Information Technology Course
Module Mobile Computing
by Pro. Dr. Armin Lehmann

SMART-HOME INFOTAINMENT USING AN EMULATED IOT ENVIRONMENT

Md Nahin Islam
mdnislam@stud.fra-uas.de | Matrikal No: 1273933

***Abstract*—This report introduces an infotainment simulation of Smart Home Use-Case exclusively developed in Java Springboot for the "Mobile Computing" course at the Frankfurt University of Applied Sciences.**

Here the IoT system receives song-lists of multiple users measuring their distances from the sensor which afterward makes a new playlist assembling the songs based on their priority. The priority is made based on the common songs among the users' playlist. Later IoT triggers the music actuator to play the songs. In addition, if the sensor finds no one within the range, Iot clears the playlist. Lastly, the updated playlist, connected users and the current song playing in the actuator is monitored on the browser.

***Kye words* – IoT, Distance Sensor, CoAP, Cf.**

I. INTRODUCTION

Smart home systems achieved great popularity in the last decades as they increase the comfort and quality of life. Here we will explore the concept of smart home in terms of 'infotainment' with the integration of IoT gateway services to it, by embedding intelligence into sensors and actuators. IoT contributes to the internet connection, incorporated with a variety of sensors. Sensors may be attached to home-related appliances or may stay standalone-device. And so, it embeds computer intelligence into home devices to provide ways to measure home conditions and monitor home.

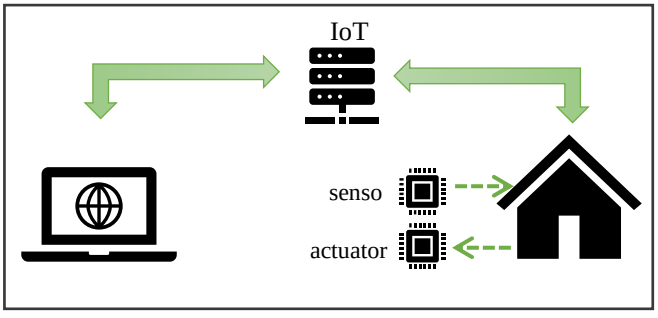

Figure 1: Basic architecture of smart-home system.

Internet of Things (IoT) is basically the concept of establishing communication between things of the real world through the internet; mainly devices that can sense the environment and do something in response. Similar to the HTTP protocol, IoT uses the Constrained Application Protocol (CoAP) designed for machine-to-machine (m2m) applications. Here, Californium (Cf) framework is used for building the IoT application exclusively in java environment with the help of the Springboot framework as a Maven Project. I will be discussing further the components and workflow in the upcoming articles.

This project is made on the concept of simulating 'Infotainment'. Infotainment refers to the combination of 'information' and 'entertainment'. Keeping this in mind, I have tried to develop such an application by which IoT will be notified with song information of an individual's mobile device within a certain distance coverage of a human presence sensor, namely *distance sensor* regarding this project. After permission from the users, a sensor will fetch the songs from the mobile and send the information of the songs and user names to the IoT. After getting the information, IoT will play some priority based songs with the help of an actuator.

II. USE-CASE OVERVIEW

To understand the project idea and its use-case we can divide into a few segments as follows.

1. ***Distance Sensor notifies presence information,*** which means it detects human presence with distance. That is within the distance covered by the sensor how far that person is. And it's hosting device, the device in which the sensor is integrated with, sends the song information to the IoT as a response in JSON format which then gets converted into java object inside IoT.
2. ***IoT Gateway*** acts as the medium for the communication of m2m. This performs the application layer functions between IoT nodes

and other entities. Enables local, short-range communication between IoT devices.

3. ***Music Actuator*** then gets a logical command from IoT to play a song.
4. ***Monitoring through Browser*** helps to visualize the current transaction and state of the running system. In this portion, the information is passed in XML format.

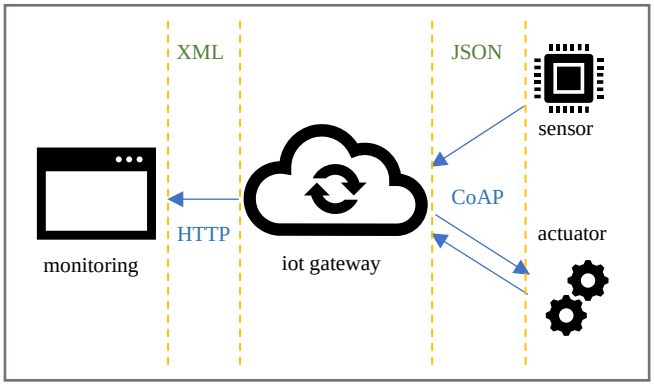

Figure 2: Application architecture.

In this report, one major part of the application development is an adequate description of CoAP and its framework Californium.

III. Constrained Application Protocol (CoAP)

"The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained networks in IoT. The protocol the for machine-to-machine (M2M) applications such as smart energy and building automation", quoted by the CoAP Technical team which is quite a description, to begin with.

CoAP is an Application Protocol for Constrained Nodes/Networks. It is intended to provide RESTful services not unlike HTTP while reducing the complexity of implementation as well as the size of packets exchanged in order to make these services useful in a highly constrained network of themselves highly constrained nodes.

A. CoAP Observer

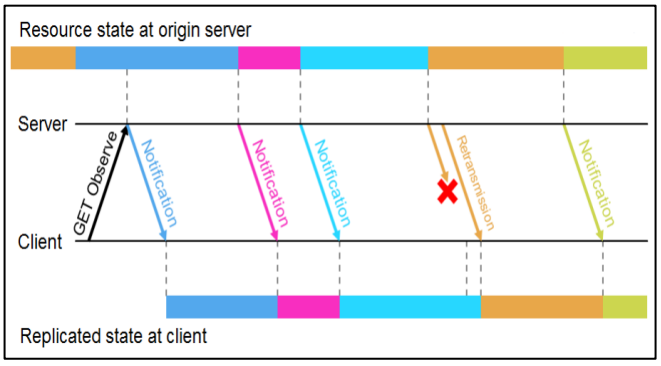

Figure 3: Example of CoAP observer state.

The state of a resource on a CoAP server can change over time. It has given the clients the ability to observe this change. However, existing approaches from the HTTP world, such as repeated polling or long-polls, generate significant complexity and/or overhead and thus are less applicable in the constrained CoAP world. Instead, a much simpler mechanism is provided to solve the basic problem of resource observation

B. Californium Framework

This framework is developed exclusively in java focusing on scalability and usability. It has made easier to build a CoAP server and the corresponding client reducing the stress of lines of codes that needs to be written in java environment. Figure 4 & 5 illustrates the architecture well.

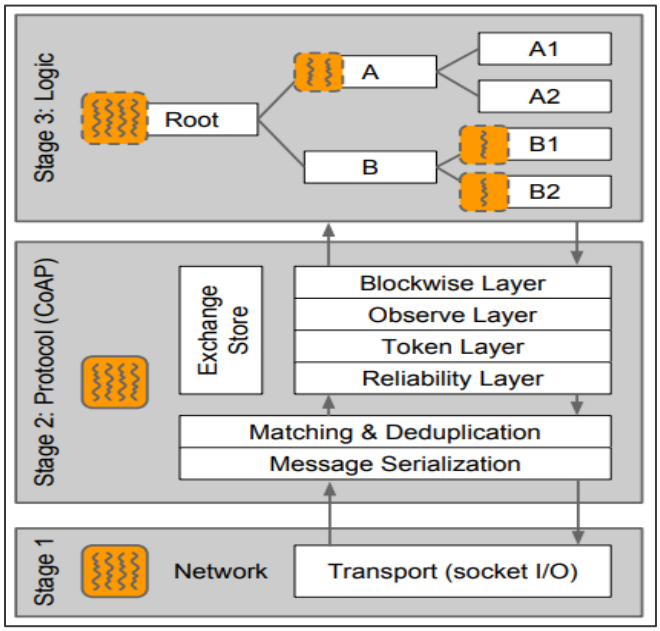

Figure 4: Californium as a Server Role.

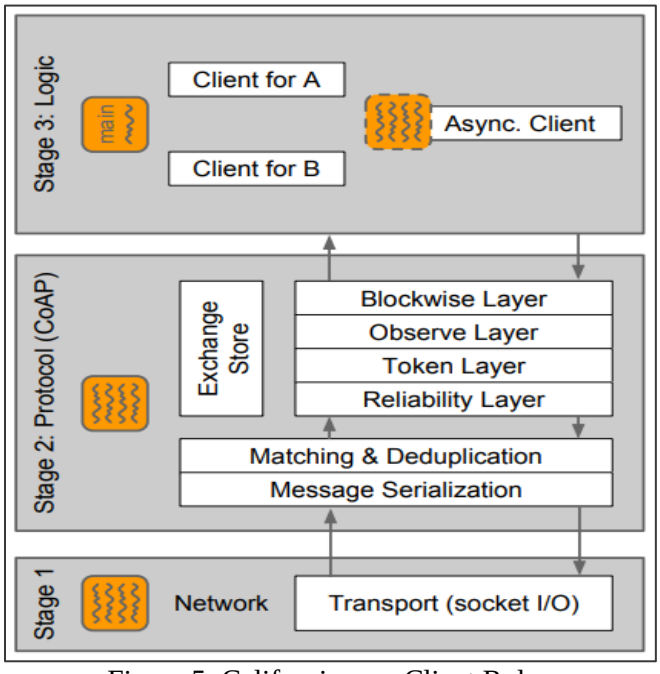

Figure 5: Californium as Client Role.

IV. PROJECT STRUCTURE

In the following sections, a broad explanation of the project development will be shown with corresponding code snippets.

A. Development Environment

Operating System: Windows 10.
Processor & RAM: intel CORE i7, 8Gb.
Programing Language & IDE: Java, Intellij.
Protocols: HTTP & CoAP(Cf).
Frontend: HTML5, CSS, Javascript(jquery).
Framework & Build tool: Springboot, Thymleaf, Maven.

B. Work Flow

a. Creating Springboot Application

Springboot is an open-source Java-based framework used to create a Micro Service. It is easy to create a stand-alone and production-ready Spring applications using Spring Boot. Spring Boot contains comprehensive infrastructure support for developing a microservice and enables you to develop enterprise-ready applications that you can **"just run"**. Tomcat is embedded as well along with starter dependency for the auto-configuration. The project is created in the following manner.
File > New > Project > Springboot > Click the check box of 'Spring' and 'Thymeleaf' then Next > Set the Artifect Name > Click Finish.

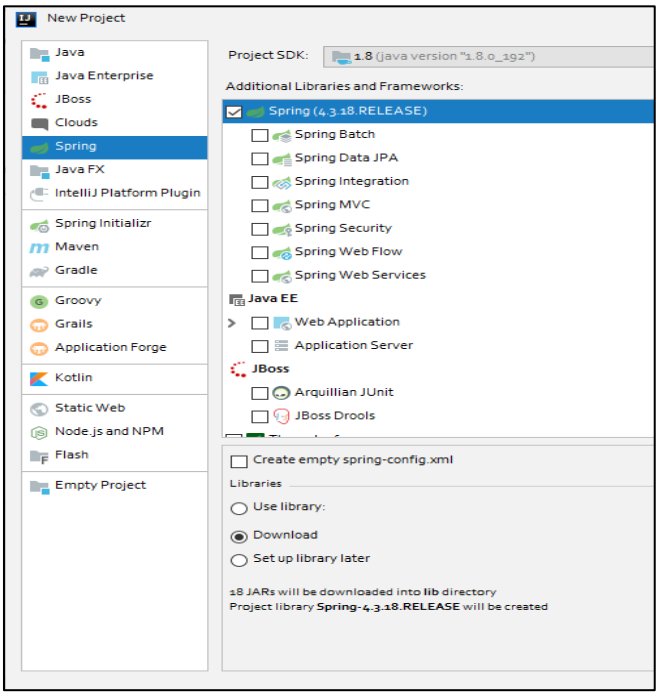

Figure 6: Springboot Project Creation.

b. Maven & Adding Dependencies

Maven is a build automation tool used primarily for Java projects. It addresses two aspects of building software: how software is built, and its dependencies. This allows users to import dependencies into their software projects and also automatically manage transitive dependencies. In order to use Maven, it is necessary to explicitly add dependencies to the Maven 'pom.xml' file. 'pom.xml' is an XML file that acts like a manifest and can be found in the project root directory.

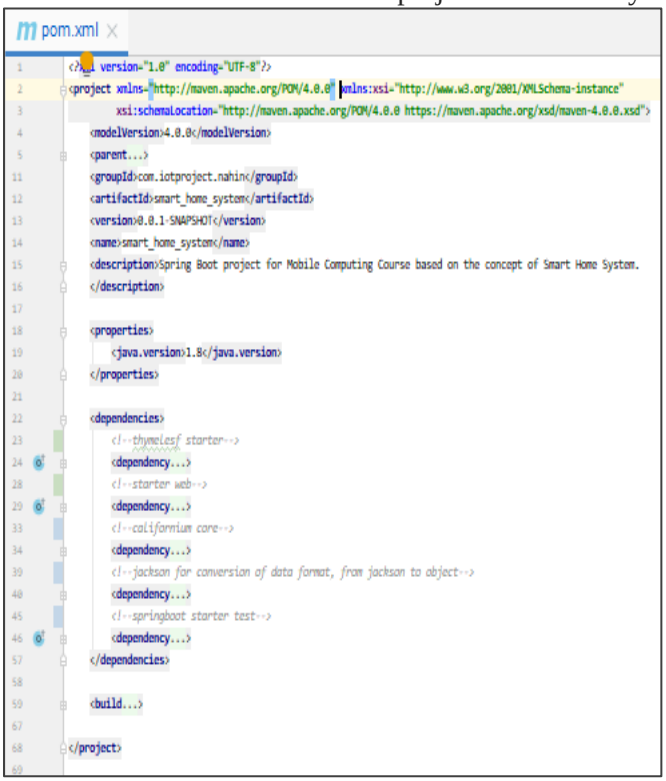

Figure 7: POM.XML file and Dependencies.

Here, in the 'pom' file 5 dependencies have been added.

1. *Thymeleaf Starter* is the dependency for maintaining the HTML file exclusively in java environment.
2. *Starter Web* helps to build web components including RESTful.
3. *Californium* is the framework for the Constrained Application Protocol for M2M communication
4. *Jackson* dependency helps to convert string format from JSON to Java Object and vice versa.
5. *Springboot Starter Test* dependency is used for testing purposes.

c. Packages & Classes

For keeping the project clean and well organized I have created several packages in the source directory under "**com.iotproject.nahin.smart_home_system**" package each of which contains java class files.

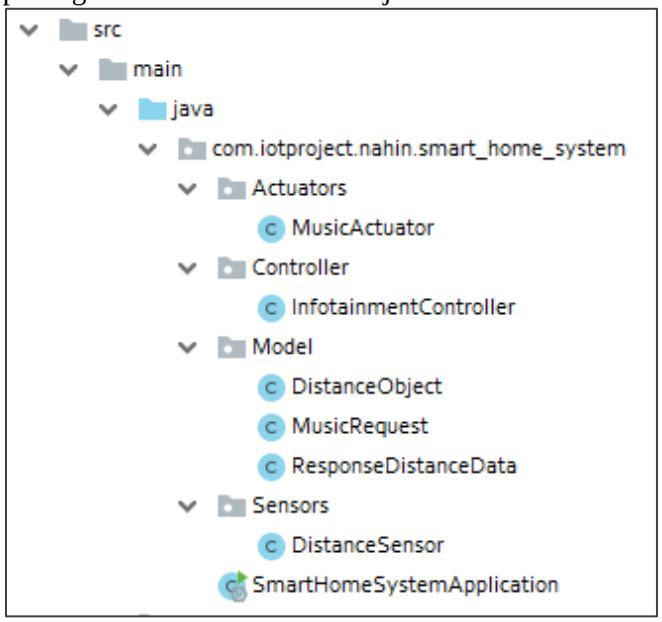

Figure 8: Java Packages with Classes.

c.1 InfotainmentController Class

First, *InfotainmentContoller* class is created inside the Controller packaged. This is a simple java class with the annotation *@Contoller*. *@Controller* is a class-level annotation which tells the Spring Framework that this class serves as a **controller in Spring MVC**.

```
@Controller
@RequestMapping("iot/")
public class InfotainmentController {

    @GetMapping("infotainment")
    public String home() { return "iotHome"; }

    @GetMapping("get-distance")
    @ResponseBody
    public ResponseDistanceData getDistance(){
        return SmartHomeSystemApplication.responseData;
    }

}
```

Figure 9: Controller Class.

Afterward, there is another annotation *@RequestMapping* that is compatible with the HTTP methods. To simply put, this annotation marks the request handler methods, that is to which URL a certain method is mapped to.

@GetMapping maps the methods with logical actions. Here this returns an HTML file to the View part.

'getDistance()' is a request handler method marked with *@ResponseBody* annotation, mean Spring treats the result of the method as the response itself which basically comes from the IoT. This method returns the playlist that is created in the IoT with the name of connected users in the response of HTTP requests.

c.2 DistanceSensor Class

After setting up the mapping a sensor class is created with some static values of the user name and a song list of each individual.

```
public class DistanceSensor extends ConcurrentCoapResource {

    public static DistanceObject distanceObject;
    public static ArrayList<String> alreadyTakenPerson = new ArrayList<>();

    public static String[] people = {"nahin", "pretom", "akib", "limon", "rony"};

    public static String[] listOfNahinSong = {"Numb - Linkin Park", "Your Sky - Coldplay", "Take Me Home,
            "Rockabye - Clean Bandit", "Annie's Song - John Denver ♥","Hiftah Zaman- Obelai"};
    public static String[] listOfPretomSong = {"Numb - Linkin Park", "Viva la Vida - Coldplay", "Rockabye
            "Your Sky - Coldplay", "Wavin' Flag - K'NAAN (FIFA 2010)"};
    public static String[] listOfAkibSong = {"Annie's Song - John Denver ♥", "Rockabye - Clean Bandit", "
            "Numb - Linkin Park", "Revolverheld feat. Marta Jandová - Halt Dich an mir fest"};
    public static String[] listOfLimonSong = {"Take Me Home, Country Roads - John Denver ♥", "Panchi. Jal
            "Sing Nachtigall Sing - Evelyn Künneke", "Revolverheld feat. Marta Jandová - Halt Dich an mir
    public static String[] listOfRonySong = {"Sing Nachtigall Sing - Evelyn Künneke", "Numb - Linkin Park
            "Viva la Vida - Coldplay", "Annie's Song - John Denver ♥";
```

Figure 10: Distance Classes with static values.

This class inherits the CoAP functionalities by extending the 'ConcurrentCoAPResource' class. Here one method called 'handleGet()' is needed to be overridden for the availability of the sensor information to the IoT through the CoAP Protocol. Inside this method the static string values, which are some string arrays, id first converted into JSON format with the help of *ObjectMapper* (an exclusive functionality provided by the *Jackson* dependency).

```
/*converts the object data into json format*/
@Override
public void handleGET(CoapExchange exchange) {
    try {
        ObjectMapper objectMapper = new ObjectMapper();
        String jsonString = objectMapper.writeValueAsString(distanceObject);
        exchange.respond(jsonString);
    } catch (Exception e) {
    }
}
```

Figure 11: sending information to IoT.

Now the IoT needs to be updated with information on current users and their song lists from the sensor. For this, an observer is needed to be attached inside the constructor which will be sending information to the IoT over time.

```
public DistanceSensor(String name) {
    super(name);
    distanceObject = new DistanceObject();

    setObservable(true);
    setObserveType(CoAP.Type.CON);
    getAttributes().setObservable();

    Timer timer = new Timer();
    timer.schedule(new ContinuousTask(),  delay: 0,  period: 3000);
}
```

Figure 12: CoAP Observer.

'ContinuousTask()' is simply a Timmer Class that combines a user name with a song list.

```
private class ContinuousTask extends TimerTask {
    @Override
    public void run() {

        distanceObject.setDistance(getRandomDistance() + "");
        String person = getRandomPerson(people);

        switch (person) {
            case "nahin":
                distanceObject.setPerson(person);
                distanceObject.setSongList(songListOfNahin);
                break;
            case "pretom":
                distanceObject.setPerson(person);
                distanceObject.setSongList(songListOfPretom);
                break;
            case "akib":
                distanceObject.setPerson(person);
                distanceObject.setSongList(songListOfAkib);
                break;
            case "limon":
                distanceObject.setPerson(person);
                distanceObject.setSongList(songListOfLimon);
                break;
            case "rony":
                distanceObject.setPerson(person);
                distanceObject.setSongList(songListOfRony);
                break;
        }

        changed();
    }
}

/*random selection from people list*/
public String getRandomPerson(String[] array) {...}

/*random distance for selected person from randoms*/
public int getRandomDistance() {
    return new Random().nextInt( bound: 21);
}
```

Figure 13: User and Songlist random selection.

c.3. DistanceObject Class

This class a Model part of the Springboot MVC.

```
public class DistanceObject {

    String distance;
    String person;
    String songList[];

    public String getDistance() { return distance; }
    public void setDistance(String distance) { this.distance = distance; }
    public String getPerson() { return person; }
    public void setPerson(String person) { this.person = person; }
    public String[] getSongList() { return songList; }
    public void setSongList(String[] songList) { this.songList = songList; }
    @Override
    public String toString() { return super.toString(); }
}
```

Figure 14: Model Class of Sensor Data.

c.4 SmartHomeSystemApplication Class

@SpringbootApplication annotation defines this class with the main method as the starting point of the Springboot Application. Putting simply, this is where the system knows this is a Springboot Application.

```
@SpringBootApplication
public class SmartHomeSystemApplication {

    @Autowired
    public static ResponseDistanceData responseData;
    public static Map<String, String[]> songPriorityMapping;
    public static Map<String, Integer> finalSortedSongs;
    public static ObjectMapper objectMapper;

    public static void main(String[] args) {...}

}
```

Figure 15: Main Class.

CoAP Server and Client are initialized and configured here at the port '5683' and '5684' for the distance sensor accordingly.

```
/*coap server configuration*/
CoapServer distanceSensorServer = new CoapServer( ...ports: 5683);
distanceSensorServer.add(new DistanceSensor( name: "distance"));
distanceSensorServer.start();

CoapServer musicActuatorServer = new CoapServer( ...ports: 5684);
musicActuatorServer.add(new MusicActuator( name: "music"));
musicActuatorServer.start();

/*client config with the corresponding server*/
CoapClient distantSensorClient = new CoapClient( uri: "coap://localhost:5683/distance");
CoapClient musicActuatorClient = new CoapClient( uri: "coap://localhost:5684/music");
```

Figure 16: CoAP Server & Client Configuration.

Moving forward, in the following code snippet variable *jasonFormatOfDistanceSensor* gets the JSON

```
CoapObserveRelation relation = distantSensorClient.observe(new CoapHandler() {
    @Override
    public void onLoad(CoapResponse coapResponse) {

        String jsonFromDistanceSensor = coapResponse.getResponseText();
        System.out.println(jsonFromDistanceSensor);
```

Figure 17: Getting JSON data from CoAP.

data from the CoAP observer and later convert it into java object. Later from the response data, the distance information is measured. If the response distance is less than 11units then the information of the person and his song list is put into a Hashmap Collection as key-value pair.

```
if (Integer.valueOf(distanceObject.getDistance()) < 11) {
    if (sensorData.containsKey(distanceObject.getPerson()) == false) {
        sensorData.put(distanceObject.getPerson(), distanceObject.getSongList());
    }
} else {
    if (sensorData.containsKey(distanceObject.getPerson()) == true) {
        sensorData.remove(distanceObject.getPerson());
    }
}
```

(a)

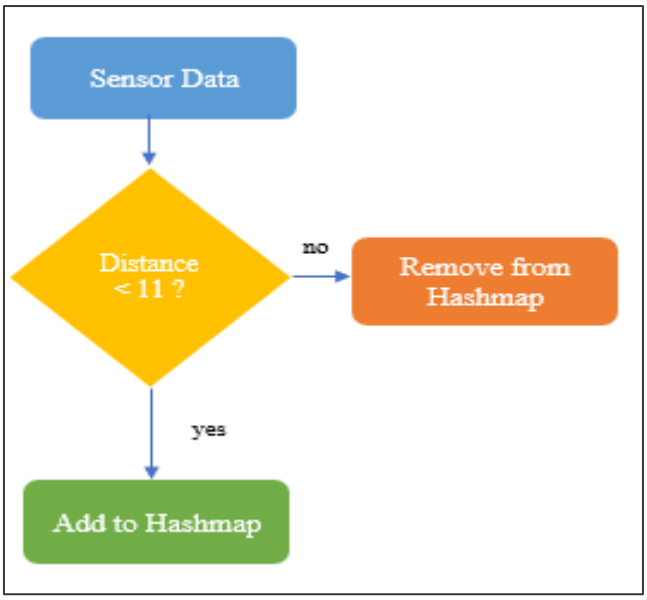

Figure 18: Adding or Removing data to Hashmap.

Next, each song of current information from the sensor is checked whether it is in a new hashmap or not. If a single song is not in the hashmap collection then it is added otherwise the integer value of the second argument of the hashmap is increased. If later any user from the connected list gets removed from the list based on a new distance then the value of a single song is decreased.

```
iotPlayList = new HashMap<>();
for (String key : sensorData.keySet()) {

    String songs[] = sensorData.get(key);

    for (int i = 0; i < songs.length; i++) {
        String songName = songs[i];
        if (iotPlayList.containsKey(songName) == false) {
            iotPlayList.put(songName, 1);
        } else {
            iotPlayList.put(songName, iotPlayList.get(songName) + 1);
        }
    }
}

Map<String, Integer> sortedIoTPlayList = sortByReverseOrder(iotPlayList);

/*  array of songs based on priority from sorted playlist*/
Collection<String> finalSongCollection = sortedIoTPlayList.keySet();
String[] finalPlayList = finalSongCollection.toArray(new String[finalSongCollection.size()]);
```

Figure 19: Creating the final Playlist.

After creating the final playlist IoT sets the data into a data model class, *ResponseDistanceData*.

```
responseData.setPerson(distanceObject.getPerson());
responseData.setDistance(distanceObject.getDistance());
responseData.setSongList(finalPlayList);
responseData.setGuestList(guestList);
```

Figure 21: Setting information into model class.

Later from the browser, AJAX (GET Request) call is made for fetching the playlist information of the IoT from this class and show them on the browser page. In the meantime, IoT calls the *MusicActuator* for playing the first song of the playlist and so on.

```
//call to music actuator to play song
if(finalPlayList.length > 0) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            MusicRequest musicRequest = new MusicRequest();
            musicRequest.currentSong = finalPlayList[0];
            String jsonRequest = null;
            try {
                jsonRequest = objectMapper.writeValueAsString(musicRequest);
            } catch (JsonProcessingException e) {
                e.printStackTrace();
            }
            try {
                musicActuatorClient.post(jsonRequest, MediaTypeRegistry.APPLICATION_JSON);
            } catch (ConnectorException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }

            responseData.setCurrentSong(finalPlayList[0]);
        }
    }).start();
}
```

Figure 20: Calling Music Actuator.

```
public class MusicActuator extends ConcurrentCoapResource {

    MusicRequest music = null;

    public MusicActuator(String name) {
        super(name);
        music = new MusicRequest();
    }

    /*converts the object data into json format*/
    @Override
    public void handlePOST(CoapExchange exchange) {
        exchange.accept();

        try {
            String jsonRequest = exchange.getRequestText();
            ObjectMapper objectMapper = new ObjectMapper();

            MusicRequest request = objectMapper.readValue(jsonRequest, MusicRequest.class);
            music.currentSong = request.currentSong;

            String jsonString = objectMapper.writeValueAsString(music);
            exchange.respond(CoAP.ResponseCode.CREATED, jsonString);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

Figure 21: Music Actuator Class.

c.4. AJAX Call and XML Data

```
function getContiniouslyAllData() {
    $.ajax({
        url: '/iot/get-distance',
        type: "GET",
        dataType: "xml",
        contentType: "application/xml",
        success: function (xmlData) {
            $(xmlData).find("ResponseDistanceData").each(function () {
                var personName = $(this).find('person').text();
                $('#personName').html(personName);

                var distance = $(this).find('distance').text();
                $('#distance').html(distance);

                var songsList = $(this).find('songList');
                var currentSongName = $(this).find('currentSong');
                console.log(currentSongName);
                $('#currentSongName').html(currentSongName);

                $("#cardSongs").empty();
                $.each(songsList, function (index, value) {
                    if (index !== 0) {
                        var songId = "songId" + index;
                        var cardSong = '<div class="card col-md-2  col-lg-2" style="background-color: crimson; height: 210px; margin: 8px">' +
                            '<div class="card-inner" style="text-align: -moz-center" id=' + songId + '>' + value.innerHTML + '</div></div>';
                        $("#cardSongs").append(cardSong);
                        //console.log(index)
                    }else{
                        $("#cardSongs").append("No song is currently available!");
                    }

                });

                var guestList = $(this).find('guestList');
                $("#guestList").empty();
                $.each(guestList, function (index, value) {
                    if (index !== 0) {
                        var songId = "songId" + index;
                        var guestName = '<li class="list-group-item" style="border: none" id=' + songId + '>' + value.innerHTML + '</li>';
                        $("#guestList").append(guestName);
                    }else{
                        $("#guestList").append("Searching for guests..");
                    }

                });

            });
        }
    });
}
```

Figure 22: AJAX Call for fetching data.

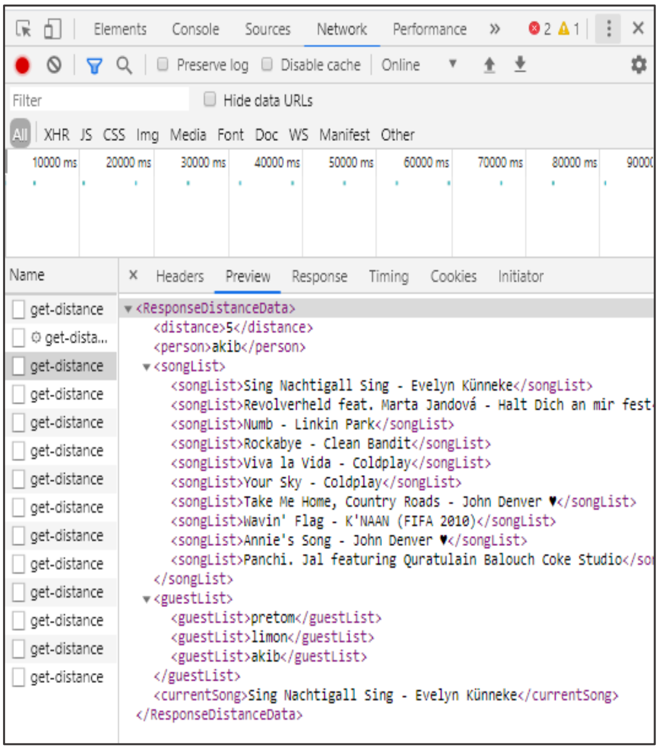

Figure 25: XML Data

c.5. Traceroute traffic with Wireshark

Protocol (CoAP)

Here we can see the response through CoAP from the data to IoT is JSON data accepting at port 5683.

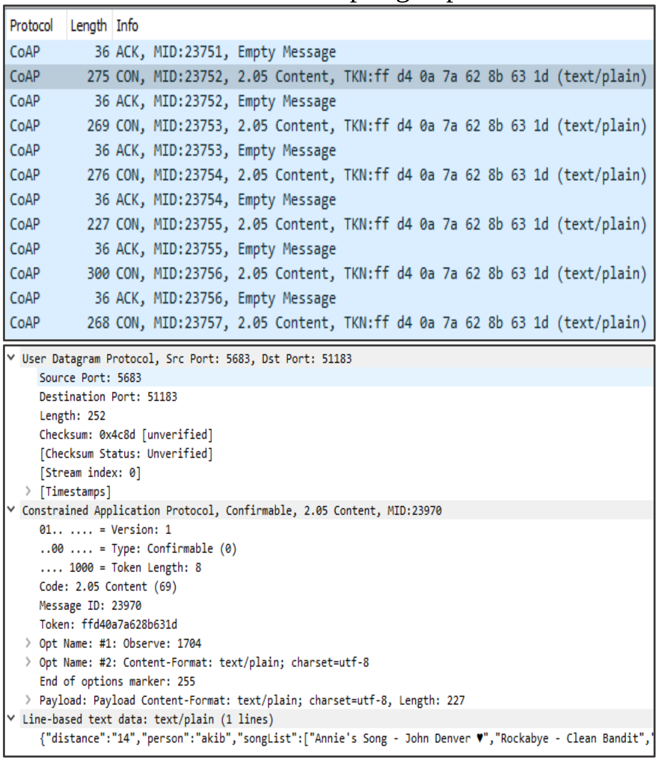

Figure 24: Data from the sensor through CoAP.

Protocol (HTTP)

Here the protocol shows some GET request from the browser to the IoT for new playlist after some time and in response, it accepts XML data at port 8085.

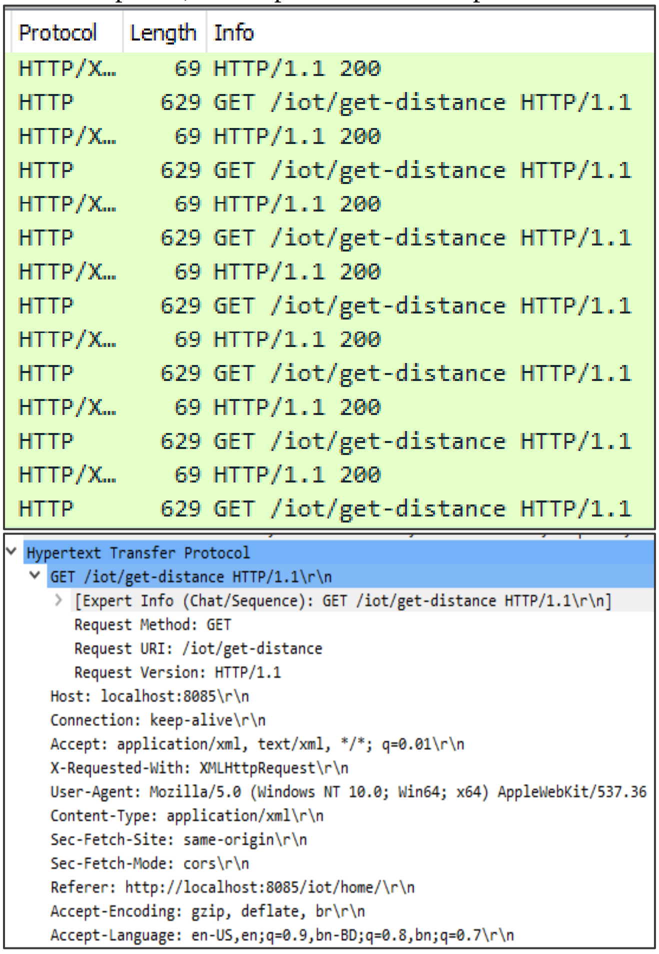

Figure 24: GET Request through HTTP.

V. VISUAL STATE

API: http://localhost:8085/iot/infotainment/

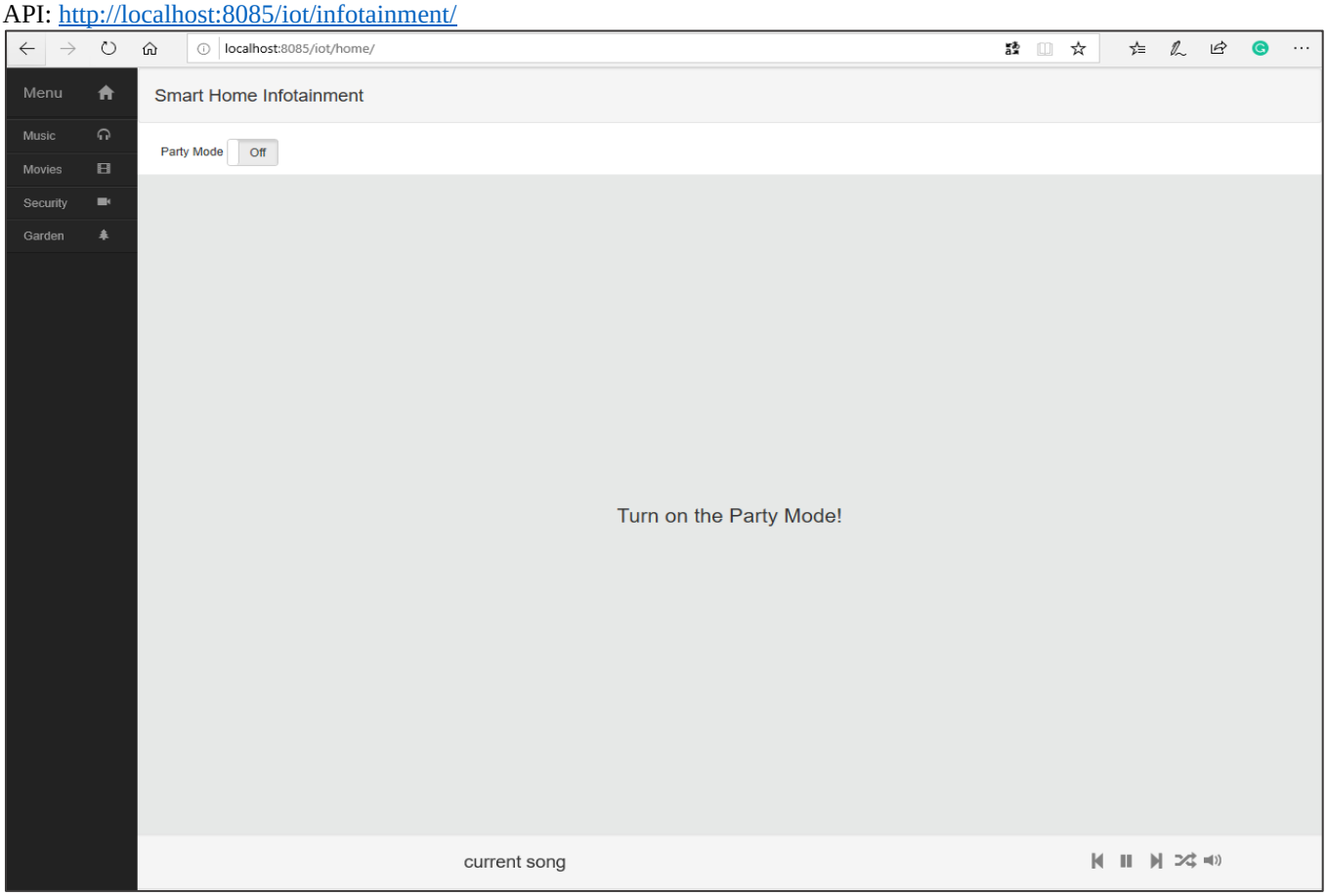

Figure: Before AJAX Call for fetching data from IoT.

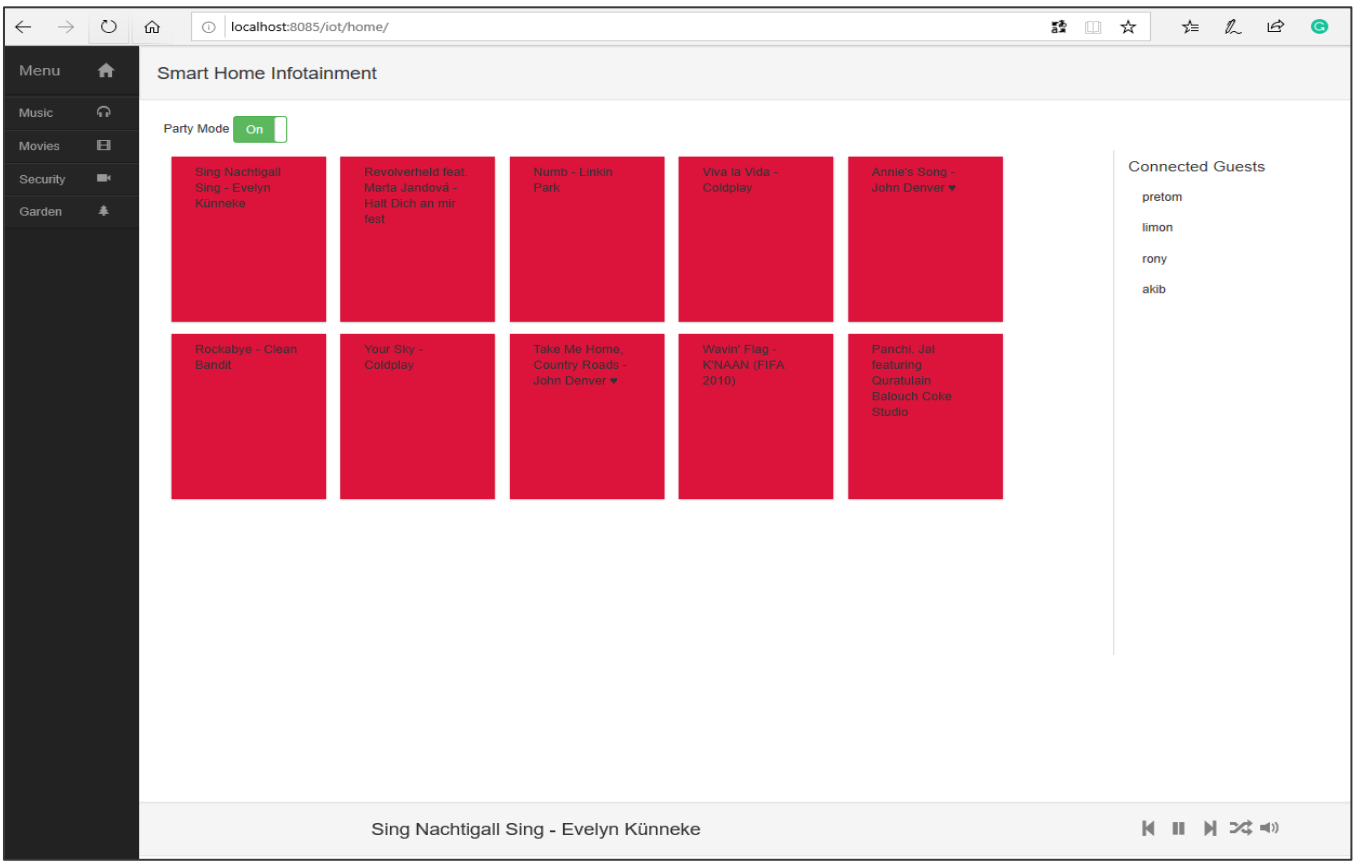

Figure: After AJAX Call for fetching data from IoT.

VI. DISCUSSION

In real-time environment device integrated with sensor will detect human presence with cell phone then it will ask permission for accessing the song list of the cell phone. After getting access it will then get the songs in account and compare with other playlist and will create a new play on basis of common songs of each cell phone and play the songs. As for simulation I have created a virtual sensor with some static data that send these informations to the IoT and IoT sends single song to a virtual actuator to play the list.

REFERENCES

[1] Internet of Things (IoT) for Automated and Smart Application. Link

[2] What is IoT? – A Simple Explanation of the Internet of Things. Link

[3] Hands-on with CoAP by eclipse.org. Link

[4] Observing Resources in CoAP. Link