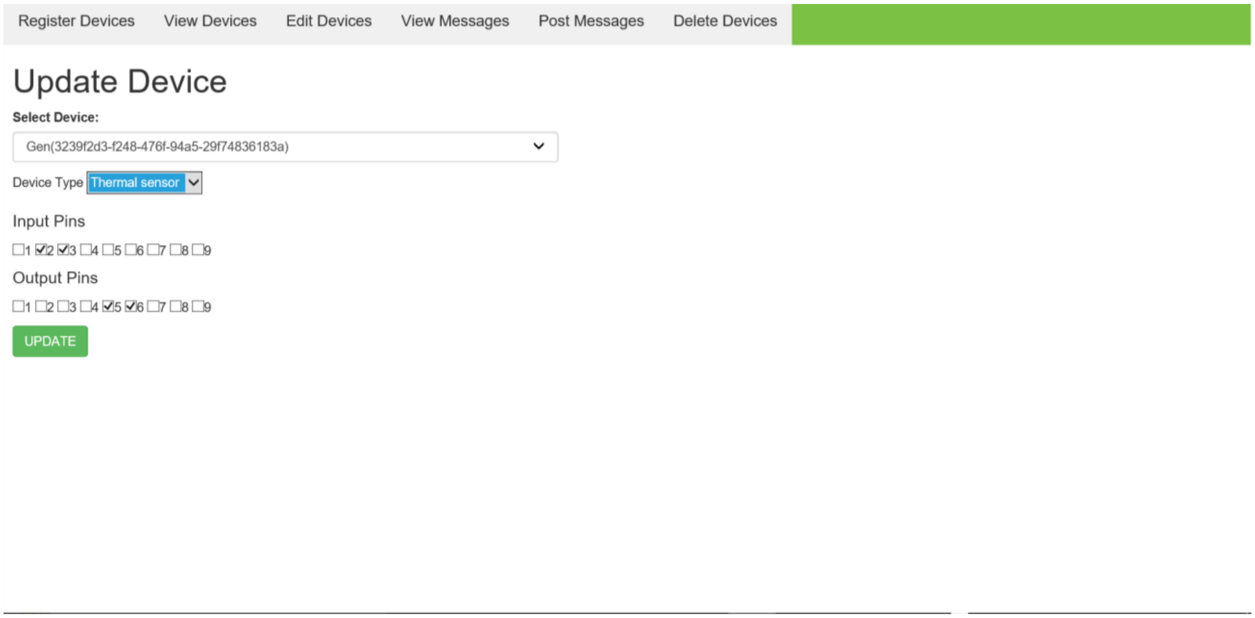
Devices

Device Name	Device ID	Device Type	Input Pins	Output Pins	Registered Date
Jenger	c3a635cd-d370-4e69-a6e9-46cebbb58478	PIR sensor	6,7	3,4	1496110171
Danger	df4a030a-0b76-4ac6-bbc5-d2e59963931d	Thermal sensor	2,5	1,6,3	1496109583
Gen	3239f2d3-f248-476f-94a5-29f74836183a	PIR sensor	3,5	4	1496109036
Hello	56e728a2-f6b4-4f70-a4ae-9913299cf9bc	PIR sensor	2,4	7,8	1496110057

Fig 6.6

6.2.3 Device Update Front End

The device to be updated will be selected and then the changes can be made. When the update button is clicked, the device information is converted to a JSON object and send to the server using ajax as a POST method.



Update Device

Select Device: Gen(3239f2d3-f248-476f-94a5-29f74836183a)

Device Type Thermal sensor

Input Pins

□1 ☑2 ☑3 □4 □5 □6 □7 □8 □9

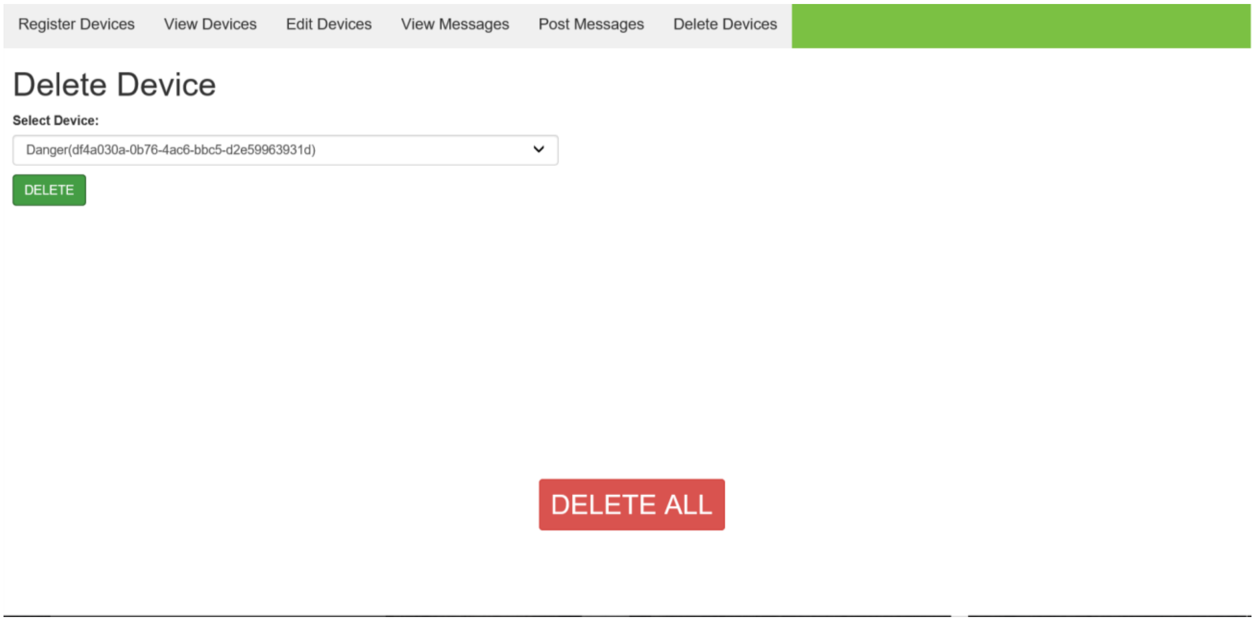
Output Pins

□1 □2 □3 □4 ☑5 ☑6 □7 □8 □9

Fig 6.7

6.2.4 Device Deletion Front-end

The device to be deleted can be selected from the combo box which is already populated with the device names. Then pressing the delete button will delete the device configuration from the server. The delete all button will delete all the devices.



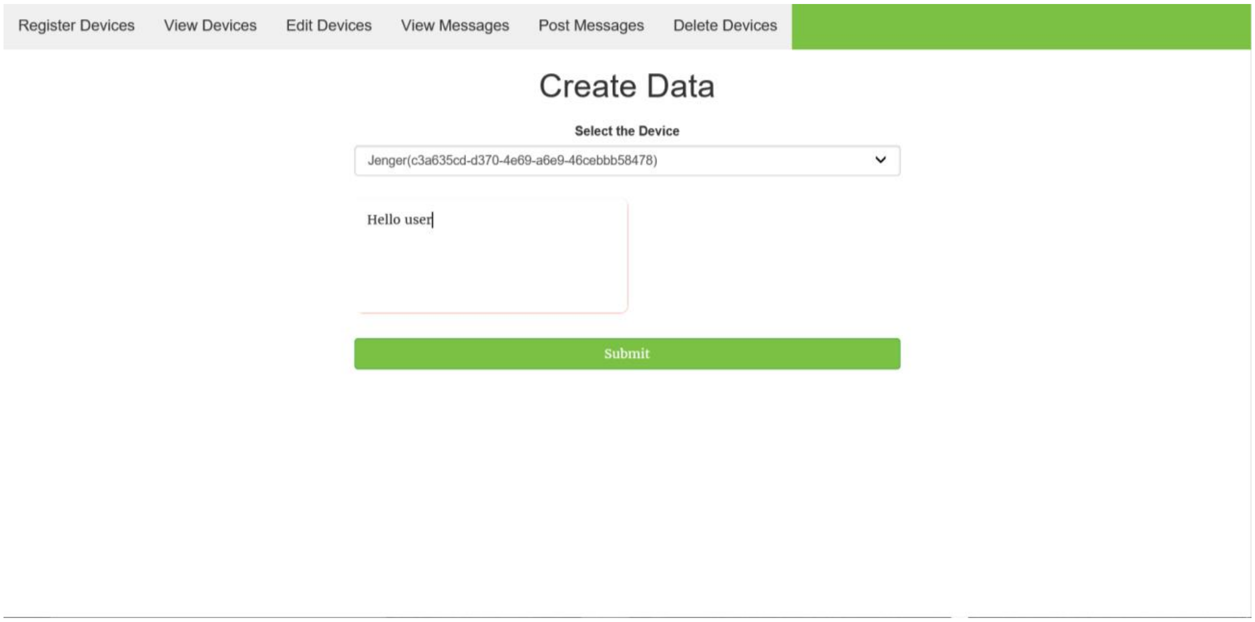
Delete Device

Select Device: Danger(df4a030a-0b76-4ac6-bbc5-d2e59963931d)

Fig 6.8

6.2.5 Messaging Front-end

In the actual scenario, the message will come from the raspberry pi. The message will contain the device name and id along with actual message. The simulation of the message is shown in the next two steps,



Create Data

Select the Device: Jenger(c3a635cd-d370-4e69-a6e9-46cebbb58478)

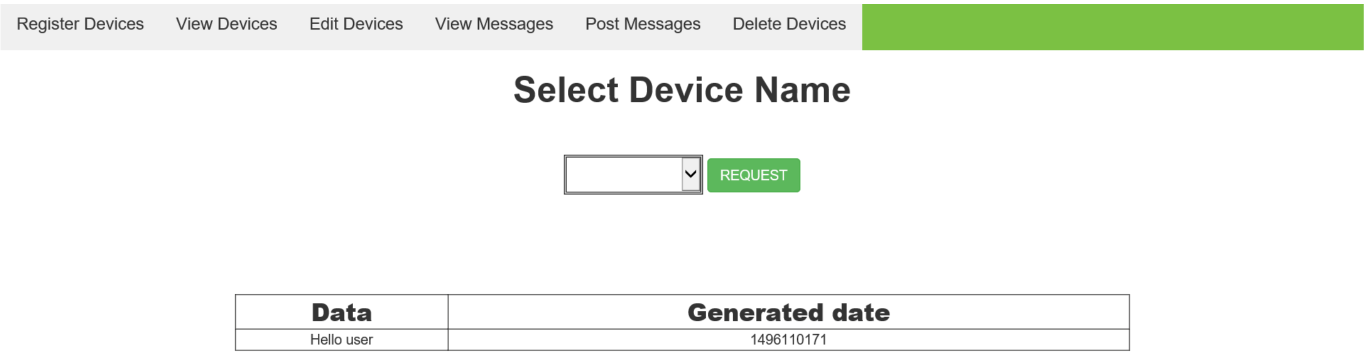
Hello user

Fig 6.9

After selecting the device from which the message is send, the message can be written in the text box. On clicking the submit button, the message is sent to the server.

6.2.6 Message History Front-end

The below figure Fig 6.10 will show the messages that are received an displays along with the registered date in epoch format.



Data	Generated date
Hello user	1496110171

Fig 6.10

6.3 Android Application

The android application demonstrates the basic CRUD functionalities and the device registrations. The following activities are developed in application.

6.3.1 Main Activity Front-end

The android application has a main activity screen. The screen contains two buttons for adding the device and viewing all the added devices.

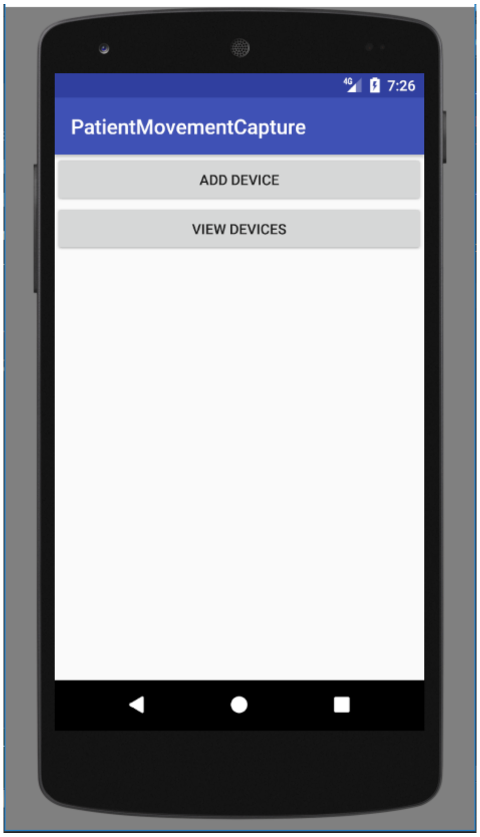
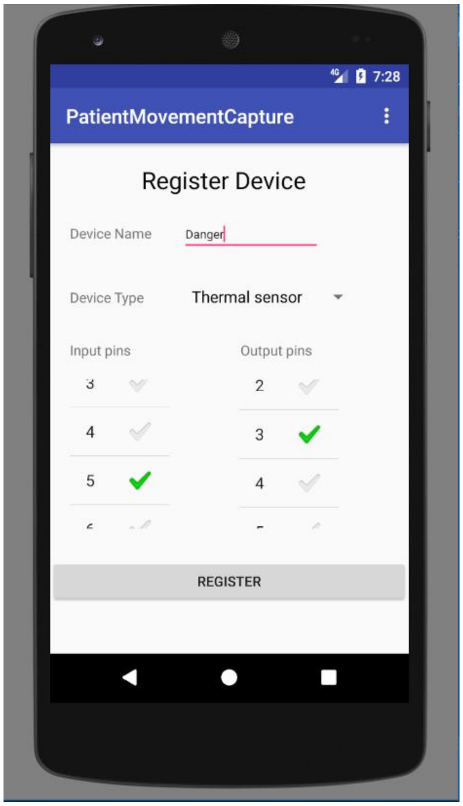


Fig 6.11

6.3.2 Registration Front End



Register Device

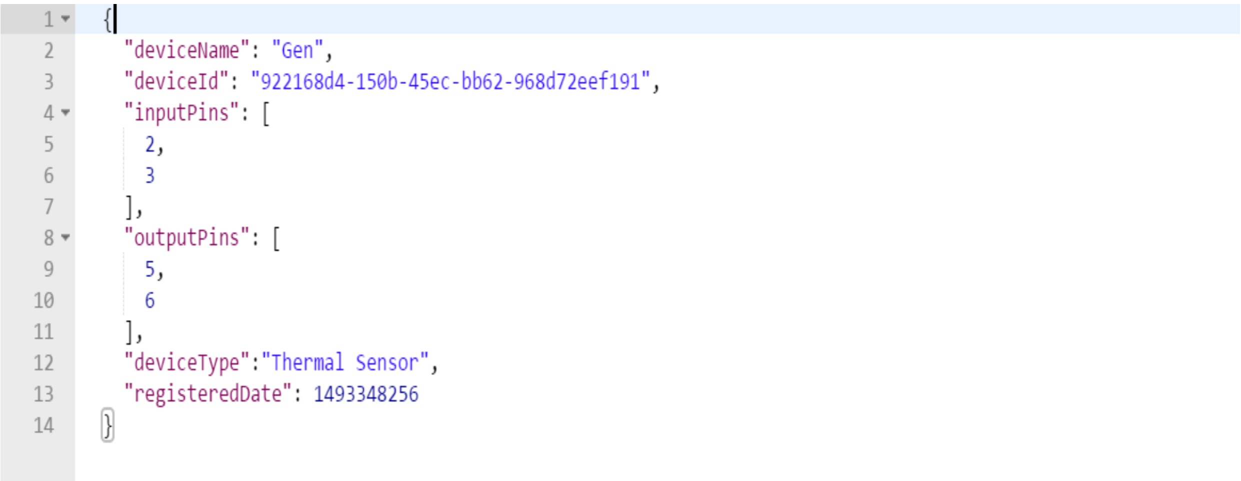
Device Name: Danger

Device Type: Thermal sensor

Input pins		Output pins	
3		2	
4		3	✓
5	✓	4	

Fig 6.12

The device name can be added to the edit box and the device type can be selected from the given selection. The sensors will have input/output pins. Specific pins will be used for getting input and output. The pins that are connected are selected here and then the register button is selected. The information is converted to a JSON object las shown in fig 6.13



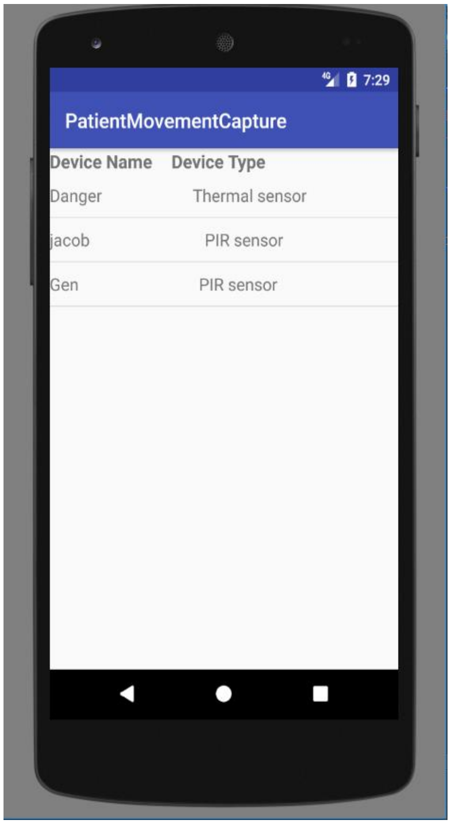
```
{
  "deviceName": "Gen",
  "deviceId": "922168d4-150b-45ec-bb62-968d72eef191",
  "inputPins": [
    2,
    3
  ],
  "outputPins": [
    5,
    6
  ],
  "deviceType":"Thermal Sensor",
  "registeredDate": 1493348256
}
```

Fig 6.13

Then they are sent to the server using HTTPConnection object method. After it is processed and stored in the server.

6.3.3 Device list Front End

The device information is got from the server while the page is loaded and the device information is got as an array of JSON objects from the server. Then it is loaded in to a table in the page.



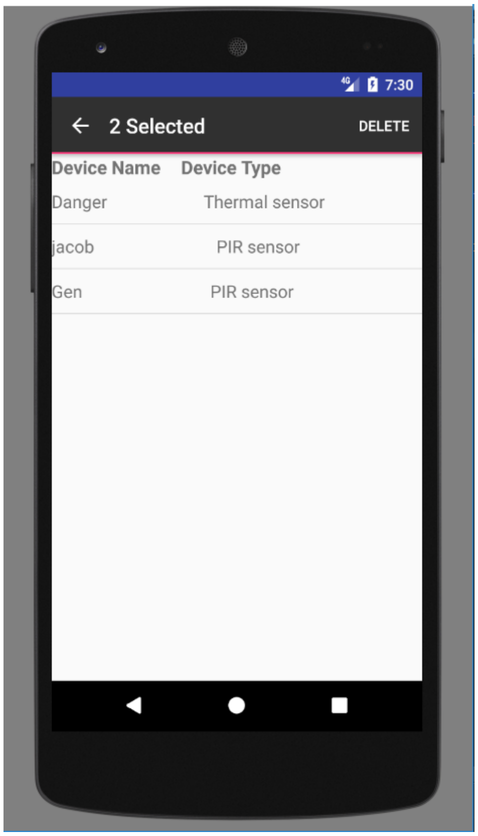
PatientMovementCapture

Device Name	Device Type
Danger	Thermal sensor
jacob	PIR sensor
Gen	PIR sensor

Fig 6.14

6.3.4 Device Deletion Front-end

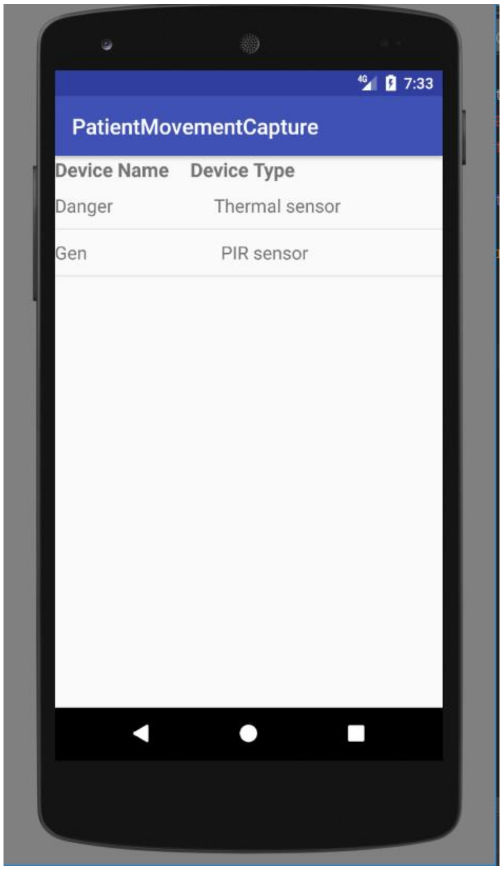
The device view activity can also be used for deleting the devices. To delete the device, long press on the device and select it as shown in fig. 6.8.



Device Name	Device Type
Danger	Thermal sensor
jacob	PIR sensor
Gen	PIR sensor

Fig 6.15

Then press the delete button in the action bar. The device is deleted from the server by sending a HTTP request message.



Device Name	Device Type
Danger	Thermal sensor
Gen	PIR sensor

Fig 6.16

The two devices were deleted from the list as shown in Fig 6.16.

The above applications were some of the example implementations to demonstrate that the web services can be accessed from any of the platforms that can transfer data through HTTP protocol.

Thus, our web services prove to be platform independent and can work over any IoT device and can handle any number of consumers.

CHAPTER 7
CONCLUSION

This project mainly aimed to develop the gateway and provide web services which will facilitate any third-party application to communicate with IoT devices. We developed it because there was a need for a platform to efficiently communicate the information as soon as received. In Cerner, it is of high importance that patient data be regularly transmitted across all devices. The HealthCare Provider needs to be notified in case of any emergency that the patient may face. This called for a system where the patient data should be sent across without loss and should be available for any number of clients.

This project dealt with developing an architecture to support messaging from any platform to other. The following were the major tasks that the project concentrated on: Registering devices on IoT, sending messages from one component to other, Restful Services for the system, Handling the delivery of the message.

The report describes about the communication gateway between the user application and the sensor devices. This is the gateway for the main project. With this gateway, we can undertake the conversion of the raw data to useful information and then creating the IoT model for the sensor system. These will be created in future work of this research project.

CHAPTER 8

REFERENCES

1. ReSTful web services - https://www.ibm.com/developerworks/library/ws-restful/index.html
2. Message Oriented Middleware - http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingComponentsIntro.html
3. RabbitMQ - https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html
4. ReSTful web services - https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html