The Design and Implementation of a RESTful IoT Service Using the MERN Stack

Preston Porter and Shuhui Yang
Department of Mathematics, Statistics, and Computer Science
Purdue University Northwest
Hammond, USA
{porter31, shuhuiyang}@pnw.edu

Xuefeng Xi
School of Electonic & Information Engineering
Suzhou University of Science and Technology
Suzhou, China
xfxi2018@gmail.com

***Abstract*— With the increasing number of Internet of Things applications (IoT), an enormous number of IoT devices are connected to the Internet and require processing from the Cloud. Although the Cloud theoretically has unlimited resources, it is still a significant challenge for the Cloud to perform a real time response. This project is a designed and developed RESTful IoT service for a quick communication between IoT devices and the Cloud, using the MERN stack. As a gateway, the proposed system gathers data from IoT devices such as wireless sensors, for the Cloud to further process on it. The Representational State Transfer (REST) model is adopted in the proposed system. It has a good performance in terms of response time and scalability. Experiments further verified the performance of the proposed system.**

Keywords—Cloud, Design, Implementation, IoT, MERN Stack, RESTful, Wireless Sensor Network,

I. INTRODUCTION

Cloud computing was popularized in the past decade to virtually provide unlimited computation, storage, and service [1][2]. However, the massive growth in the scale of data generated for the Cloud to process in a real time manner has been observed, and becomes a new challenge [3]. The other emerging field, the Internet of Things (IoT), connects a large number of physical devices to the Internet to realize intelligent systems [4][5]. IoT devices are characterized by small objects with limited storage and processing capacity. The integration of Cloud computing and IoT solves this issue in IoT and presents an architecture that is considered as a part of the Internet of the future.

However, with the connection of an enormous number of small physical devices, which generate data 24/7, the data processing ability of Cloud computing is further challenged. Additionally, real-time processing is desired. While improved data processing technology at the Cloud end is developed, advanced data gathering/pre-preprocessing at the gateway of the Cloud and IoT are also drawing attention. For example, Edge computing [6][7] is a solution to pull the computing closer to the location it is needed for the improvements of response times and bandwidth efficiency.

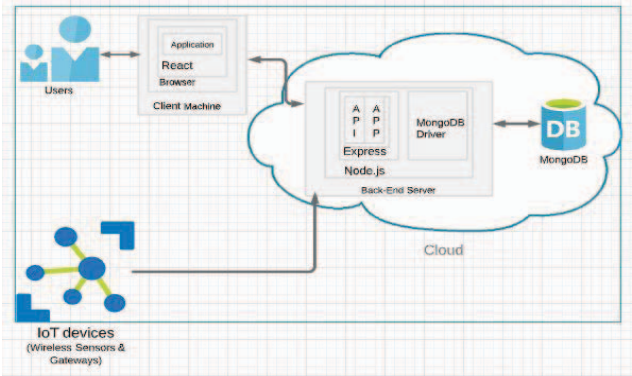

Fig. 0 System Illustration.

In this paper, we work towards an effective and efficient data gathering system that collects data from IoT devices, such as wireless sensors, and feeds them to the Cloud for further processing and data storage. The Representational State Transfer (REST) model is adopted to develop a RESTful IoT service, and the trending MERN Stack is used to build the web-based IoT service, for scalability and efficiency of development. Fig. 0 is the illustration of the proposed data gathering system.

The contribution of this work includes

- The study of the application of the MERN stack for a RESTful web-based IoT service, for efficient data gathering purposes.
- The design and implementation of a RESTful IoT system using the MERN stack.
- Performance analysis of the proposed system through experiments.

The rest of the paper is organized as follows. Section II introduces the research background. Section III presents the proposed RESTful IoT Service using the MERN stack, including the system design and implementation details. IV is the experiments to evaluate the proposed system. Section V concludes the paper.

978-1-7281-4121-3/19/$31.00 ©2019 IEEE
DOI 10.1109/MASSW.2019.00035

II. PRELIMINARIES

A. IoT, Cloud, and RESTful Service

The integration of IoT and Cloud computing provides an architecture that interconnects the mobile sensing/monitoring devices in IoT system with data processing virtual server in the Cloud [2][8]. In a general cloud-IoT architecture, there are 5 layers, including (1) end-user interaction, (2) cloud structure, (3) Internet access, (4) IoT backbone, and (5) IoT devices. They work together to transmit data collected from IoT devices , generating results and reports for end users to make decision. As stated in [2], new challenges arise when IoT devices connect to the cloud. In this novel network architecture, new protocols that facilitate big data streaming from IoT devices to the cloud are needed.

In [9], Constrained Application Protocol (CoAP) is developed as the communication protocol for the data stream from wireless sensor network to the Internet. However, in practice, the direct connection of IoT devices, such as the power impoverished wireless sensors, to the Internet is not feasible, due to the complex network congestion handling issues. A gateway system at the IoT backbone layer is needed for data gathering purpose. This gateway is expected to work with, or instead of, CoAP to full data from sensors and forward to the Cloud in an effective and efficient way.

Representational State Transfer (REST) is a model and architectural style for web-service [10][11], where the native HTTP operations are mapped on the four fundamental database operations. When this model is used for gateway API design, the Cloud will be able to communicate with devices such as wireless sensors through it. The gateway with a RESTful service is to transmit data between Cloud and IoT devices. We will be using this model for a RESTful IoT service design.

B. The MERN Stack

In web development terminology, a "stack" refers to a full stack set of technologies to create a working web application. It is common for businesses to employ developers as either a 'front-end', 'back-end', or 'full-stack' developer. The front-end developer works on the 'front end' part of a web application, such as the user interface, which is also known as the client side of an application. A back-end developer works on the 'back end' part of a web application, which the users cannot see, and keeps the web application up and running, such as the server and the database. A full stack developer works on both the client side and server-side components to make a fully function web application.

The LAMP stack is a common web stack solidified in the early 2000s which consists of a Linux Operating system, an Apache web server, a MySQL database, and a server-side scripting language which is most commonly PHP. All of these technologies were backend components while JavaScript was typically a component of the front end.

Today as Javascript becomes more efficient and powerful, the old stacks are becoming replaced with better ones. According to a 2019 stack overflow survey of 90,000 professional developers, Javascript has become the most popular programming language by 69.7 percent [12]. In the survey Javascript has been the most popular language for the last 9 years in a row.

The MERN stack is entirely based off of Javascript, which makes it easier to use than other stacks. This reduces overhead since in the LAMP stack there are several languages, such as MySQL, PHP, not to mention the front-end languages. However, in the MERN stack everything is done in one language. The MERN stack consists of different JavaScript frameworks, MongoDB (M), Express (E), React (R), and Node.js (N). The front end consists of React, while MongoDB, Express and Node.js are the backend components.

- **MongoDB**: This is the database of the MERN stack where all the data is stored. It is a NoSQL database. It has a flexible schema, which is very scalable. MongoDB scales horizontally while MySQL scales vertically, which makes MongoDB more ideal for IoT devices like a wireless sensor, since it is easier to add more sensors to the network. Some well-known companies that use MongoDB in 2019 are Ebay, Cisco, Adobe, EA Sports, Google, Facebook, Nokia, SAP, GAP, and Verizon [13].
- **Express**: Express is a web framework for Node.js, which is responsible for setting URL routes. This is done using a RESTful methodology based on native HTTP requests such as GET, POST, DELETE, and UPDATE. Express is responsible for the middleware in a web application, which will be explained in more detail in the system architecture. The libraries in Express are to work with cookies, sessions, user logins, URL parameters, `POST` data, security headers, and many more [14]. Some well-known companies that use Express in 2019 are Uber, Accenture, and IBM [15].
- **React**: React is the front-end component of the MERN stack developed and maintained by Facebook. In our proposed system architecture, React can be adopted for front-end development in the future. Another front-end framework which is used with the other components of this stack is Angular. An advantage of React over Angular is that it has a slightly less steep learning curve than Angular. A feature of React is that it uses JSX to structure the data. It is an extension of the JavaScript language. React uses an in-memory data structure cache to display the DOM. Another feature is that it uses a virtual Document Object Model, or virtual DOM. The DOM is used for HTML and XML documents as a tree data structure where each node is an object representing part of the document.
- **NodeJS**: The server of a MERN stack is done in NodeJS. It is an asynchronous server based off of Google's V8 engine which was initially built for Google Chrome. It uses a single process, so we do not need to create a new thread for every request, reducing bugs. Whenever a new I/O operation is needed, such as reading the network, accessing the database or file system, instead of blocking the thread, waiting for a new request and wasting CPU cycles, Node.js will resume operations before waiting for the response [16]. In performance evaluations conducted Node.js can handle about 2.5 times as many requests per second compared to that of PHP [17]. In a hashing test, the

This research has been partially supported by the National Natural Science Foundation of China under grant 61750110534, 61672371,61876217, Science & Technology Development Project of Suzhou under grant SYG201817. The authors would acknowledge Christopher Piccaro for his contribution on the experiments of this project.

number of failed requests in an apache server starts to increase around 20 concurrent connections, rising to 7 failed requests by 30 connections, while a Node.js server has 0 failed requests [17]. Node.js is used by many large corporations in 2019 such as Walmart and Paypal.

III. The Proposed RESTful IoT Service

In this section, we will introduce our proposed system, a RESTful IoT service using MERN stack for data gathering and communicating with the Cloud.

A. Software Architecture

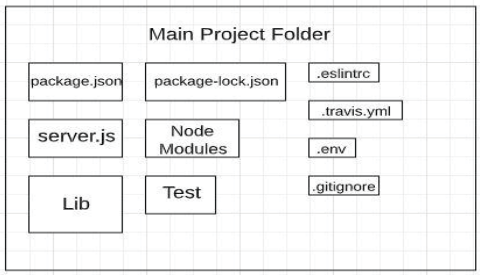

Fig 1. The software system.

The software architecture of the proposed system is shown in Fig 1. The main project folder named *node-IOT-master* includes the folders, *lib*, *node-modules*, and *test*. The files of this system in the main project folder are *.env*, *.eslintrc*, *.gitignore*, *.travis.yml*, *package-lock.json*, *package.json*, and *server.js*. Everything is uploaded to a git repository to track updates on the project except those listed in the .gitignore file. Travis is a deployment tool. It runs its own tests on a docker container to make sure the code is ready to be deployed. The *.env* file sets the environment and has authorization tokens/api keys. The *.eslintrc* is a linter used to make the code cleaner. The *test* file is used to test the models and routes in the *lib* file. *Server.js* runs the app on a certain port. Node modules are modules automatically installed when using *Node.js*. The *package.json* file are dependencies which the project uses. It also contains project properties, descriptions, author & license information, and scripts. *package-lock.json* records the exact version of each installed package.

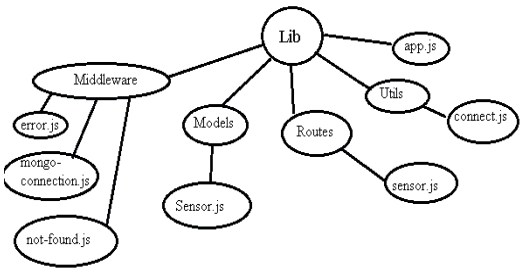

Fig. 2 Architecture of *lib*.

The main folder of the project, *lib* folder, in Fig 2 consists of *middleware*, *models*, *routes*, and *utils* folders as well as an *app.js* file. The *middleware* folder consists of the files *error.js*, *mongo-connection.js*, and *not-found.js*. Middleware is an important part of web applications. It is the code that provides instructions to decide what happens between a request initiated to the server and a response given to the client. The *models* folder consists of a *Sensor.js* file. This is the sensor model which is uploaded to the MongoDB database using a schema definition. The routes folder consists of a *sensors.js* file. This defines the URL routes that is used to contect to the server from the internet browser. The *utils* folder consists of a *connect.js* file. This file contains Uniform Recource Identifiers (URIs) which access an object using an internet protocol. The *app.js* file imports the previous folders and files into one JavaScript file.

B. Detailed Design

In this subsection, we will use sample code to illustration the design of the system with the MERN Stack.

Fig. 3 *error.js*

```
// eslint-disable-next-line no-unused-vars
module.exports = (err, req, res, next) => {
  res.status(err.status || 500).send({ error: err.message || err });
};
```

Shown in Fig. 3 is the *error.js* file, which indicates the error status 500. The HyperText Transfer (HTTP) 500 internal server error response code indicates that the server encountered an unexpected condition that prevented it from fulfilling a request. It is a unique kind of middleware that has 4 parameters compared to all other types of middleware which only have 3. When Express sees 4 parameters it knows the first parameter is an error. When it sees 3 parameters it knows the first parameter is a request, the second parameter is a response, and the last parameter is a higher order function which calls the next middleware.

```
const mongoose = require('mongoose');
const state = require('mongoose/lib/connectionstate');

module.exports = (req, res, next) => {
  const { readyState } = mongoose.connection;
  if(readyState === state.connected || readyState === state.connecting) {
    next();
  } else {
    const err = new Error('Unable to connect to data base');
    err.status = 500;
    next(err);
  }
};

```

Fig. 4 *mongo-connection.js*

As shown in Fig. 4, the *mongo-connection.js* file uses Mongoose to connect to the MongoDBdatabase. If the readyState status is connected or connecting then the middleware waits for the next request. Otherwise it throws an error 500.

```
module.exports = (req, res, next) => {
  const err = new Error('Not found');
  err.status = 404;
  next(err);
};

```

Fig. 5 *notfound.js*

In Fig. 5, an error status 404 is a standard HTTP error response indicating that the server itself was found but that the server was not able to retrieve the requested page. Function next() at the end of each middleware is called to go to the next middleware function. In this case the next middleware function is an error.

- Models

Unlike in SQL, the data modeling in MongoDB has a flexible schema. In SQL you must define a table's schema before inserting data. In MongoDB, the collection of documents can have different schemas [18].

```
const mongoose = require('mongoose');

const sensorSchema = new mongoose.Schema({
address: {
    type: String,
    required: true
},
temperature: {
    type: Number,

},
humidity: {
    type: Number,

},
});
const Sensor = mongoose.model('Sensor', sensorSchema);
module.exports = Sensor;
```

Fig. 6 *sensor.js* in *models* Folder

In Fig. 6 we have the schema for a sensor in MongoDB. This schema requires the sensor to have an address as the data type of a string. If the sensor does not have an address it will not be uploaded to the MongoDB database. This is the basic schema for a sensor. Later we can add more information that a sensor outputs such as humidity, temperature, etc.

- Routes

In Fig. 7, we have the *sensor.js* file for establishing our routes. Here we have a GET and POST command. The POST command creates data for a new sensor onto the server to go to the database. In Fig. 7, on line 13 we are creating our POST command with the sensor model. The GET command retrieves data from the database for a given sensor. Here we get the data from the sensor at address 127.1.2.3.

```
/*routes are lowercase and plural
    models are uppercase and singular
    classes are uppercase in javascript */
const {Router} = require('express');
const Sensor = require('../models/Sensor');

module.exports = Router()
.post('/', async(req, res, next) => {//req =input res = output
    const{
        address
    } = req.body;
    try {
        const createdSensor =await Sensor.create({address});
    res.send(createdSensor);
    } catch (err) {
        next(err);
    }
})

.get('/', async(req,res, next) => {
    res.send([
        {
          address: '127.1.2.3'
        }
    ])
});
```

Fig. 7 *sensor.js* in *routes* Folder

```
const mongoose = require('mongoose');
const { parse } = require('url');

const redact = dbUri => {
  const parsedDbUri = parse(dbUri);
  const authPart = parsedDbUri.auth ? '***:***@' : '';

  return `${parsedDbUri.protocol}//
  ${authPart}${parsedDbUri.hostname}:
  ${parsedDbUri.port}${parsedDbUri.pathname}`;
};

const addEvent = (event, dbUri) => {
  mongoose.connection.on(event, () => {
    // eslint-disable-next-line no-console
    console.log(`MongoDb connection ${event} on ${dbUri}`);
  });
};

module.exports = (dbUri = process.env.MONGODB_URI) => {
  const redactedUri = redact(dbUri);
  ['open', 'error', 'close',
  'disconnected', 'reconnected'].forEach(
    event => addEvent(event, redactedUri));

  return mongoose.connect(dbUri, {
    useCreateIndex: true,
    useFindAndModify: false,
    useNewUrlParser: true
  });
};

```

Fig. 8 *connect.js* in *Utils* folder

- Utils

The first part of Fig 8. on line 4 is the redact function. The redact function parses the username and passcode of the user. If there is no username and password then the *authPart* is an empty string and the URI is the same as it was when it was passed through redact function. The URI is based off of the

.env file. In this file there is only one line of code which is *MONGODB_URI=mongodb://localhost:27017/node-server.* This URI is the address for the mongo database. When there is a connection to the Mongo database it will console log the connection. Whenever an event happens such as 'open', 'error', 'close', 'disconnected', or 'reconnected, this even gets logged to the console. Lines 26 through 30 connects our app to the database with certain settings.

```
1  const express = require('express');
2  const app = express();
3  const cors = require('cors');
4  //https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
5  const mongoConnection =
6  require('./middleware/mongo-connection');
7
8  app.use(cors());
9
10 app.use(require('morgan')('tiny', {
11  // skip: () => process.env.NODE_ENV = 'test'
12 }));
13
14 app.use(express.json());
15
16 app.use('/api/v1/sensors', mongoConnection,
17  require('./routes/sensors'));
18
19 app.use(require('./middleware/not-found'));
20 app.use(require('./middleware/error'));
21
22 module.exports = app;
23
```

Fig. 9 *app.js*

Fig. 9 is our *app.js* file. This is the main file where we import the previous files into one file. We require Express and CORS. Require is *Node.js* way of importing modules. On line 10 we are using Morgan which is a logger that records how long it takes for the server to respond to a request. In line 14 is a body parser. Line 16 and 17 are the routes.

```
1  require('dotenv').config();
2  require('./lib/utils/connect')();
3  const app = require('./lib/app');
4
5  const PORT = process.env.PORT || 8080;
6
7  app.listen(PORT, () => {
8    // eslint-disable-next-line no-console
9    console.log(`listening on ${PORT}`);
10 });
11
```

Fig. 10 *server.js*

Fig. 10 contains the *server.js* file which runs *app.js* (our web application) on port 8080.

IV. EXPERIMENTS

A. Experiment Setup

In this experiment we recorded the time between the request and response of the server using a console logger named Morgan. Express and CORS were installed using NPM packet manager. The project was implemented using VS Code. A 404 response indicates an error that we could not find the page we were looking for. A 304 response indicates the time it took to acquire information about a sensor's data. We repeated the requests 10 times in two different browsers, Chrome and Firefox, and recorded the results of response time in milliseconds.

B. Experiment Results

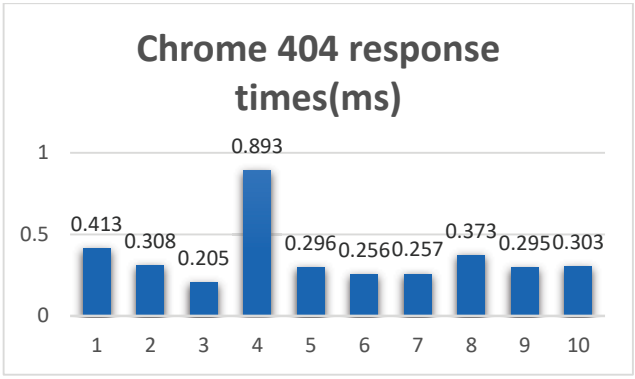

Fig. 11 Chrome 404 repsonse time

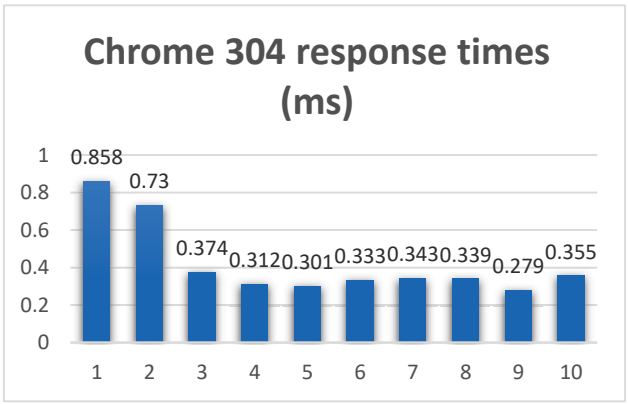

Fig. 12 304 response time

As shown in Fig. 11-14, the average response time for 304 responses using the Firefox browser is 0.89 milliseconds and for 404 responses using the Firefox browser is 0.54 milliseconds. Using Google Chrome, the average response time for 404 responses is 0.36 milliseconds and for 304 responses is 0.42 milliseconds. It should be noted that the average time for the responses is all under 1 millisecond which shows Node.js' speed with its V8 engine. Google's API PageSpeed benchmark for a server response time states it should be under 200ms [19]. Our server exceeds this benchmark by approximately 200times.

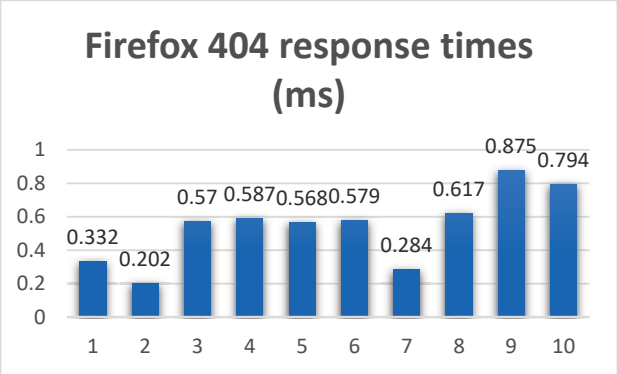

Fig. 13 Firefox 404 response time

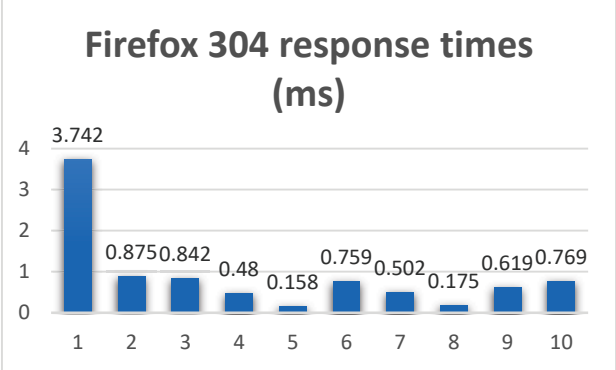

Fig. 14 Firefox 304 response time

V. CONCLUSIONS

In this paper, we study the scalable and efficient data gathering issue in the IoT/Cloud integrated environment. We propose a RESTful IoT service using the MERN Stack, as a gateway to gather data from IoT devices. REST model is known to be able to build web API that is easily connected with database. MERN Stack is used to achieve the smooth flow of the development process. The proposed system is expected to gather data in an efficient and scalable way. The experiments further verified the performance of the development system.

REFERENCES

[1] J. Rittinghouse and J. Ransome, "Cloud Computing: Implementaiton, Management, and Security", *CRC Press, Inc*. 2009.

[2] S. Sivakumar, V. Anuratha, and S. Gunasekaran, "Survey on Integration of Cloud Computing and Internet of Things Using Application Perspective*", International Journal of Emerging Research in Management and Technology*. 6. 101-108. 10.23956/ijermt/SV6N4/101.

[3] I. Hashem, I. Yaqoob, N. Anuar, S. Mokhtar, A. Gani, S. Khan, "The rise of "big data" on cloud computing: Review and open research issues," *Information Systems*, Volume 47, 2015, Pages 98-115.

[4] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," in *IEEE Communications Surveys & Tutorials*, Volume 17, Number 4, pp. 2347-2376, 2015.

[5] S. Li, L. Xu, and S. Zhao, "The Internet of Things: a Survey," *Information Systems Frontiers*, Volume 17, Issue 2, pp. 243-259. April 2015.

[6] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, Volume 3, Number 5, pp. 637-646, Oct. 2016.

[7] M. Satyanarayanan, "The Emergence of Edge Computing," *Computer*, Volume 50, Number 1, pp. 30-39, Jan. 2017.

[8] M. Díaz, C. Martín, B. Rubio, "State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing," *Journal of Network and Computer Applications*, Volume 67, pp. 99-117, 2016.

[9] IETF CoRE Working Group, "Constrained Application Protocol (CoAP)," Intetnet-Draft, ver. Draft-ieft-core-coap-18, 2013.

[10] M. Karagoez and C. Turgut, "Design and Implementation of RESTful Wireless Sensor Network Gateways Using Node.js Framework," *European Wireless 2014; 20th European Wireless Conference*, Barcelona, Spain, 2014, pp. 1-6.

[11] A. Poulter, S. Johnston and S. Cox, "Using the MEAN stack to implement a RESTful service for an Internet of Things application," *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Milan, 2015, pp. 280-285.

[12] https://insights.stackoverflow.com/survey/2019#technology-_-programming-scripting-and-markup-languages

[13] https://www.mongodb.com/who-uses-mongodb

[14] https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

[15] https://expressjs.com/en/resources/companies-using-express.html

[16] https://nodejs.dev/

[17] K. Chaniotis, D. Kyriakou, and N. Tselikas, "Is Node.js a Viable Option for Building Web Applications? A Performance Eveluation Study," *Computing*, 97(10), pp. 1957-1967.

[18] https://docs.mongodb.com/manual/core/data-modeling-introduction

[19] https://developers.google.com/speed/docs/insights/Server