CPSC 597 Graduate Project

Project W.I.N.S.T.O.N

Building an Efficient IoT Pipeline

Project Advisor:

Lidia Morrison

Submitted by:

Andrew Soby

Department of Computer Science and Engineering

California State University Fullerton

Table of Contents

Contents

Table of Contents 1
Abstract 3
Introduction 4
Background and Motivation 4
Related Work 5
Proposed Ideas and Significance 7
Problem Statement 8
Target Audience 8
Users Receiving and Storing IoT Data 9
Users Viewing IoT Data 9
Users Utilizing IoT Data 9
Tools and Development Environment 10
Development Environment Specifics and Tool / Software versions 11
Development Environment Specifics 11
Tool / Software versions 11
Modules and Diagrams 18
Overall Architecture 18
Simulated Device 19
Microservice Interaction 20
Microservice Descriptions 21
Relay Microservice 21
Ingest 21
Normalize 22
Store 22
Alarm 23
Export 23
Query 24
API Access 24
Front End Use Case Diagram 25
Front End Pages 26

Query Page....27
Alarm Page....30
Export Page....32
Implementation and Steps to Run....35
System Run Dependency Flow....35
Prerequisites to Running the System:....36
Running the System Using provided .bat Files....37
Instructions: Commands and Text....38
Instructions: Run System With Pictures....44
Test Runs....52
Future Work....56
Conclusion....57
References....58

Abstract

Internet of Things, or IoT, devices are becoming extremely common and helpful in many industries. Typically, IoT devices communicate to a network and send messages to be ingested in an environment capable of handling the massive amount of data sent from these devices, such as a Cloud Computing environment. Gaining access to this data is one of the biggest challenges these companies face as it is both expensive to ingest and handle this data. This process can be costly and challenging to perform, leading companies to avoid using IoT devices in their business, providing many benefits when used correctly. As more IoT devices continue to connect to these networks and as data grows, it is essential to handle these IoT systems effectively. As IoT evolves, it will be crucial to view these problems coupled with Big Data issues to support these systems most effectively. Handling these IoT devices as a Big Data issue will allow more companies to use IoT devices to benefit their business through efficient systems that promote using, collecting, and analyzing this data.

Keywords: Internet of Things, Big Data, Cloud Computing

Introduction

Internet of Things devices have grown in popularity in recent years and are increasing as more devices connect to these networks and send out sensor data. The data sent out from these devices is continuously growing rapidly due to an increasing number of devices and larger message payload sizes. It is vital that systems can scale and handle the amounts of data ingested into these systems that utilize this data. These devices have become helpful in many different industries ranging from building management to agricultural industries.

As a result of more IoT devices connecting to the internet and the data from these sensors growing, it has created an issue known as Big Data. This issue is where it is expensive to handle and use the data produced from these IoT systems. A requirement of an IoT system should efficiently ingest, store, and transform this increasing amount of data for usage in areas that impact business decisions, such as data analysis. Developing a solution to IoT systems with Big Data in mind will help future proof these systems to create an efficient and scalable pipeline that will aid and help mitigate the issues relating to Big Data.

Background and Motivation

The growing number of devices and message size in IoT systems have caused many difficulties for different companies to process and analyze this data. These expenses are potential factors that can prevent various industries from adopting the use of these devices regardless of the beneficial impact of this data. Many different businesses are not utilizing the benefits of IoT devices and analyzing this data because of expenses that can be solved with effective system design when dealing in these IoT and Cloud Computing environments. Processes such as storing, moving, analyzing, and utilizing this data have become expensive in

these systems and have only worsened. Therefore, it is necessary to design a system efficiently to combat these expenses while also remaining scalable to reduce the costs of requiring a total system redesign as more devices are connecting to a system in the future.

As a motivating factor, these IoT devices continue to grow in popularity and use, so finding a solution to this problem will serve better sooner than later. IoT devices can also benefit and impact various industries, proving these devices' usefulness. High expenses can prevent many companies from investing in IoT technology because of the unique data requirements surrounding these IoT-Big Data systems. Providing a solution to this issue can be impactful, both lessening costs and fundamentally altering these industries for the better. IoT information can positively benefit business decisions by providing details through data analysis. Additionally, a positive impact is promoted with an efficient IoT system that can handle large amounts of data flowing through them.

Related Work

The large number of IoT devices sending data into a system can result in too much data to handle for smaller businesses. An article by Oh et al. states that IoT devices "data are wildly overwhelming in not only its volume, but also its diversity due to the number of Internet of Things (IoT) devices have rapidly increased. It becomes hard to search, discover, process and analyze the proper data from the whole" (pg. 40120). This overwhelming data can negatively influence a business's decision to utilize these devices. It also shows how widespread this issue of big data affecting companies is.

IoT devices produce large data sets, which means that IoT and Big Data are heavily interconnected, proving the importance of solving these issues together to promote an effective

system. By Plageras et al., "The most common type of BD is the IoT-Big Data. It can also be said that IoT and BD are interdependent technologies and should be developed jointly" (pg. 349). The interdependence between these two technologies can weigh heavily on reducing the cost and increasing the effectiveness of these systems. It is crucial to apply Big Data concepts to solve issues resulting from the massive amounts of data sent into these IoT Systems.

While IoT issues overlap with the same issues as Big Data, it is not solely a Big Data issue. Scaling IoT systems also include issues that are involved in Cloud Computing. An article by Li et al. mentions that "Among the many challenges raised by IoT, one is currently getting particular attention: making computing resources easily accessible from the connected objects to process the huge amount of data streaming out of them. Cloud computing has been historically used to enable a wide number of applications" (pg. 668). This Cloud Computing aspect can help benefit the ingest sides of these massive systems; however, it can lead to significant expenses which need to be minimized to allow efficient use of these systems. It is essential to consider different Cloud Computing aspects for both scalability and cost of these systems, which significantly affect efficiency. It also is beneficial to develop a system that can easily be expanded upon in a Cloud Computing environment to utilize the different levels of scalability.

This issue is necessary to solve as the popularity and usefulness of IoT systems are growing. A journal by Metadillou et al. mentions that "use of alternative energy sources, reduction of gas admissions, the contribution of Internet of Things technology to monitor energy consumption and control energy performance is of vital importance" (pg. 63679). These systems are vital in many different industries and use cases, so the benefits obtained from maximizing efficiency in these environments can have vast impacts.

If this prevalent issue in IoT systems is solved, it can significantly improve workers' lives and businesses involved with these systems. Analyzing this data can provide insightful knowledge to both operators and owners of a company allowing benefits to different aspects. A paper by Muangrapthub et al. mentions that "the Internet of Things has begun to play a major role in daily lives, extending our perceptions and ability to modify the environment around us. Particularly the agro-industrial and environmental fields apply IoT in both diagnostics and control" (pg. 467). This statement shows that diagnostics and controls are influenced dramatically by an effective and efficient IoT system.

Proposed Ideas and Significance

Some ideas proposed to make these IoT systems more efficient and scalable involve using Big Data techniques and other standard Cloud Computing techniques when developing these IoT Systems. Utilizing Big Data techniques will make an IoT system as effective as possible to handle the large amounts of data streamed in amongst all the devices. Using Cloud Computing techniques such as horizontally scaling resources as needed will increase efficiency in the IoT pipeline. Additionally, implementing these techniques can reduce cost because performing data manipulation and analysis is lessened in the cloud environment where much of the typical cost occurs, and these systems can be scaled down when required. Exploring both areas of Big Data and Cloud Computing can maximize the potential of these IoT systems.

This issue is significant because it can be a driving factor for companies utilizing these IoT systems or ignoring the valuable data produced from these devices due to complexities and expenses. If the price is too high, it can deter these companies and negatively impact their businesses, causing more costs in the future. However, these IoT systems can benefit industries ranging from agriculture to smart buildings through analyzing and predicting based on

the IoT data to make these industries more efficient. The widespread benefit of these systems can save the environment, time, money, and potentially lives. It is also essential to address this issue early as data is only growing, so proposing better solutions to this issue will be better sooner rather than later.

Problem Statement

The number of different devices that connect to the internet and gather information is increasing rapidly. These large amounts of devices make it difficult to ingest, transform, store, analyze and utilize this information without an efficient and scalable infrastructure to handle massive amounts of IoT devices connecting to systems. This lack of efficient and scalable systems prevents companies from using the data from these IoT devices to their fullest potential.

Target Audience

This system is designed for users who want to receive and store Internet of Things device data, users who wish to view that data, and users that want to use this information. The users can be a wide array of users, from students to companies that want to learn more or gain insight into handling IoT-centric systems more efficiently.

Users Receiving and Storing IoT Data

As previously mentioned, users who want to receive and store Internet of Things data can be a wide range of different types of users. These users can be businesses, individuals running tests, or anyone who wants to receive information and later store it from many devices. This project shows the users the importance of carefully setting up an efficient IoT pipeline to ensure a system can handle IoT information. As mentioned earlier, the number of IoT devices that send data is increasing rapidly; therefore, it is crucial to design a system with this in mind, as this project does. IoT systems require enabling individuals and companies to handle receiving device information now and in the future is essential to use this data for its benefits.

Users Viewing IoT Data

Users who view IoT data can additionally range from many different types of users. It can be a security officer monitoring doors, a data center operator making sure the systems are being cooled, or just an individual that wants to view live data being sent from the system. Viewing this data allows users to see if the system is functioning correctly or if action needs to be taken due to an issue in the areas being monitored by IoT devices. Users can easily view the live information or view recent alerts in this system to be able to make actionable decisions to ensure that the systems that these IoT devices are connected to are working properly

Users Utilizing IoT Data

This system is also created to benefit users who utilize IoT data for analysis and prediction. This system can help data analysts and machine learning specialists because it makes the data readily available to plug into models in the format they desire. This data can be

exported for specific devices for their needs or the entire list of devices if needed. This is useful for these individuals to proactively protect these systems that IoT devices sit on top of to ensure business operations are running smoothly.

Tools and Development Environment

List of Tools / Software

1. Java
2. Gradle
3. Spring Boot / Microservices
4. React / Javascript / HTML / CSS
5. Python
6. IntelliJ IDEA
7. PostgreSQL
8. pgAdmin 4
9. MongoDB
10. MongoDB Compass
11. Apache Kafka
12. Mosquitto / MQTT Broker
13. GitHub / GitHub Desktop
14. Google Docs / Microsoft Word
15. LucidChart
16. Excel

Development Environment Specifics and Tool / Software versions

Development Environment Specifics

The development environment I used was my personal computer to build out this graduate project. My personal computer has a Ryzen 7 3700x CPU, a Radeon 5600XT GPU, 32 GB CL 16 3200 RAM, 1 TB m.2 SSD storage, and an Asus B550-f motherboard. The development was done locally instead of using a cloud provider to save costs. Some testing was later done on a higher-end system with a Ryzen 9 5900x CPU, EVGA 12GB 3080 GPU, and 64 GB CL 16 3600 RAM resulting in faster results due to vertical scaling.

Tool / Software versions

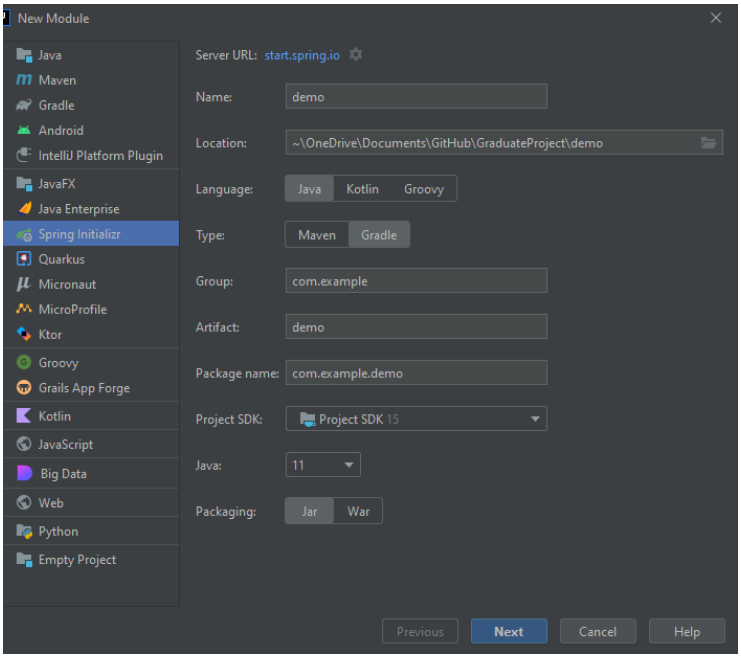
1. **Java** - Java is used as the primary programming language for the microservices in this project. This decision was because the Spring Boot framework is relatively mature and has many viable libraries and dependencies that are easy to use.

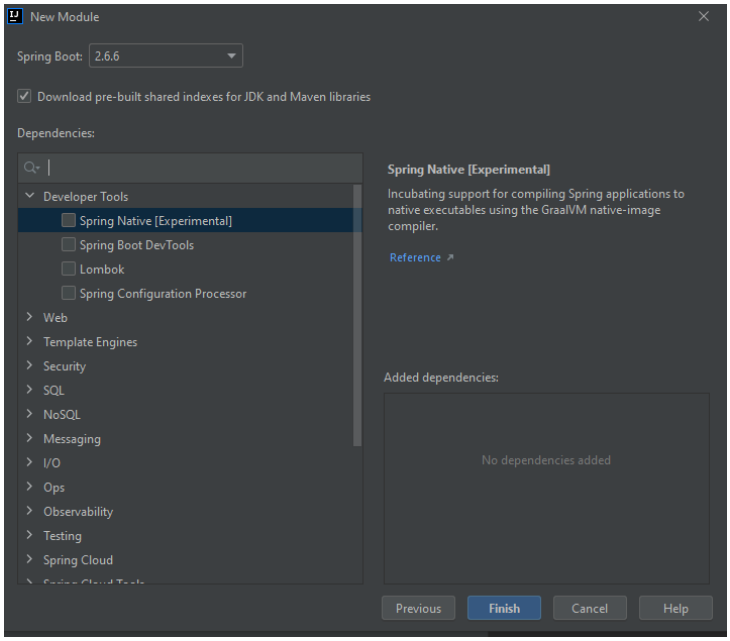
```
C:\Users\sobya\OneDrive\Documents\GitHub\GraduateProject>java -version
openjdk version "15.0.2" 2021-01-19
OpenJDK Runtime Environment AdoptOpenJDK (build 15.0.2+7)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 15.0.2+7, mixed mode, sharing)
```

2. **Gradle** - Gradle is used as the build and run tool of the project as it can efficiently run Spring Boot projects and build them into jars if need be.

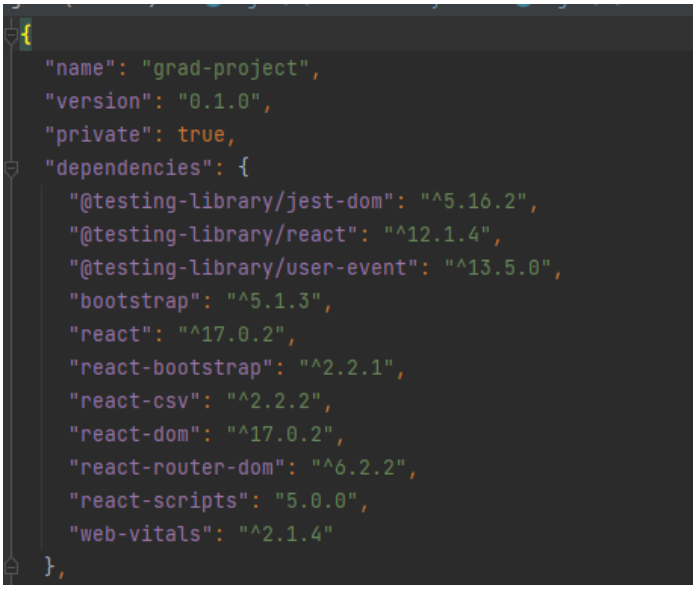
```
C:\Users\sobya\OneDrive\Documents\GitHub\GraduateProject>gradle -v

------------------------------------------------------------
Gradle 7.3.3
------------------------------------------------------------

Build time:   2021-12-22 12:37:54 UTC
Revision:     6f556c80f945dc54b50e0be633da6c62dbe8dc71

Kotlin:       1.5.31
Groovy:       3.0.9
Ant:          Apache Ant(TM) version 1.10.11 compiled on July 10 2021
JVM:          15.0.2 (AdoptOpenJDK 15.0.2+7)
OS:           Windows 10 10.0 amd64
```

3. **Microservice Settings / Spring Boot** - Spring Boot is a java framework with many options and libraries to help speed up the development purpose of the microservices. Below are the general settings I would use, excluding the exact naming structure for each microservices. They utilized Java 11, Gradle as a build tool, and Spring boot 2.6.6. This initializer tool is in IntelliJ and creates projects (for my purpose, microservices).





4. **React / Javascript / HTML / CSS** - React version 17.0.2 was used to create the front end, and all the other dependencies mentioned in the package.json file aided in making the react app. Additionally, HTML, CSS, and Javascript were used for front end development of my project.



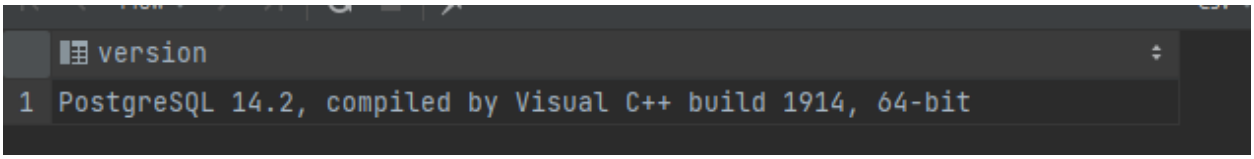
5. **Python** - Python was used to create a script to replicate device behavior that could send thousands of messages to an MQTT broker replicating many IoT devices sending

messages simultaneously. Python version 3.8.5 was used. Python used device information modified from a Kaggle data set located here: https://www.kaggle.com/datasets/edotfs/dht11-temperature-and-humidity-sensor-1-day .

```
C:\Users\sobya\OneDrive\Documents\GitHub\GraduateProject>python --version
Python 3.8.5
```

6. **IntelliJ IDEA**- IntelliJ IDEA was used as an IDE of choice for this project because of the vast majority of java based microservices that I was using. It is efficient and effective for initializing new microservices and plugging in databases to test if the connections and other areas are working quickly. Intellij also was used to develop Python for the simulated device and Javascript / HTML / CSS for my react app.

7. **PostgreSQL** - PostgreSQL is used to store and effectively query data once normalized. PostgreSQL version 14.2 was used



8. **pgAdmin 4**- pgAdmin was used as a GUI to access PostgreSQL easily; this included running test queries and inserting, creating, and deleting tables.

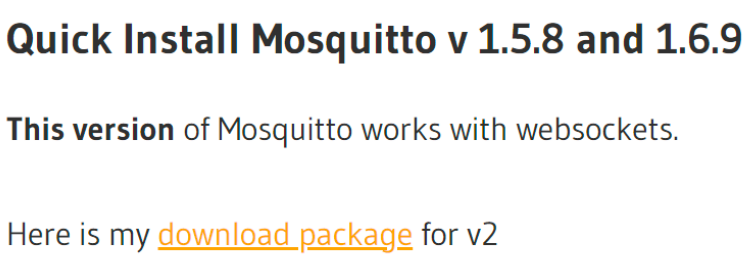
9. **MongoDB** - MongoDB is used as a document store in my project. It stores the data in its entire state so that the unaltered data can be used for processing and analyzing. MongoDB version 5.0.6 was used.

```
C:\Users\sobya\OneDrive\Documents\GitHub\GraduateProject>mongo --version
MongoDB shell version v5.0.6
Build Info: {
    "version": "5.0.6",
    "gitVersion": "212a8dbb47f07427dae194a9c75baec1d81d9259",
    "modules": [],
    "allocator": "tcmalloc",
    "environment": {
        "distmod": "windows",
        "distarch": "x86_64",
        "target_arch": "x86_64"
    }
}
```

10. **MongoDB Compass** - Similar to pgAdmin 4, MongoDB Compass was used as a GUI to access MongoDB to view, store, and update the created collections.

11. **Apache Kafka** - This is used as a message queue that can handle large amounts of messages. I downloaded the binary scala 2.12 version for Kafka 3.1.0 from https://kafka.apache.org/downloads

12. **Mosquitto / MQTT Broker**- I followed http://www.steves-internet-guide.com/install-mosquitto-broker/#manual this guide and just downloaded the binary files for Mosquitto to quickly run them using Mosquitto commands once in the directory. The screenshot below shows the section I downloaded the binary from.

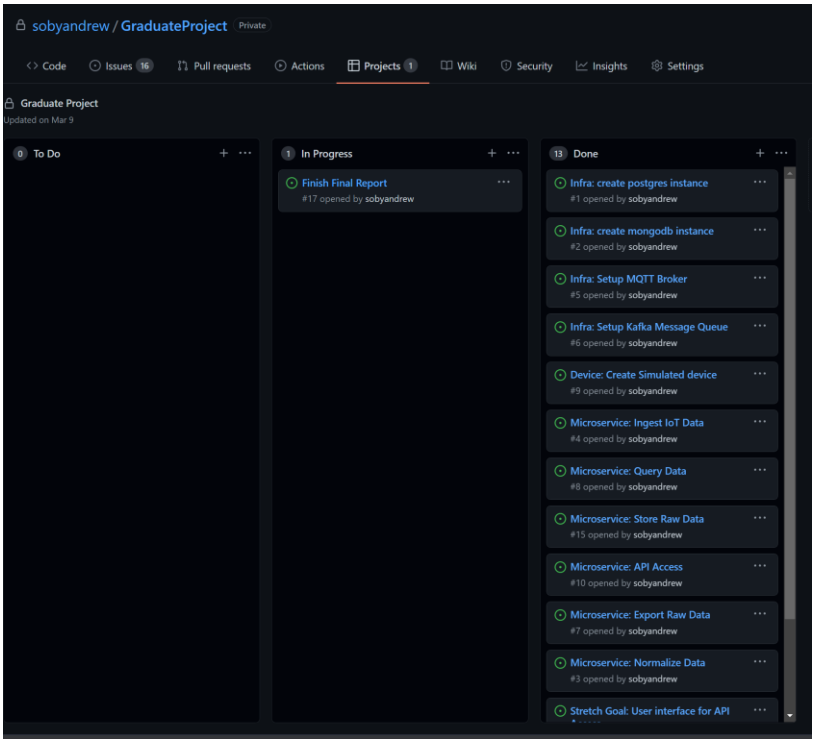


After downloading that package and extracting it, you can navigate to the proper folder and then run it using standard Mosquitto commands to start the MQTT broker.

Additionally, the binary files were included in my project repo, so anyone wanting to run my system can utilize that for ease of running it on a windows system; however one can reach the same results by installing Mosquitto as a service on their system. I am using Mosquitto version 2.0.10, and this can be multiple types of MQTT brokers.

```
C:\Users\sobya\OneDrive\Documents\GitHub\GraduateProject\Infrastructure\MQTTBroker\mos2>mosquitto -h

mosquitto version 2.0.10

mosquitto is an MQTT v5.0/v3.1.1/v3.1 broker.
```

13. **GitHub / GitHub Desktop** - I used Github and Github desktop as my version control and project management. My repo exists with all my source code for the microservices, front end, and simulated device section that I created for this project. Additionally, it contains the Mosquitto binaries I downloaded from the site, as mentioned earlier, to run the broker easily.



14. **Google Docs / Microsoft Word** - I used this to write my final project report and documentation.

15. **LucidChart** - This was used to create most of my charts and diagrams for my project

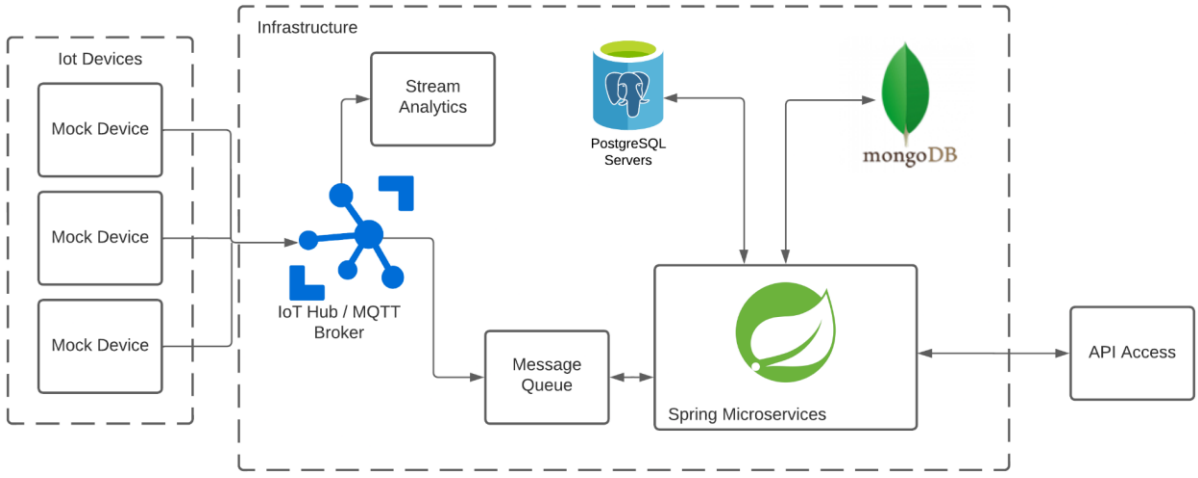
16. **Excel** – I used excel to create the bar graph and plot for displaying my Test run information

*Rest of page left intentionally blank to start Modules and Diagrams section.

Modules and Diagrams

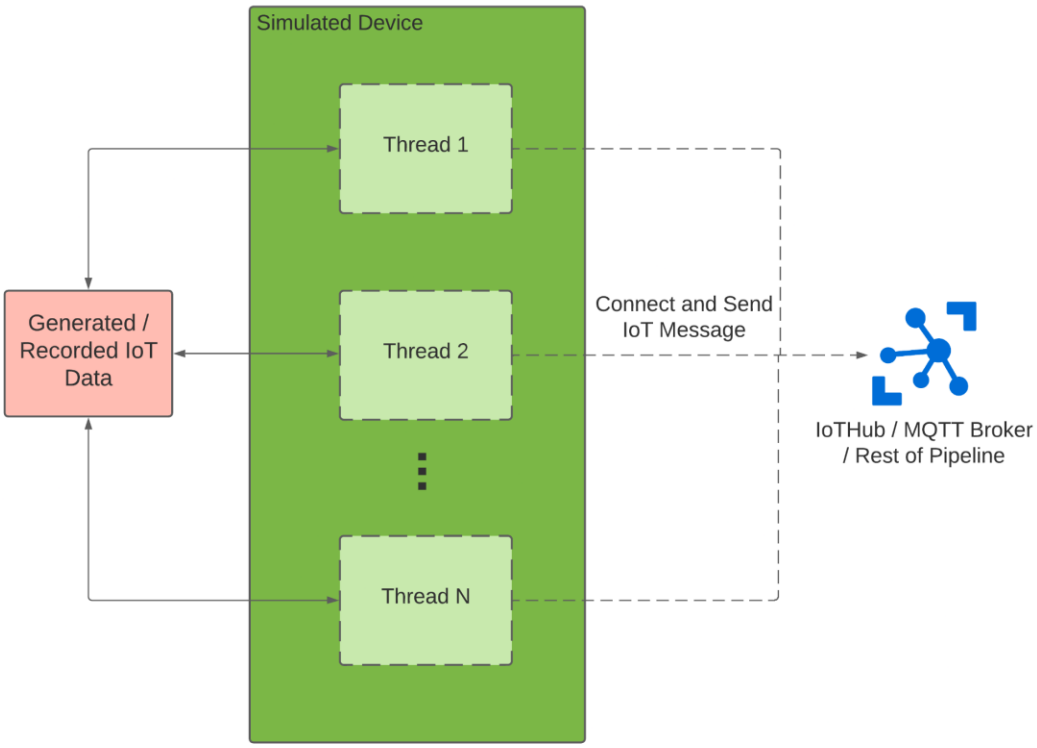
Overall Architecture

This diagram contains the basic structure of the overall architecture of the system. It includes the general ideas corresponding to simulated devices and how it sends messages throughout the system, and the specific technologies used as the infrastructure to propagate the messages to become accessible.



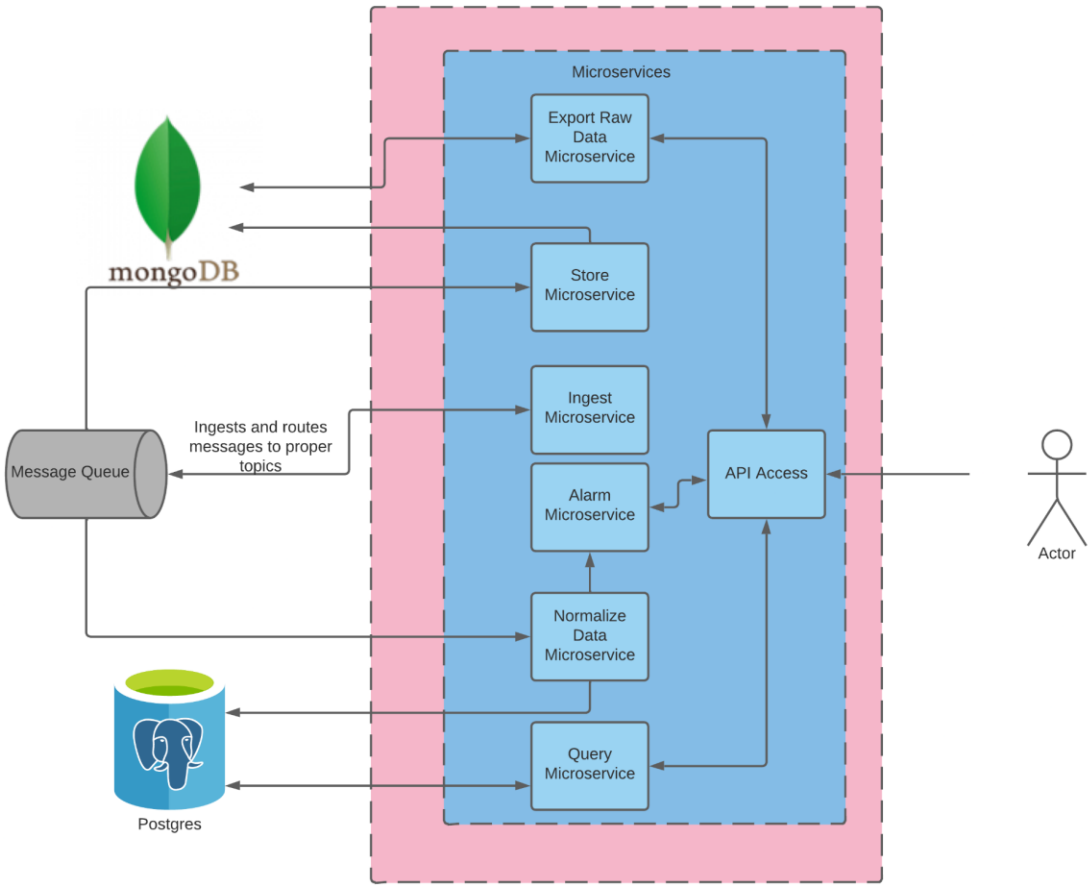
Simulated Device

This diagram contains the information for the simulated device. This Section can act like many IoT devices at once, where each thread can be considered a different device. Additionally, each of these threads will be able to send a specified amount of simulated messages. This section will use previously recorded messages to send messages to replicate an actual device best.



Microservice Interaction

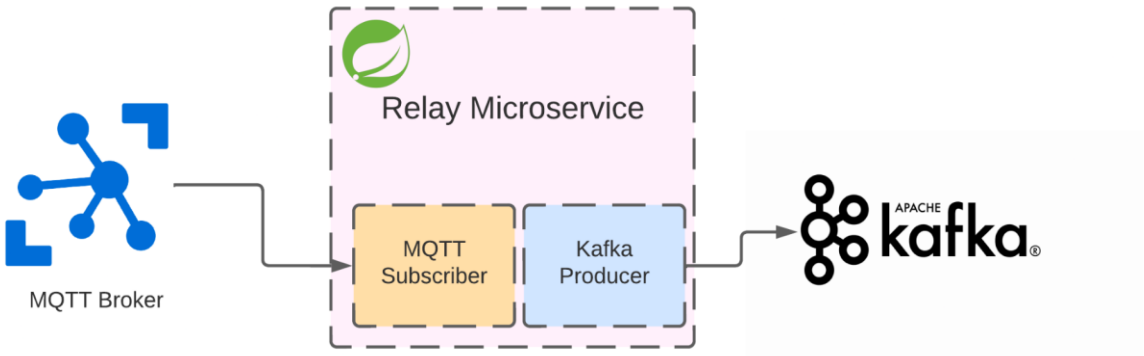
This diagram contains the information for all of the microservices required to be used in the system for base functionality. Messages will come from the message queue and be ingested in the ingest microservice. This service will forward the message onto the correct message topics so that the other microservices can handle the messages. The store and normalize microservice will receive these messages and perform any necessary transformations to store in MongoDB and the PostgreSQL instance, respectively. The export service will gather the requested information that can be exported for the user, and the query service will query the PostgreSQL normalized data. The alarm service receives, stores, and displays alarms that previously occurred. The API access microservice provides a method for users to access these services.



Microservice Descriptions

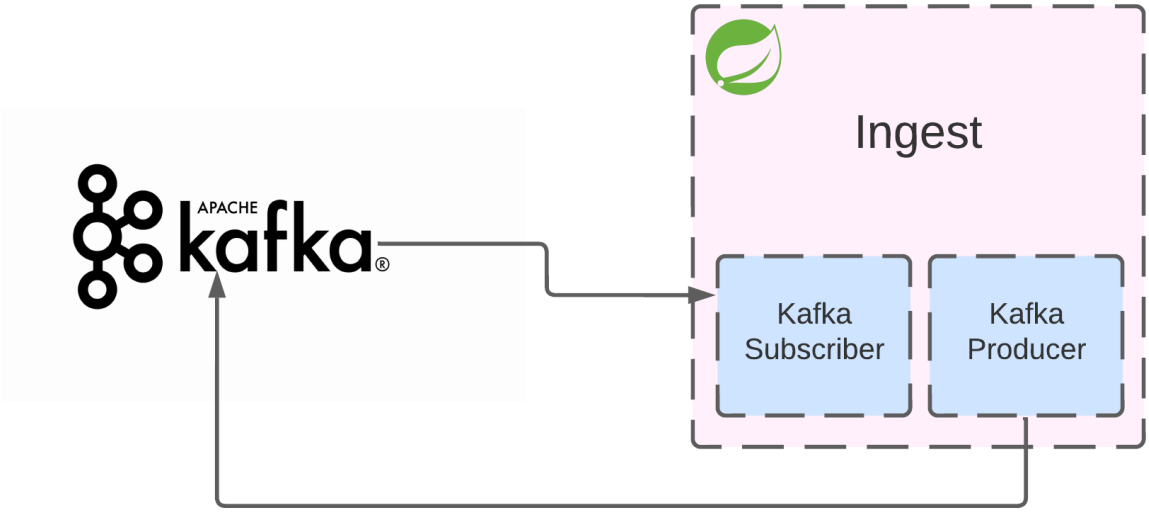
Relay Microservice

This microservice serves to subscribe to the MQTT Broker and place messages onto Kafka for the ingest service to utilize.



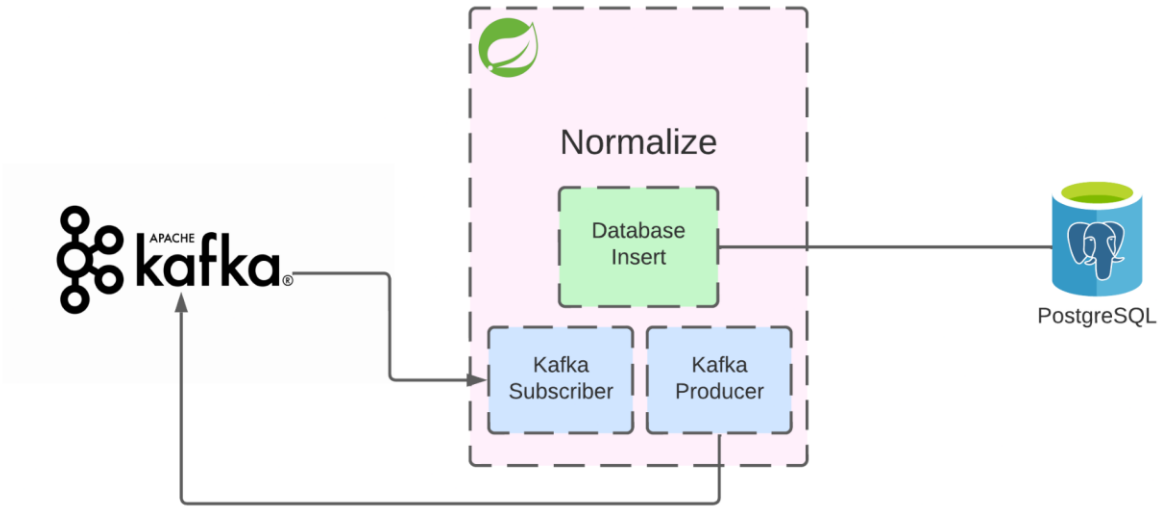
Ingest

The ingest service subscribes to Kafka to receive messages routed from the relay microservice and then produces messages to Kafka for both the normalize and store service.



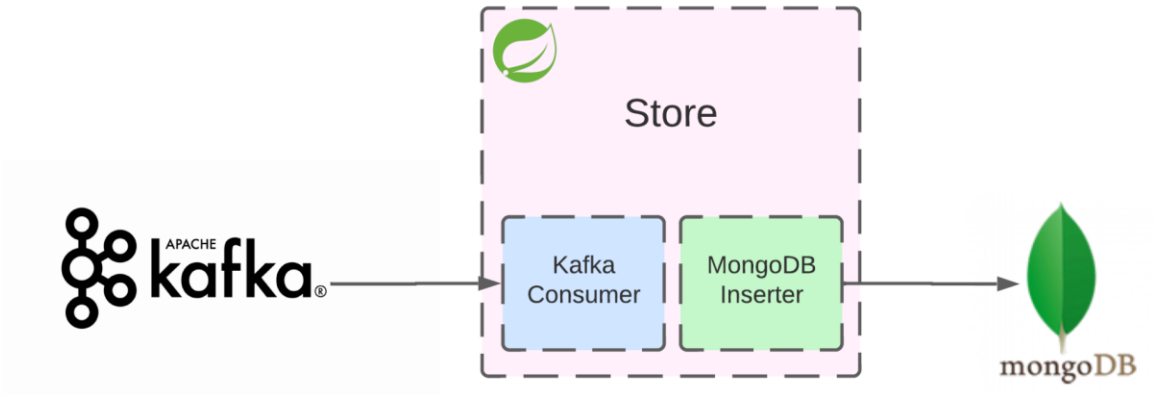
Normalize

The normalize service subscribes to Kafka messages and then batch inserts messages into PostgreSQL. It also evaluates the messages, and once a threshold has been passed, it produces the message to Kafka for the alarm service to ingest.



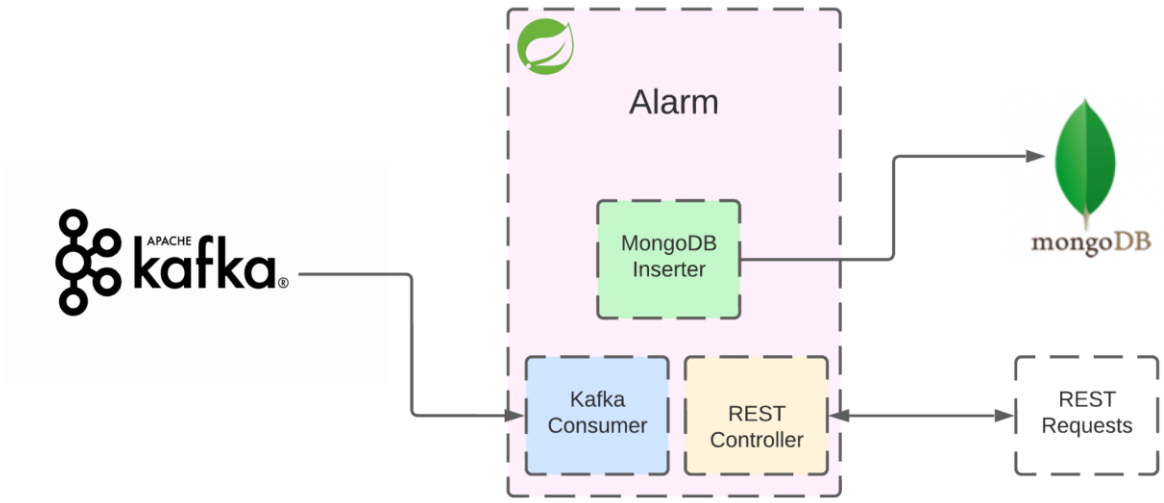
Store

The store service consumes Kafka messages from ingest and stores the data into MongoDB as a document store of these messages without changing or normalizing their content. This service is to preserve the data in its original unaltered format.



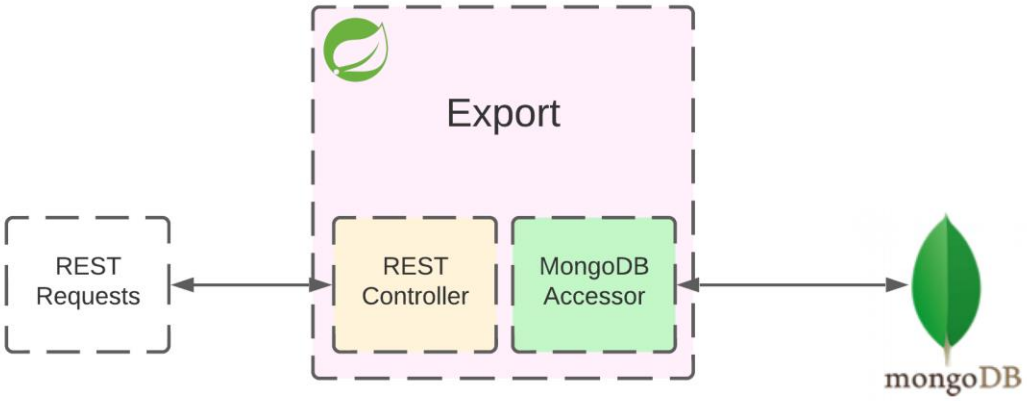
Alarm

The alarm service consumes Kafka messages from the Normalize service and then inserts alarms into MongoDB to be stored and maintained. The alarm service also has a REST controller to display the last ten alarms.



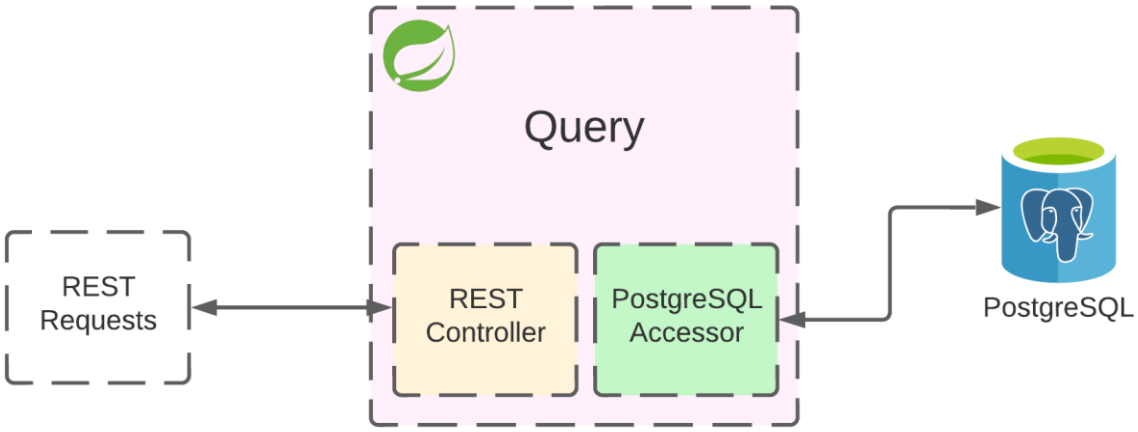
Export

The export service has a REST controller that can accept device export requests and has access to MongoDB to gather the device information.



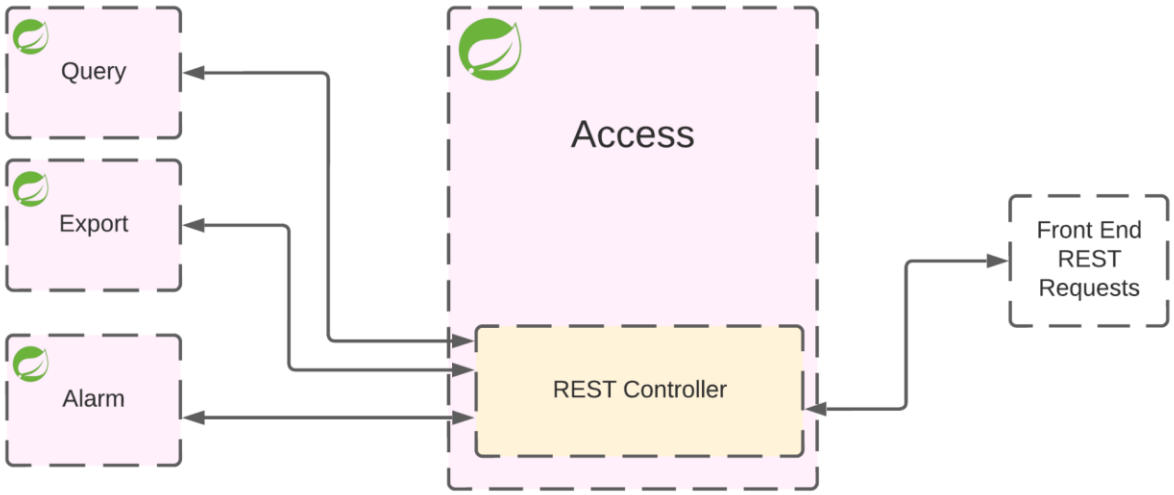
Query

The query service has a REST controller that takes query requests and has a PostgreSQL accessor to gather the requested query results.



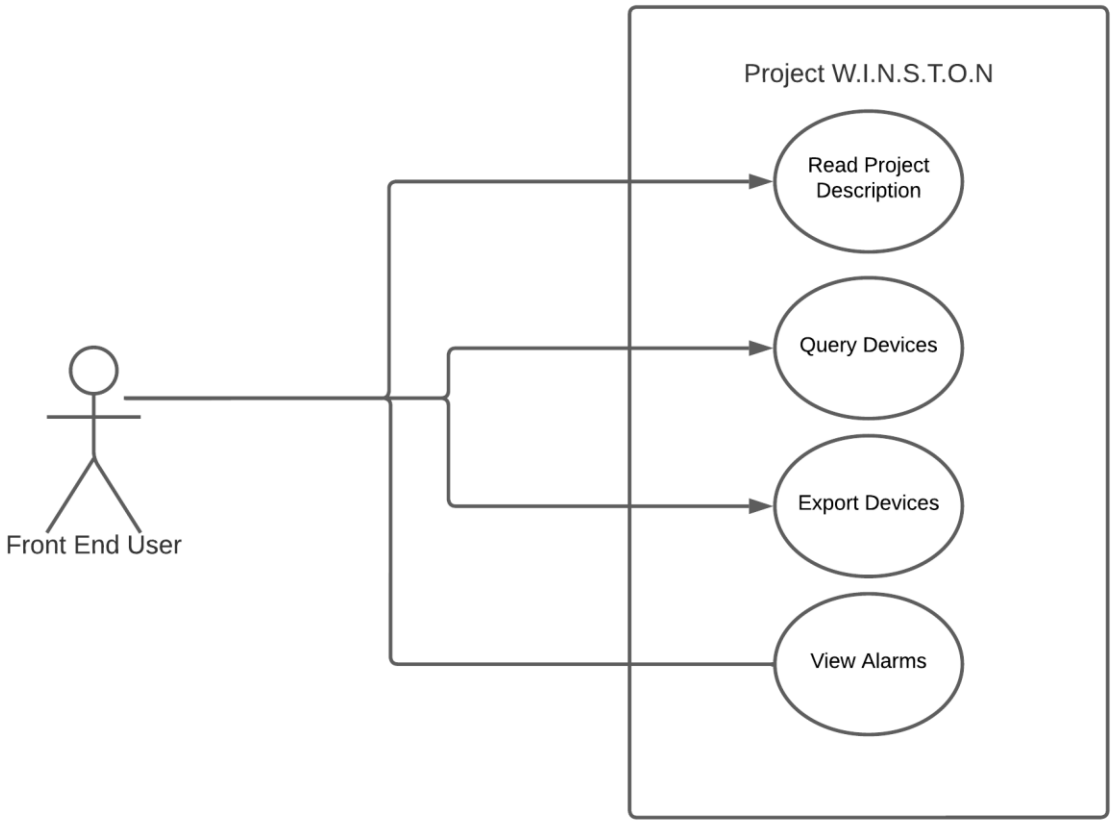
API Access

The API access service acts as a dispatcher for requests from a unified location for the front end to use the query, export, or alarm service easily.



Front End Use Case Diagram

This diagram illustrates the options that a user can view on the front end, including the home, query, export, and alarms pages.

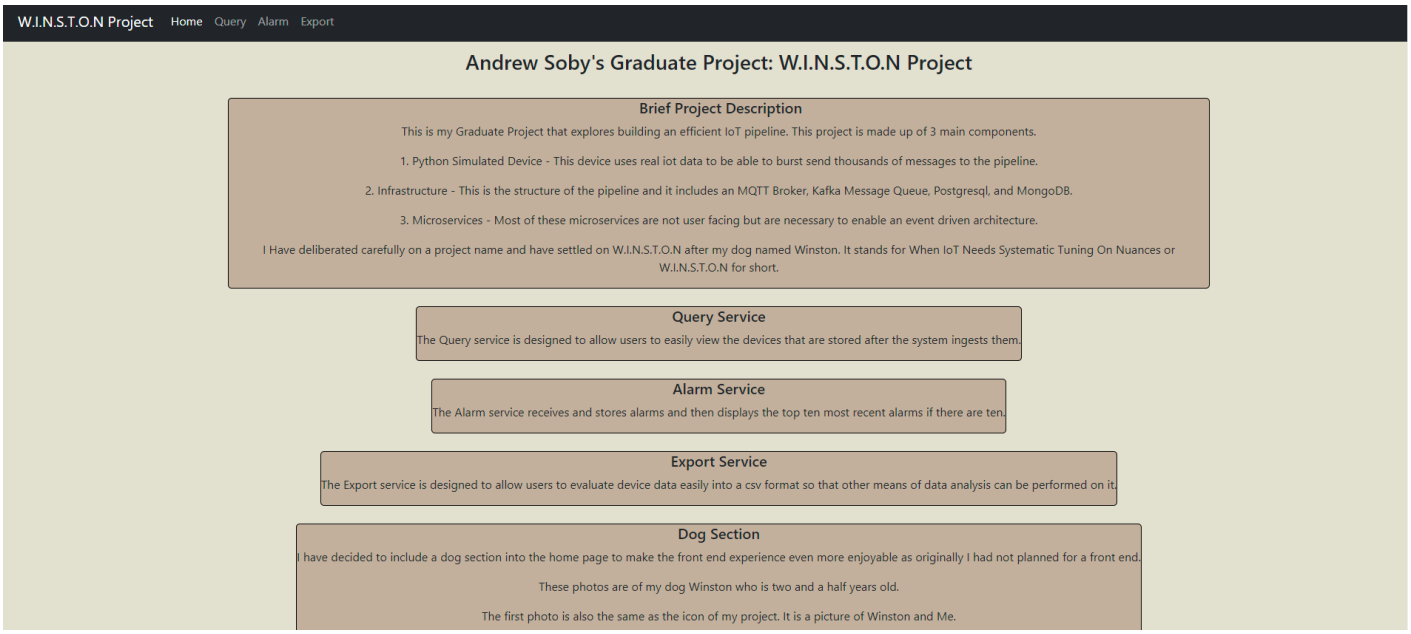


Front End Pages

The Front end of this project was built using React to quickly stand up a more friendly user interface than simple API documentation to display the capabilities of the microservices. There are four separate pages for this project; home, query, alarm, and export. While the home page does not interact directly with any microservices, the three other pages directly interact with API access and their respective microservice.

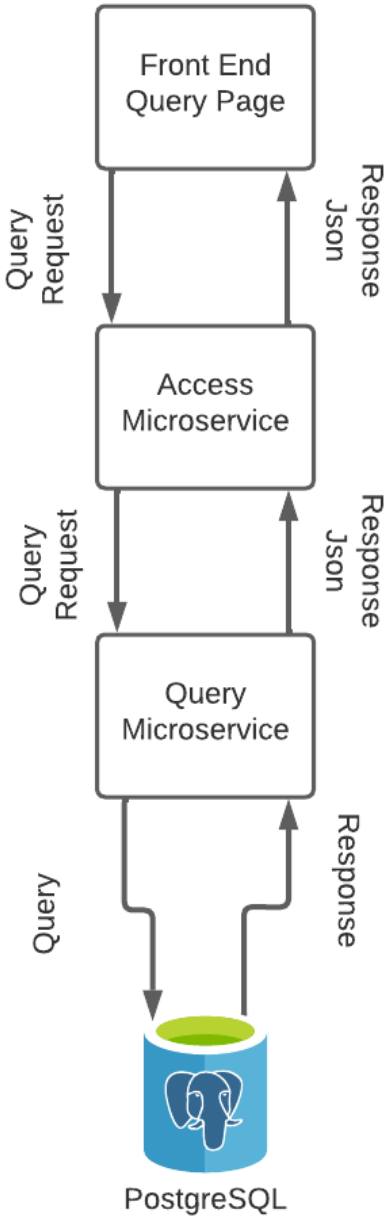
Home Page

The screenshot below is the home page of my Graduate project. It contains a brief description of the project and the other pages that can be accessed on the front end, including their functionality. Additionally, it includes information about the motivation for naming my project and dog pictures.



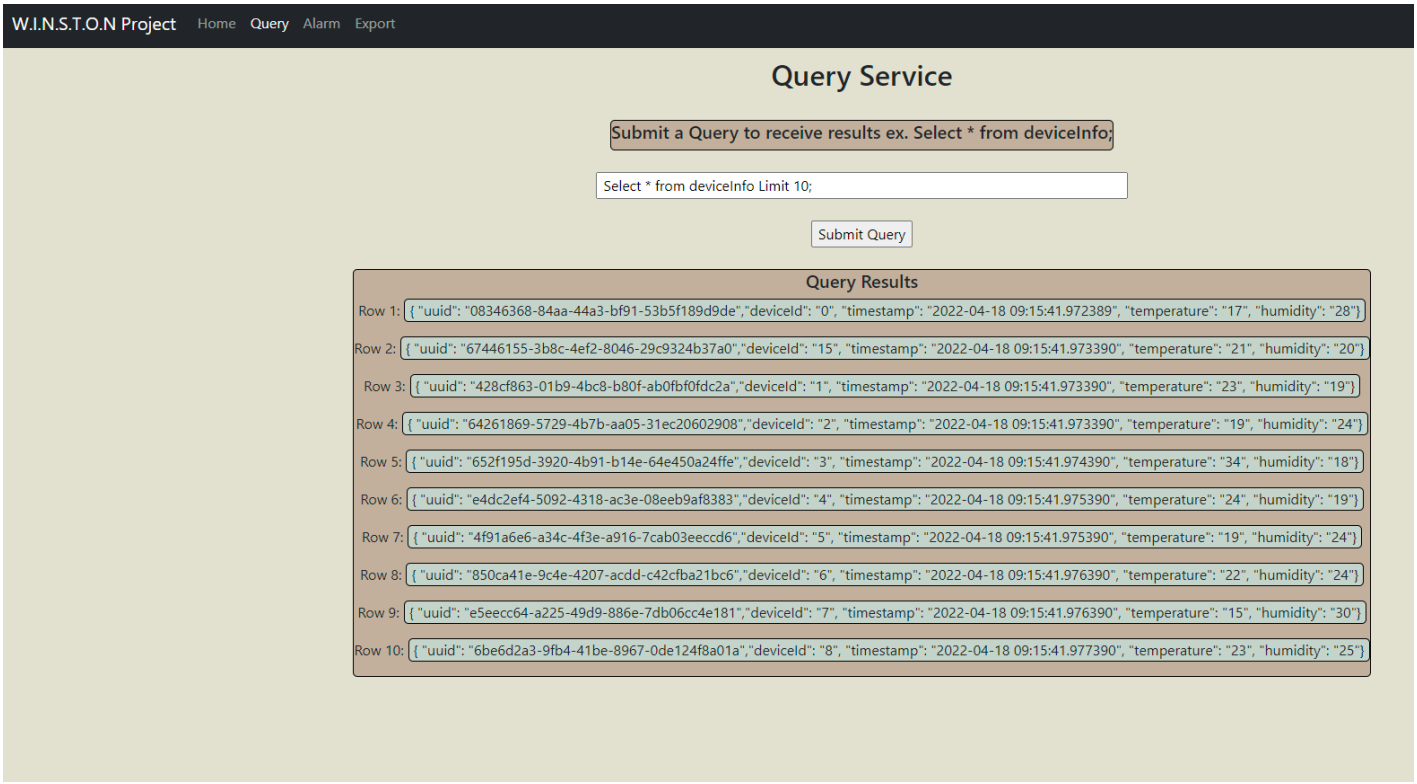
Query Page

The query page is the first page that interacts with the microservices in the back end. First, it sends its query request via a JSON payload to the access microservice. Then this is dispatched to the query microservice that performs the query in PostgreSQL and returns the response to the front end.

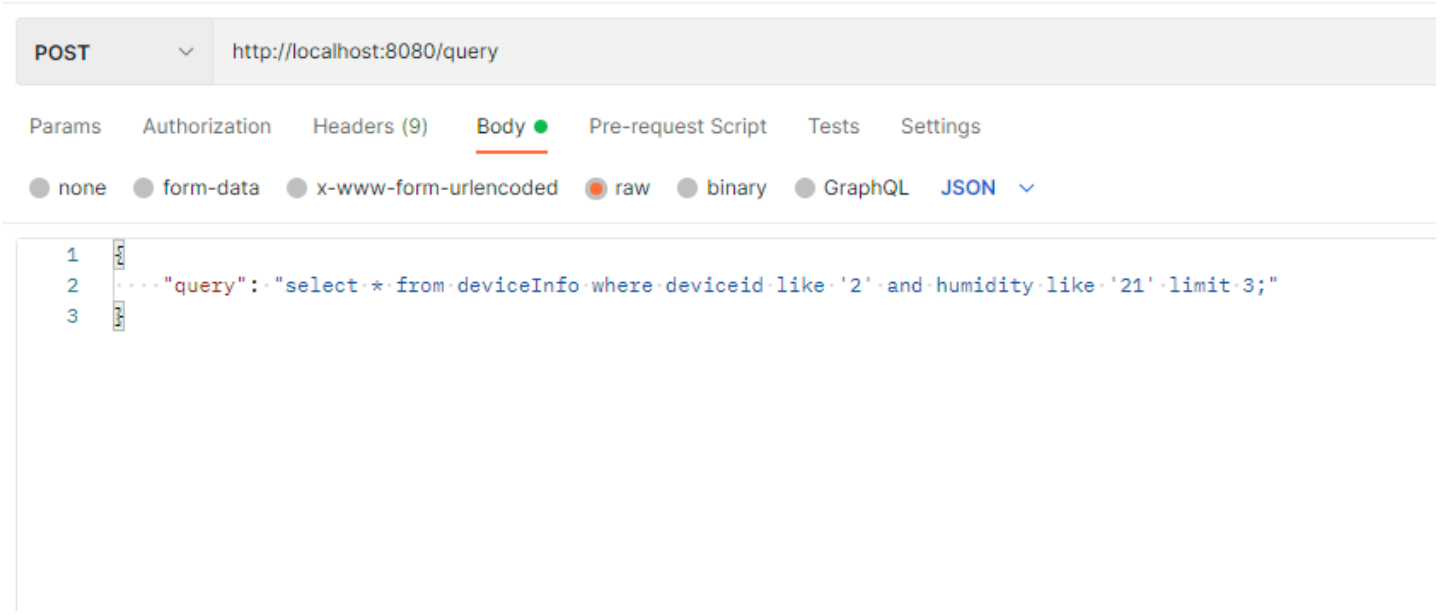


The following screenshot is the query page before a query has been submitted.

The following screenshot is the result of the query page after a query has been submitted. It contains the output returned by the query.



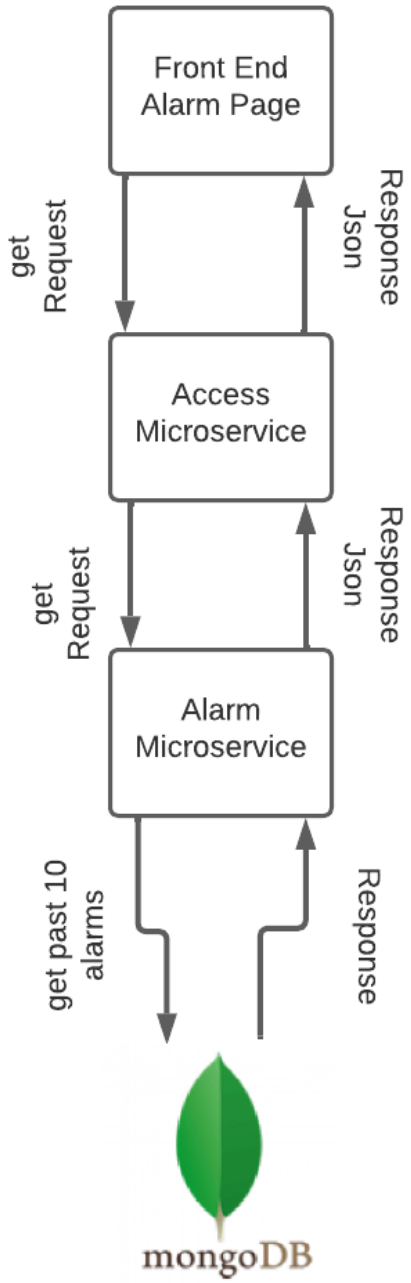
The following screenshot contains an example of the body sent to the backend that handles the query



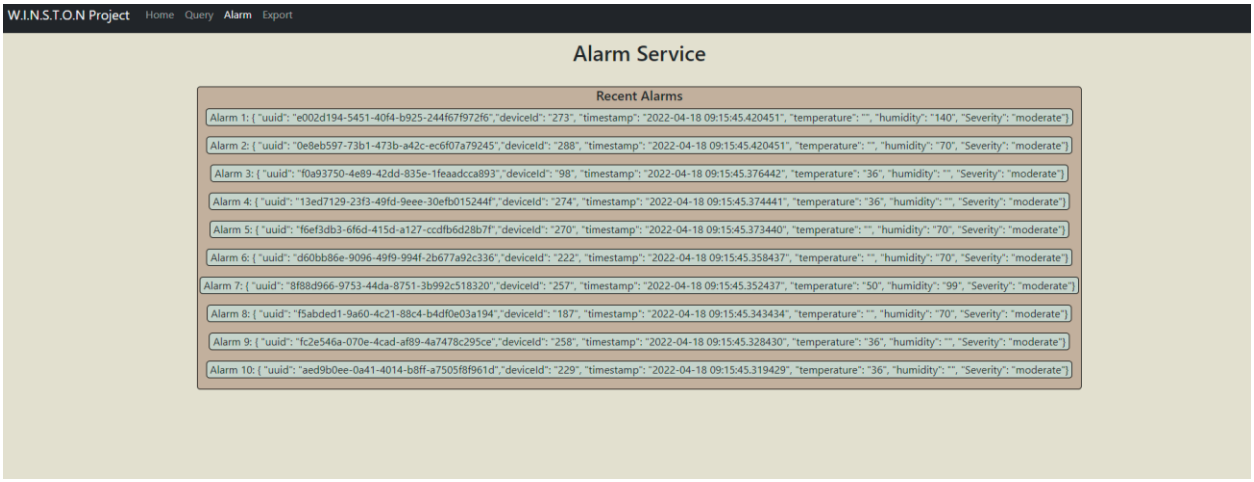
* Rest of page left blank to move on to the next front end page.

Alarm Page

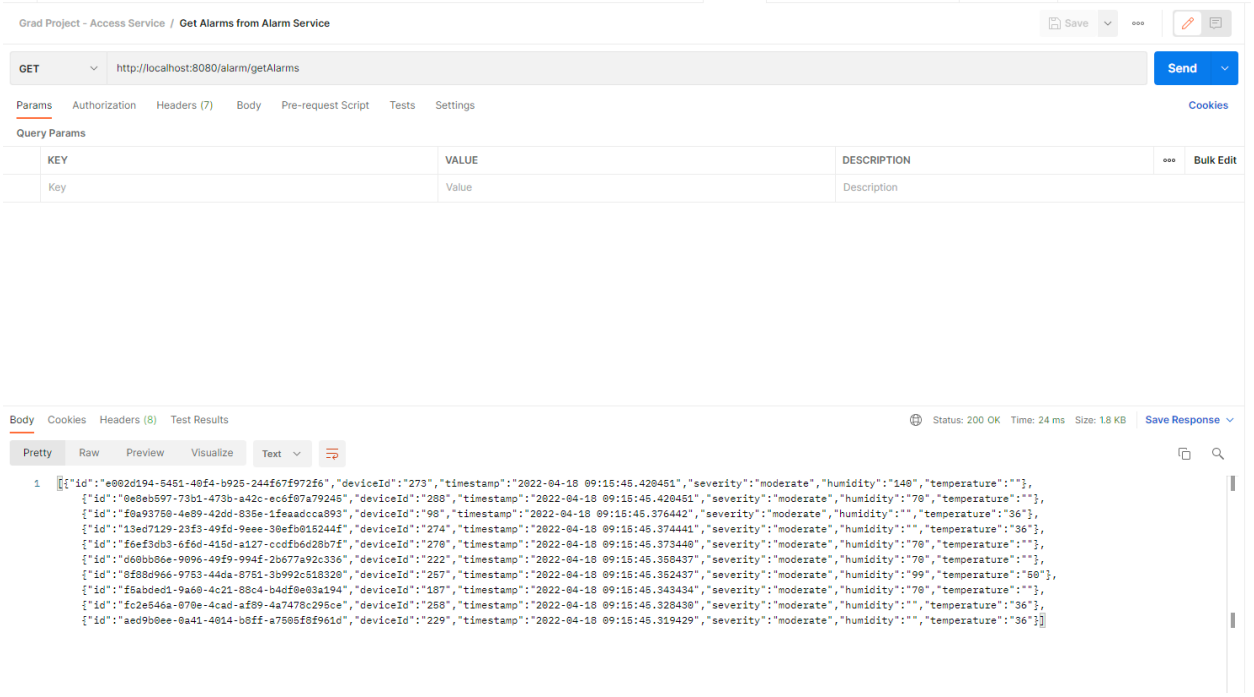
The Alarm page requests the last ten alarms from within the system. It first requests the data through the access microservice, then forwards this request to the alarm microservice and returns the previous ten alarms from MongoDB to the front end.



The Following screenshot contains the image of the alarm page showing the ten most recent alarms.

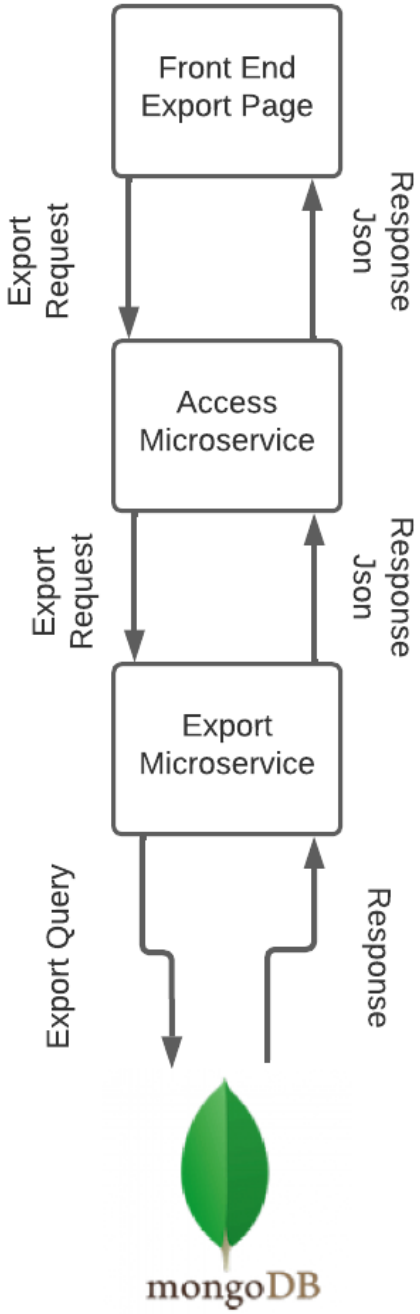


This image displays the GET request sent to the backend for the ten most recent alarms and its corresponding JSON response.

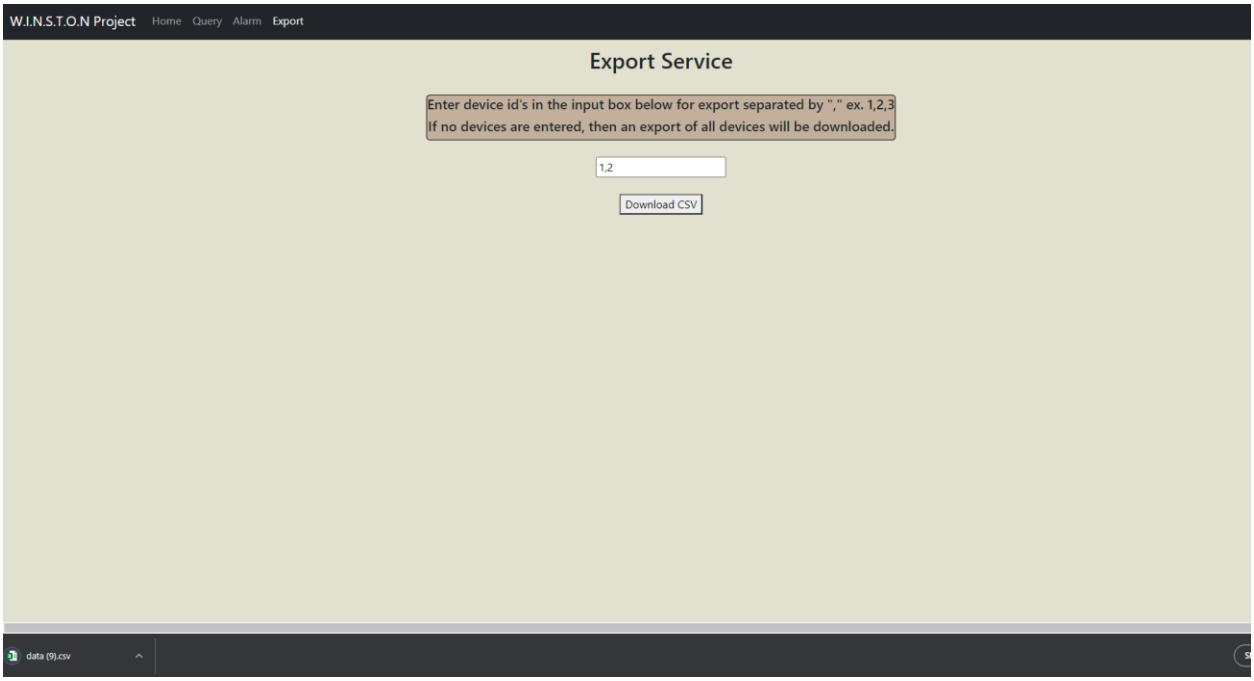


Export Page

The export page works by forwarding its request to the access microservice, propagating to the export service. Based on the request, it will query the data from MongoDB and then return the JSON response to the access service and then the front end.



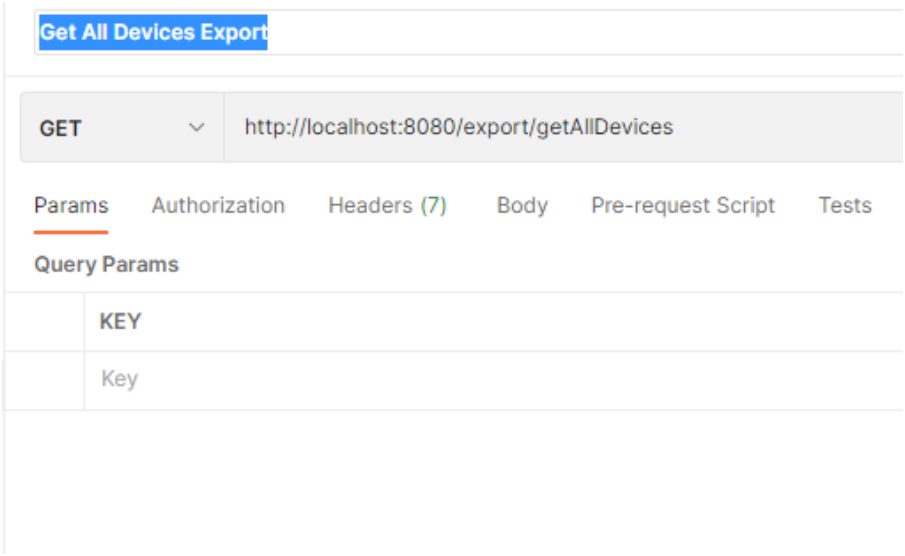
The following is an example of the export page, exporting the information for a device with id 1 and 2



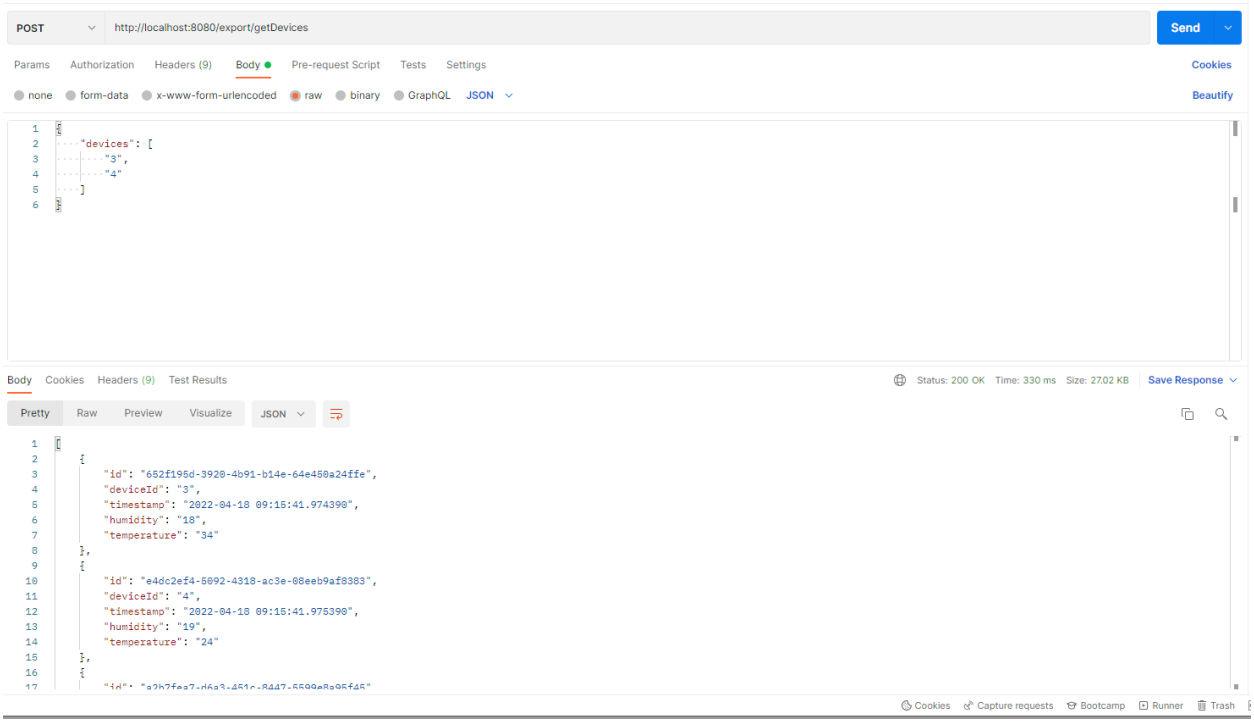
Pressing download CSV then downloads a CSV for the user that looks similar to the following.

	A	B	C	D	E
1	id	deviceId	timestam	humidity	temperature
2	428cf863-(	1	15:42.0	19	23
3	64261869-	2	15:42.0	24	19
4	3d648ffd-4	1	15:42.0	21	20
5	5d719cae-	2	15:42.0	20	21
6	e2aa2d5e-	1	15:42.0	14	34
7	5f081d25-	2	15:42.0	16	30
8	0094efc2-1	2	15:42.0	22	23
9	3d70169d-	1	15:42.0	19	23
10	2bbc10dd-	1	15:42.0	26	22
11	985428a9-	1	15:42.0	23	16
12	49ba3bef-	2	15:42.1	22	18
13	2f6ca519-1	1	15:42.1	23	16
14	d0f61958-	2	15:42.1	23	16
15	16bf9913-	1	15:42.1	21	19
16	95aea655-	2	15:42.1	21	23
17	021948ed-	1	15:42.1	20	22

The following screenshot is an example of a request exporting all the devices.



The following screenshot is an example of the request to export specified devices and their response.

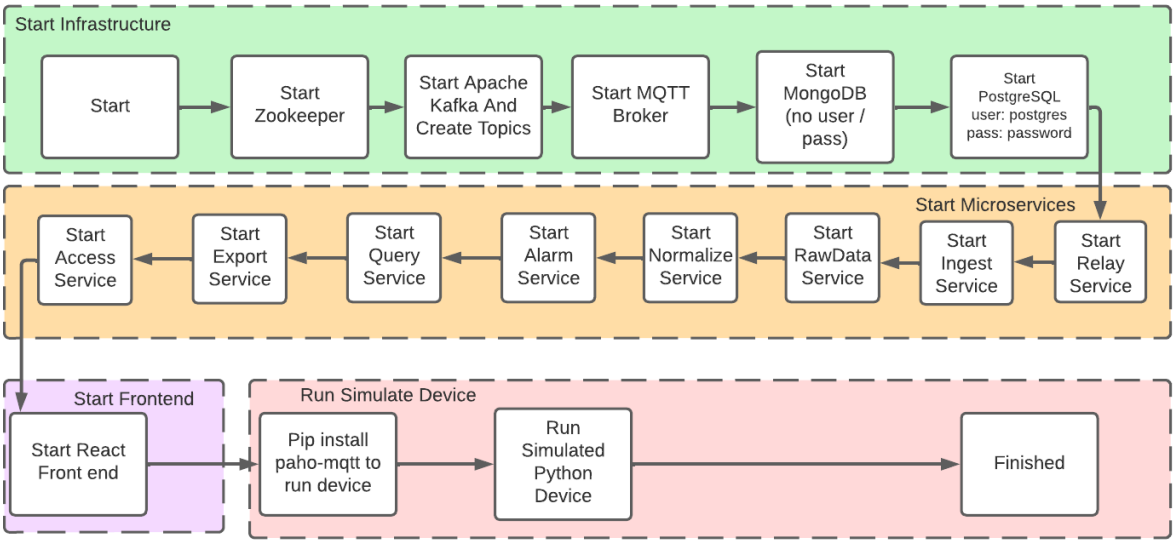


Implementation and Steps to Run

This section is designed to show the implementation of an efficient IoT pipeline, starting from showing the different areas required to have my total system running. It has been split into two separate sections, one with commands and text-only, then one with commands and pictures. This was a design choice for the commands to be easily accessible and decluttered if no issues arise when running the text-only version. The Section containing images was created to add additional help to show expected output if running was not as straightforward or problems arose.

System Run Dependency Flow

This diagram displays the dependency flow to run the system and visualizes the proceeding steps to ensure the system is up and running.



Prerequisites to Running the System:

1. Must be running on Windows 10 64 bit system
2. Must download graduate project repo or be given a zip of the repo
3. Have Java / Python Gradle / MongoDB / PostgreSQL Server / NodeJS installed
    a. Also, run **pip install paho-mqtt**
    b. This step installs the necessary library for the simulated device
4. Start MongoDB and PostgreSQL Server
    a. For this project, MongoDB doesn't require a username or password. If your system requires a username or password to insert and create collections, some steps will need to change later.
    b. For this project, a PostgreSQL user is created with the ability to create, insert, and query from tables. The username for my project is: postgres, and the password is: password.
        i. This is chosen for simplicity, but if your system does not have this same user, then steps later will need to be updated to reflect this
5. Many commands require changing directories using the cd commands; these should be used as examples, and the specific command will depend on where the root of this project is stored on your system.

Running the System Using provided .bat Files

This section is the easiest method to run this project as long as all the prerequisite tasks have been completed. If you run the system successfully using the four batch files, you can skip the Instructions: Commands and Text section and the Instructions: Run System with Pictures section.

a. To begin this section, all you need to do is ensure the prerequisites are set up, and the Kafka folder is located at C:\kafka

1. Run runKafka.bat - this starts zookeeper and kafka
    b. **double click the runKafka.bat** in file explorer
    c. this step requires the previous Kafka folder to be located at C:\kafka
    d. two command prompts should be running after about 20 seconds of Zookeeper first, then Kafka
    e. leave these 2 terminal windows up and running for the remainder of the session
    f. if Kafka does not start, delete the Kafka log folder and re-try again set in server.properties
2. Run createTopics.bat **ONLY NEEDS TO BE DONE IF TOPICS DON'T ALREADY EXIST**
    a. **double click the createTopics.bat** in file explorer
    b. this step will cause an error if the topics already exist, but nothing negative happens affecting other steps
    c. these four terminals that are opened can be closed
3. Run runProj.bat - this starts an MQTT Broker, then the eight microservices, then the web front end, and lastly, runs the simulated device once
    a. **double click the runProj.bat** in file explorer

   b. this step will require PostgreSQL and MongoDB to be running already, as mentioned earlier, as well as pip install paho-mqtt to be done
   c. this step opens ten more terminal windows, and all must be kept open for the remainder of testing for a total of 12 open
4. Run runDevice.bat - this is an optional step. This step runs the simulated device again
   a. if step 3 was run successfully, then this does not need to be done but can be to send more messages
   b. **double click the runDevice.bat** in file explorer

You can skip the following two sections if the project has successfully run. These are just there to show the commands and pictures to run the system without using the batch files.

Instructions: Commands and Text

1. Run Zookeeper
   a. Must change your zookeeper.properties dataDir to your directory on your system
      i. Open the **GraduateProject\Infrastructure\kafka\config\zookeeper.properties** file where GraduateProject is the root folder of this project.
      ii. **Change the line of the dataDir variable** on line 19 to match your systems directory structure and save the file as shown below. This is an example of a windows system where the file project exists on my system.
      iii. `dataDir=c:/users/sobya/OneDrive/Documents/GitHub/GraduateProject/Infrastructure/kafka/zookeeper-data`

   b. Must change your server.properties logs.dir to your directory on your system

i. Open the **GraduateProject\Infrastructure\kafka\config\server.properties** file where GraduateProject is the root folder of this project.

ii. **Change the line of the logs.dir variable** on line 62 to match your systems directory structure and save the file as shown below.

iii. `log.dirs=c:/users/sobya/OneDrive/Documents/GitHub/GraduateProject/Infrastructure/kafka/kafka-logs`

c. **IMPORTANT YOU MUST COPY THE WHOLE KAFKA FOLDER CLOSE TO THE ROOT OF YOUR SYSTEM. FOR ME, I PLACED IT AT C:\kafka**

i. If this step is not followed, you will get an error saying the input line is too long, and you will not be able to start up Zookeeper

ii. This is a known error with Apache Kafka and must be done before attempting to start Zookeeper or Kafka.

iii. I placed the Kafka directory inside the infrastructure into my c drive's root on windows 10.

iv. To move the directory via the command prompt terminal, you can do the following command. Alternatively, you can use file explorer to drag and drop the directory into the specified location

v. **move <your systems path to the root of my project>\GraduateProject\Infrastructure\Kafka C:**

d. Open a command prompt and change the directory into the newly placed Infrastructure/kafka folder close to your root

i. ex: **cd ..\..\kafka**

e. In the same command prompt, start Zookeeper with the following command

f. **ex: .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties**

g. **Leave this command prompt terminal open and running**

2. Run Kafka
    a. Requires Zookeeper to be up and running and requires the server.properties file to be set. If you did step 1b then this is already done.
    b. Open a command prompt and change the directory into the Kafka folder, which is close to the root of your drive
        i. **ex: cd ..\..\kafka**
    c. In the same command prompt, start Kafka with the following command
        i. **ex: .\bin\windows\kafka-server-start.bat .\config\server.properties**
    d. **Leave this command prompt terminal open and running**
3. If there are any errors when running Kafka / Zookeeper, try to delete the log.dirs directory and restart again.
4. Create four message topics on Kafka
    a. Open a new command prompt terminal and change directory into the kafka folder
        i. **ex: cd ..\..\kafka**
    b. using the following commands create topics for each microservice
        i. **.\bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic ingest**
        ii. **.\bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic raw**
        iii. **.\bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic normal**
        iv. **.\bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic alarm**
        v. These topics should already exist but just doing this step incase they don’t on another system.

5. Start MQTT Broker
    a. Open new command prompt terminal and cd into Infrastructure\MQTTBroker\mos2 directory at the root of the graduate project folder
        i. **ex: cd OneDrive\Documents\GitHub\GraduateProject\Infrastructure\MQTTBroker\mos2**
    b. Run Mosquitto in verbose mode to ensure messages are sending with the verbose command; however, this can lead to dropped messages. After running it in verbose mode to ensure it is running, you can stop that instance and run MQTT without the -v flag.
        i. **ex: mosquitto -p 443 -v**
        ii. **ex: mosquitto -p 443**
            1. **Running without the -v flag allows for the MQTT broker to run more efficiently -v means verbose and prints whenever a message is sent, or the client connects. This results in messages dropped on this low resource broker because > 15,000 messages can be sent in a second with many simultaneous connections resulting in many print statements.**
    c. **Keep this command prompt terminal open and running**
6. Ensure MongoDB and PostgreSQL Server are up and running
7. Run All microservices; this can be done in many different ways
    a. Navigate to the build-jar folder and run java -jar for each microservice jar by changing the directory to build-jars. **ex: cd build-jars** from the root directory of the graduate project.

i. **Java -jar access-1.0-Final.jar**
ii. **Java -jar alarm-1.0-Final.jar**
iii. **Java -jar export-1.0-Final.jar**
iv. **Java -jar ingest-1.0-Final.jar**
v. **Java -jar normalize-1.0-Final.jar**
vi. **Java -jar query-1.0-Final.jar**
vii. **Java -jar rawdata-1.0-Final.jar**
viii. **Java -jar relay-1.0-Final.jar**

b. **If you changed the password or username for MongoDB or PostgreSQL, you must run using different java commands, and they are listed below; change what is in the angle bracket.**

i. **Java -jar access-1.0-Final.jar**
ii. **Java -jar alarm-1.0-Final.jar –**
spring.datasource.username=<username> –
spring.datasource.password=<password>
iii. **Java -jar export-1.0-Final.jar –**
spring.datasource.username=<username> –
spring.datasource.password=<password>
iv. **Java -jar ingest-1.0-Final.jar**
v. **Java -jar normalize-1.0-Final.jar –**
spring.datasource.username=<username> –
spring.datasource.password=<password>
vi. **Java -jar query-1.0-Final.jar –**
spring.datasource.username=<username> –
spring.datasource.password=<password>

vii. **Java -jar rawdata-1.0-Final.jar –**
spring.datasource.username=<username> –
spring.datasource.password=<password>

viii. **Java -jar relay-1.0-Final.jar**

c. Navigate to each microservice folder with gradle.settings in each folder
    i. Gradle bootrun
d. Additionally, you can run all the services in the background by double-clicking the jars
e. If all the microservices are set up as projects in IntelliJ, you can also run the spring boot applications as services in the IntelliJ GUI (this is typically how I ran it because it was easy to test and start-up all in one place).

8. Run Simulated python device to send messages
    a. Open the command prompt and change the directory into the simulated device directory at the root of this project's folder.
        i. **ex: cd OneDrive\Documents\GitHub\GraduateProject\SimulatedDevice**
    b. Must ensure the Paho MQTT library is installed before running the simulated device
        i. **ex: pip install paho-mqtt**
    c. Send the messages running the device
        i. **ex. python SimulatedDevice.py**
9. Messages should have propagated the system now, so they should be visible via the front end
10. Start Front end

a. This can be done using **npm start** in the frontend/grad-project folder, open a new command prompt terminal and change the directory into the frontend\grad-project directory
    i. **ex: cd OneDrive\Documents\GitHub\GraduateProject\frontend\grad-project**
b. Install npm dependencies and start
    i. **npm i**
    ii. **npm start**
c. Additionally, you can use **npm run build** and then cd into the build folder and run **npx serve** to serve the build
    i. Build is located in frontendBuild folder at the root of the project
    ii. **ex: cd OneDrive\Documents\GitHub\GraduateProject\frontendBuild**
    iii. Then run **npx serve**

This concludes the section on running this system with commands and text-only. The following section has screenshot examples to show what it should look like including the directories used to make it work.

Instructions: Run System With Pictures

This section is similar to the above section. One main difference is that it shows screenshots of expected results and the output to provide visual guidance in running this project. Again, these two sections are separated for ease of viewing because it is only necessary to have the commands, and in others, it is helpful to have the screenshots for guidance.

1. Apache Kafka - Set zookeeper.properties and server.properties.

```
15  # the directory where the snapshot is stored.
16  dataDir=c:/kafkatwo/zookeeper-data
```

```
# A comma separated list of directories under which to store log files
log.dirs=c:/kafkatwo/kafka-logs3
```

These lines must be changed to reflect the proper log folder for your system after downloading the scala version from Apache Kafka.

2. Apache Kafka - Run ZooKeeper / Kafka (Must be in kafka directory)
    a. Command to run ZooKeeper(must be ran first) - **.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties**

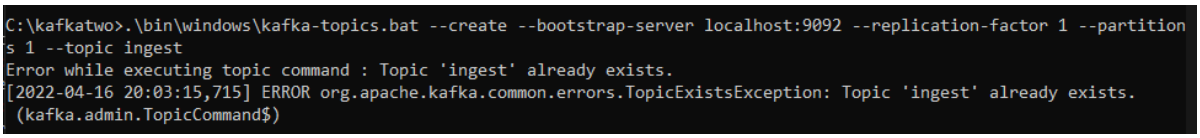
```
C:\kafkatwo>.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
[2022-04-16 19:53:28,854] INFO Reading configuration from: .\config\zookeeper.properties (org.apache.zookeeper.server.quorum.Q
orumPeerConfig)
[2022-04-16 19:53:28,863] INFO clientPortAddress is 0.0.0.0:2181 (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2022-04-16 19:53:28,864] INFO secureClientPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2022-04-16 19:53:28,864] INFO observerMasterPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2022-04-16 19:53:28,864] INFO metricsProvider.className is org.apache.zookeeper.metrics.impl.DefaultMetricsProvider (org.apac
e.zookeeper.server.quorum.QuorumPeerConfig)
[2022-04-16 19:53:28,866] INFO autopurge.snapRetainCount set to 3 (org.apache.zookeeper.server.DatadirCleanupManager)
```

    b. Command to run Kafka (must be ran after starting up ZooKeeper in a separate window or tab) -

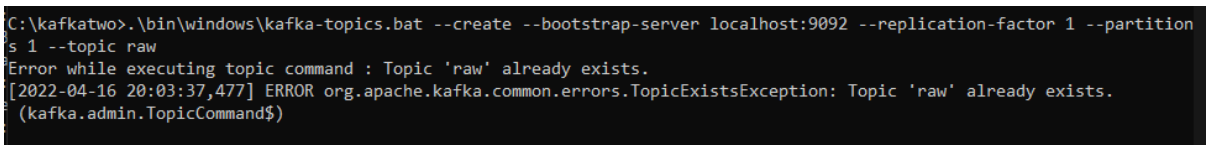
    .\bin\windows\kafka-server-start.bat .\config\server.properties

```
C:\kafkatwo>.\bin\windows\kafka-server-start.bat .\config\server.properties
[2022-04-16 19:56:28,492] INFO Registered kafka:type=kafka.Log4jController MBean (kafka.utils.Log4jControllerRegistratio
n$)
[2022-04-16 19:56:28,805] INFO Setting -D jdk.tls.rejectClientInitiatedRenegotiation=true to disable client-initiated TL
S renegotiation (org.apache.zookeeper.common.X509Util)
[2022-04-16 19:56:28,892] INFO starting (kafka.server.KafkaServer)
[2022-04-16 19:56:28,893] INFO Connecting to zookeeper on localhost:2181 (kafka.server.KafkaServer)
```

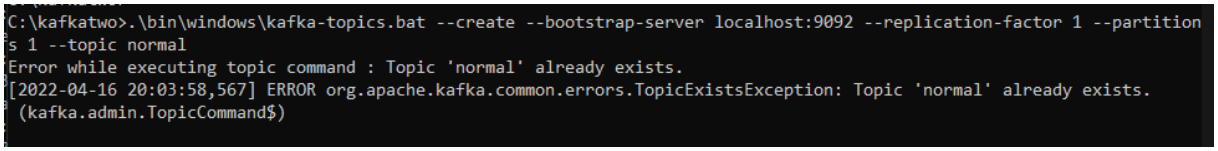
3. Create topics in Kafka - four topics need to be created in Kafka for my system. They are ingest, raw, normal, and alarm.
    a. **Ingest:** .\bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic ingest



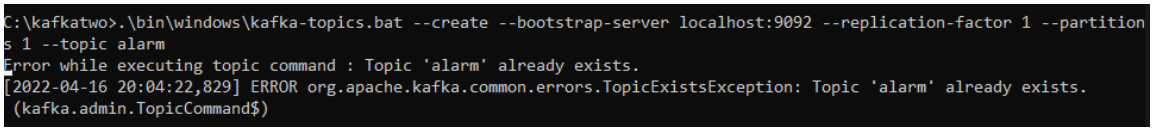
    b. **Raw:** .\bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic raw



    c. **Normal:** .\bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic normal



    d. **Alarm:** .\bin\windows\kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic alarm



4. Start Mqtt Broker - navigate to Infrastructure / MqttBroker / Mos2 folder within the project to be at the correct binary level to run Mosquitto
    a. Start broker using Mosquitto with the following command - **mosquitto -p 443 -v**

```
C:\Users\sobya\OneDrive\Documents\GitHub\GraduateProject\Infrastructure\MQTTBroker\mos2> mosquitto -p 443 -v
1650164779: mosquitto version 2.0.10 starting
1650164779: Using default config.
1650164779: Starting in local only mode. Connections will only be possible from clients running on this machine.
1650164779: Create a configuration file which defines a listener to allow remote access.
1650164779: For more details see https://mosquitto.org/documentation/authentication-methods/
1650164779: Opening ipv4 listen socket on port 443.
1650164779: Opening ipv6 listen socket on port 443.
1650164779: mosquitto version 2.0.10 running
```

    b. Note that -v stands for verbose and adds print statements occasionally; this can lead the MQTT broker to drop messages because it is printing instead

5. Start up / Connect to PostgreSQL
    a. Normal and query microservice access PostgreSQL they require fields in their application.properties to be updated to connect successfully

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=password
```

6. Start up / Connect to MongoDB
    a. Raw and export microservices require fields in their application.properties to be updated to connect successfully

```
spring.data.mongodb.database=device
```

Password and username must also be set if used; however, it is not used in the current instance of MongoDB.

7. Run All microservices; this can be done in many different ways
    a. Navigate to the build-jar folder and java -jar each jar
        i. Java -jar access-1.0-Final.jar
        ii. Java -jar alarm-1.0-Final.jar
        iii. Java -jar export-1.0-Final.jar
        iv. Java -jar ingest-1.0-Final.jar
        v. Java -jar normalize-1.0-Final.jar
        vi. Java -jar query-1.0-Final.jar
        vii. Java -jar rawdata-1.0-Final.jar
        viii. Java -jar relay-1.0-Final.jar
    b. Or if changing the PostgreSQL user or the MongoDB User
        i. Java -jar access-1.0-Final.jar
        ii. Java -jar alarm-1.0-Final.jar –spring.datasource.username=<username> –spring.datasource.password=<password>
        iii. Java -jar export-1.0-Final.jar –spring.datasource.username=<username> –spring.datasource.password=<password>
        iv. Java -jar ingest-1.0-Final.jar
        v. Java -jar normalize-1.0-Final.jar – spring.datasource.username=<username> – spring.datasource.password=<password>
        vi. Java -jar query-1.0-Final.jar –spring.datasource.username=<username> –spring.datasource.password=<password>

vii. Java -jar rawdata-1.0-Final.jar –
spring.datasource.username=<username> –
spring.datasource.password=<password>

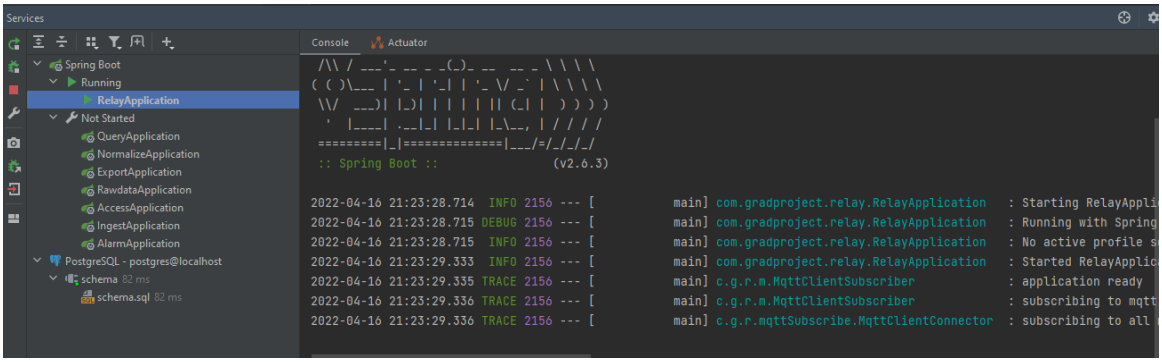
viii. Java -jar relay-1.0-Final.jar

c. Navigate to each microservice folder with gradle.settings in each folder

i. Gradle bootrun

d. Additionally, all services can be run in the background by double-clicking the jars

e. If all the microservices are set up as projects in IntelliJ, you can also run the spring boot applications as services in the IntelliJ GUI(this is typically how I ran it because it was easy to test and start up all in one place). This is an example of having started relay and the ability to start the others still.



8. Python Simulated Device (ran after at least relay, ingest, normalize, rawdata, and alarm services are started) will randomly choose messages from a modified version of the referenced Kaggle dataset.

a. Can alter the number of messages sent and the number of simulated devices. The default sends around 10,000 messages but this example shows 30,000

```
numSimulatedDevices = 300
numMessagesPerDevice = 100
```

b. Navigate to the simulated device folder and

   i. Ensure the paho library is installed by running: **pip install paho-mqtt**

   ii. run: **Python SimulatedDevice.py**

```
C:\Users\sobya\OneDrive\Documents\GitHub\GraduateProject\SimulatedDevice>python SimulatedDevice.py
3510.0750000000003
total
```

This returns the number of milliseconds it took to send all the messages, so it took around 3.5 seconds to send 30000 messages.

9. Messages should now have propagated through the system, creating alerts and being stored in MongoDB and PostgreSQL

10. Start Front End

a. This can be done using **npm i** to install dependencies, then **npm start** in the frontend/grad-project folder to start the react app

npm i

```
C:\Users\Andrew Soby\Documents\GitHub\GraduateProject\frontend\grad-project>npm i

up to date, audited 1260 packages in 2s

177 packages are looking for funding
  run `npm fund` for details

6 moderate severity vulnerabilities

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

npm start

```
You can now view grad-project in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.37:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

assets by path static/ 12.5 MiB
  asset static/media/winnie2.0a9d6123748443fce729.jpg 2.98 MiB [emitted] [immutable] [from: src/pages/pictures/winnie2.jpg] (auxiliary name: main)
  asset static/js/bundle.js 2.71 MiB [emitted] (name: main) 1 related asset
  asset static/media/winnie3.a60dbe3246861ee2f17e.jpg 2.7 MiB [emitted] [immutable] [from: src/pages/pictures/winnie3.jpg] (auxiliary name: main)
  asset static/media/winnie4.bf409f04c093d93bb643.jpg 2.69 MiB [emitted] [immutable] [from: src/pages/pictures/winnie4.jpg] (auxiliary name: main)
  asset static/media/Winston.1cb87f4429f63028d791.jpg 1.41 MiB [emitted] [immutable] [from: src/pages/pictures/Winston.jpg] (auxiliary name: main)
asset index.html 1.72 KiB [emitted]
asset asset-manifest.json 518 bytes [emitted]
cached modules 2.65 MiB (javascript) 9.79 MiB (asset) 28.5 KiB (runtime) [cached] 446 modules
webpack 5.72.0 compiled successfully in 990 ms
```

b. Additionally, you can use **npm run build** and then cd into the build folder and run **npx serve** to serve the build

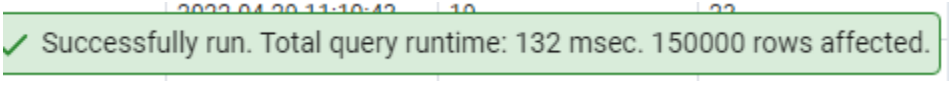
Test Runs

This Section explores how efficiently this system ingests thousands of messages. All of these runs were run without the -v flag on the MQTT broker because occasionally, the broker will drop more messages due to it producing over 60,000 print statements within the time frame. This issue can easily be fixed using a managed service or an MQTT broker with more resources. Still, for the purpose of this project, that is outside the scope as it is a relatively straightforward solution for individuals spinning up these resources.

Below is a screenshot of 5 Test runs where the first value is the milliseconds it took to complete and the second print out is the total messages sent

```
C:\Users\Andrew Soby\Documents\GitHub\GraduateProject\SimulatedDevice>python SimulatedDevice.py
1799.635
total messages sent = 30000

C:\Users\Andrew Soby\Documents\GitHub\GraduateProject\SimulatedDevice>python SimulatedDevice.py
1818.231
total messages sent = 30000

C:\Users\Andrew Soby\Documents\GitHub\GraduateProject\SimulatedDevice>python SimulatedDevice.py
1788.625
total messages sent = 30000

C:\Users\Andrew Soby\Documents\GitHub\GraduateProject\SimulatedDevice>python SimulatedDevice.py
1806.64
total messages sent = 30000

C:\Users\Andrew Soby\Documents\GitHub\GraduateProject\SimulatedDevice>python SimulatedDevice.py
1793.6270000000002
total messages sent = 30000
```

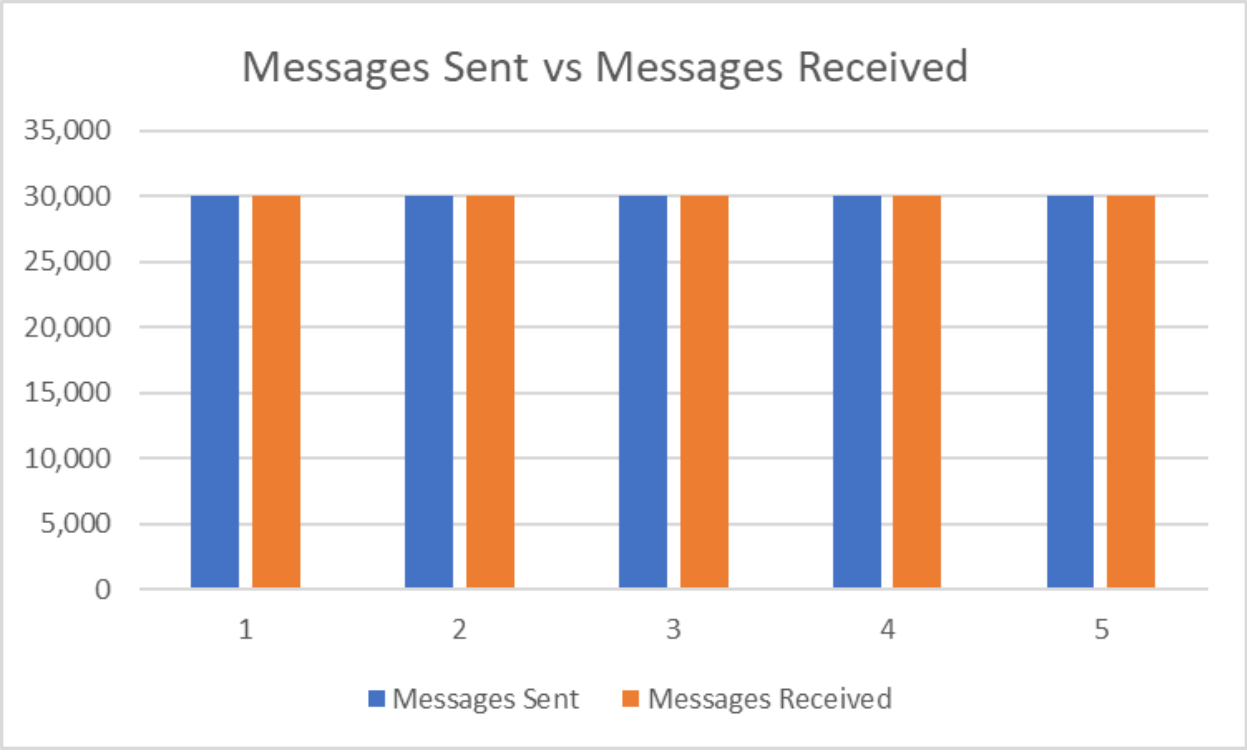
Below is a screenshot of a total of 150,000 rows of devices input into the system for five total runs, each sending 30,000 messages resulting in 150,000 total messages showing all the messages were sent through



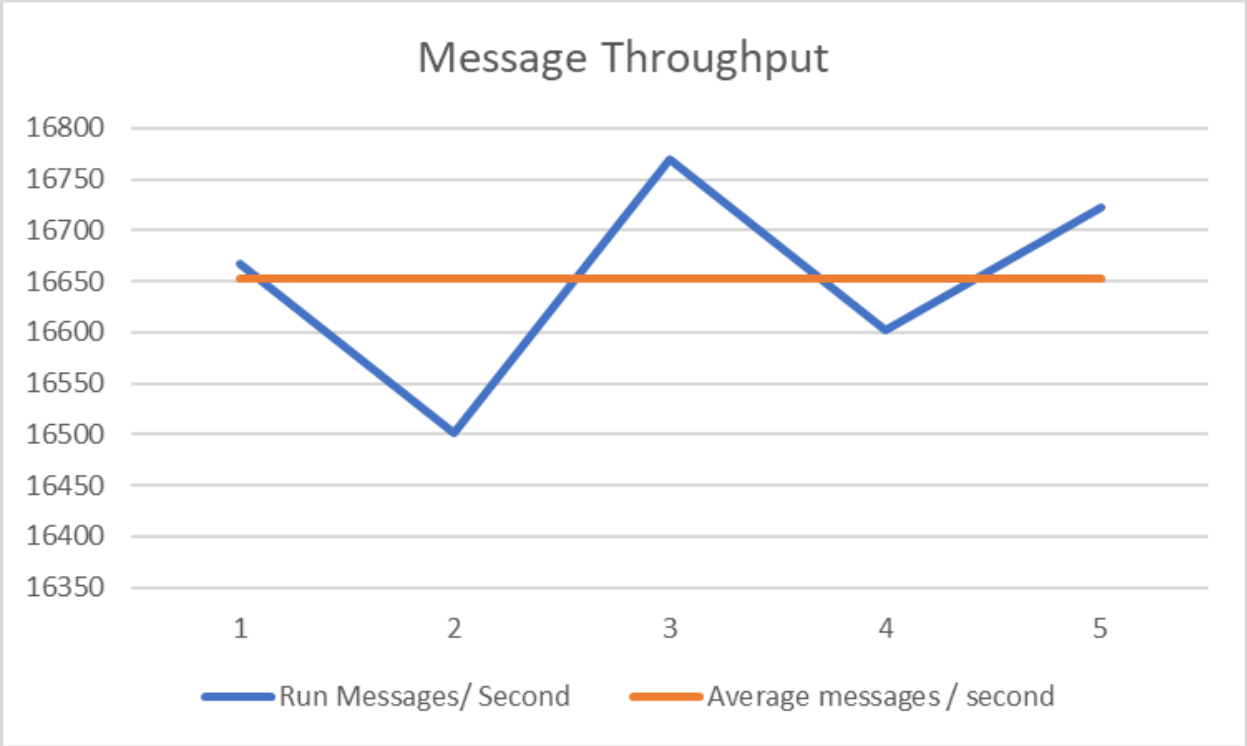
The table below contains the results for all of the columns from the previous five test runs and shows the average amounts of messages ingested per second. These runs were all added together to display the tests' total and consistency.

Run #	Messages Sent	Messages Received	Time (Seconds)	Messages Dropped	Message/ Second
1	30000	30000	1.8	0	16666.66667
2	30000	30000	1.818	0	16501.65017
3	30000	30000	1.789	0	16769.14477
4	30000	30000	1.807	0	16602.10293
5	30000	30000	1.794	0	16722.40803
Total	150000	150000	9.008	0	16651.86501

The following graph contains a bar chart of the messages sent and received per run where the messages sent are in blue and the messages received are in orange. Each of the runs sent 30,000 messages as there were 300 simulated devices worth of threads sending 100 messages each. An additional note to add is if the MQTT broker is run with the verbose -v flag, it can drop more messages because it prioritizes printing the client connections and print statements for messages rather than handling new messages and connections to send out. During this testing, no messages were dropped; however, it would likely be on the IoT device to the MQTT broker side if messages were dropped. This issue can be solved by implementing message send queues into IoT devices to ensure they keep messages in the queue until they are confirmed to have been received.



This graph contains the message throughput per second. The blue line includes the change from each run's specific message throughput, while the orange line has the total throughput average from the end run. The highest value we received was around ~16,800 messages ingested per second. The lowest message throughput was just under ~16,500 messages per second. Again, these tests were performed with the better computing environment of the two that this project was run on and had relatively increased results due to vertical scaling.



Future Work

This project explores building an efficient IoT system locally, leaving room to be expanded on. Future work for this project would involve choosing a cloud environment to deploy this infrastructure. This step was not done in this project because of the cost, but this step would likely be relatively straightforward if time and pricing allow for this to occur. Initially, when I was going to do this project, I was going to use Microsoft Azure's free education credits; however, my account had already been depleted of those credits, so I did not proceed with that direction and instead implemented a front end. Additionally, deploying instances of Kafka, MQTT Broker, MongoDB, and PostgreSQL would be reasonably quick when using a cloud provider and likely could be done using managed services providing the necessary scalability for horizontal and vertical scalability. In addition to the cloud infrastructure being set up, this would allow for an instance of Kubernetes to be used as well. This step would pave a path to dockerize and containerize each of the microservices so that Kubernetes could orchestrate them.

With a cloud environment like Azure, this project could dockerize each microservice and deploy them into Azure Kubernetes Service or AKS. This would allow for the microservices to be orchestrated by Kubernetes so instances can be spun up or down depending on health and demand. Additionally, with new network requirements due to AKS being implemented, using managed services would be helpful to deploy to make it easier to connect the dockerized services with the Kafka, MQTT, PostgreSQL, and MongoDB. The most beneficial managed service to add would be Kafka. It would be much easier to add sink connectors to automatically dump MQTT messages into Kafka, reducing the need for a relay microservice.

Conclusion

This project explores the idea of building an efficient data ingest pipeline for an IoT system by creating an event-driven microservice architecture focused on the ability to scale both vertically and horizontally. It addresses the massive amounts of data being sent by growing numbers of IoT devices and addresses issues that will continue to rise as more devices connect and send data. After testing the system, we can see that messages can be reliably ingested at the lower end of system scalability. Different areas can be scaled to efficiently handle more messages by allocating resources or clones of microservices.

Three separate sections can represent this project, including the simulated device, infrastructure, and microservices. The simulated device emulates realistic IoT messages sent to a system at a high throughput rate. The infrastructure was set up to show how available open-source tools can effectively be leveraged to handle large amounts of messages. While these infrastructure options are available freely, they can also be paid for in managed services that provide benefits such as effortless scalability or setup. The microservices are designed to be simplistic while removing application state logic to provide a microservice that can be scaled easily in a system like Kubernetes. In addition to completing these three sections, an additional section of a react front end was created to benefit the user experience of this system, and an additional alarm microservice was added, showing the ability to propagate alarms in a system like this to drive business decisions.

This project demonstrates the ability to create a data-focused system that can ingest a large and growing number of messages. This is useful for the future of IoT systems as more devices will continue to connect and send data to these systems. The goals laid out for this work were both met and expanded upon to create a system that effectively handles these issues. In addition, it also provides a clear path forward for future work by deploying into a cloud environment and orchestrating the microservices in Kubernetes.

References

Li, Y., Orgerie, A., Rodero, I., Amersho, B. L., Parashar, M., & Menaud, J. (2018). End-to-end energy models for Edge Cloud-based IoT platforms: Application to data stream analysis in IoT. *Future Generation Computer Systems, 87*, 667-678. doi:10.1016/j.future.2017.12.048

Metallidou, C. K., Psannis, K. E., & Egyptiadou, E. A. (2020). Energy Efficiency in Smart Buildings: IoT Approaches. *IEEE Access, 8*, 63679-63699. doi:10.1109/access.2020.2984461

Muangprathub, J., Boonnam, N., Kajornkasirat, S., Lekbangpong, N., Wanichsombat, A., & Nillaor, P. (2019). IoT and agriculture data analysis for smart farm. *Computers and Electronics in Agriculture, 156*, 467-474. doi:10.1016/j.compag.2018.12.011

Oh, H., Park, S., Lee, G. M., Heo, H., & Choi, J. K. (2019). Personal Data Trading Scheme for Data Brokers in IoT Data Marketplaces. *IEEE Access, 7*, 40120-40132. doi:10.1109/access.2019.2904248

Plageras, A. P., Psannis, K. E., Stergiou, C., Wang, H., & Gupta, B. (2018). Efficient IoT-based sensor BIG Data collection–processing and analysis in smart buildings. *Future Generation Computer Systems, 82*, 349-357. doi:10.1016/j.future.2017.09.082

Kaggle IoT Dataset: used and modified for sending simulated device data. https://www.kaggle.com/datasets/edotfs/dht11-temperature-and-humidity-sensor-1-day

Mosquitto / MQTT Broker: I followed an online guide that additionally had the binary download for Mosquitto that was used. http://www.steves-internet-guide.com/install-mosquitto-broker/#manual

Apache Kafka: used to download scala version to run Kafka locally https://kafka.apache.org/downloads