Technical Report - Project specifications

Smartive

Course:	IES - Introdução à Engenharia de Software
Date:	Aveiro, 04/12/2021
Students:	Daniela Filipa Pinto Dias, nMec 98039 Hugo Miguel Teixeira Gonçalves, nMec 98497 Gonçalo Fernandes Machado, nMec 98359 José Pedro Marta Trigo, nMec 98597
Project abstract:	Interactive system to manage inter-connected IoT devices.

Table of contents:

1 Introduction

The Internet of Things is becoming more and more popular with the advance in technology. With new technologies such as 5G and the development of a metaverse, the trends indicate this type of device will become even more popular in the next few years. With this in mind, our group decided to create a project to easily manage a network of IoT devices in order to automate and create a smart home.

With this project, we will be able to put into practice some of the learned concepts in the IES course, such as the creation of a RestAPI to create a bridge between the backend and frontend, the different types of architectures with their advantages and disadvantages, how to efficiently manage a GitHub project to collaborate with a team, between others.

Roles of each member

Team manager: Gonçalo Machado
Architect: Hugo Gonçalves
Product Owner: Daniela Dias
DevOps: José Trigo

2 Product concept

Vision statement

Our product, Smartive, is a system with the objective of connecting and controlling all electronical/IOT devices in a house. The user will be able to connect devices, control them, group devices together and activate them according to a schedule, see what resources were used by a specific device or a group of them during a specific time and share the “house” account with other users. By using our product, the user will have a much easier time managing the devices that exist in his house.

Personas

Name: Frank Washington
Gender: Male
Age: 45
Location: Suburbs
Marital Status: Married
Lives with: Wife, 1 kid(13 years old)
Goals:

- Be able to control devices in his house from an app
- Add his family to the “house”
- Hook different devices together

Name: Isabelle Mendez
Gender: Female
Age: 34
Location: Big city
Marital Status: Single
Lives with: No one
Goals:

- Connect new devices to the application
- See the usages of energy, water, etc. in the house
- Have a safe home, utilizing a security system

Main scenarios

Scenario 1

Isabelle lives in an apartment in a big city. She has a lot of electronic and IOT devices, but since she works a lot and usually gets home late, she can't take too much time to set everything up the way she wants. To fix this, she uses the *Smartive* app to connect all her devices to the app so she can control them from wherever she wants.

Scenario 2

Frank lives in the suburbs and, since his wife and child are home a lot, they use a lot of electricity and water, among other resources. Since Frank likes to have good control over the finances (in order to save money and teach his child to not waste resources), he utilizes the *Smartive* app to check every week how many resources were spent and where.

Scenario 3

Franks lives with his wife and child. Since they have a lot of electronic devices, Frank started using the *Smartive* app to have all devices in one place. In order for his wife and child to be able to control the household devices as he does, Frank adds the wife and child to his account, making his wife have the same privileges as him and restricting what devices his kid can control.

Scenario 4

Isabelle lives all alone in her apartment and since she is in a big city, she is afraid of being robbed. She then bought some security systems to make her house safer, but she also wants to receive notifications when the security system detects something, so she will use the *Smartive* app to connect her security system to it and have it send notifications whenever a device related to security detects something.

Scenario 5

Isabelle is not a morning person, so she takes a little bit of time to get out of bed. Due to this, she always has to eat her breakfast on her way to work and is sometimes late, causing some stress. So Isabelle decides that she will use the *Smartive* app to hook all the devices she needs in the morning together and make them turn on in the morning. (For instance, at 07:30, her blinds open, her alarm goes off, the kettle starts heating up and the shower water starts to heat up)

3 Architecture notebook

Key requirements and constraints

Our system data stream depends on data generated by a set of different virtual sensors. Each one of those sensors is guaranteed to have common characteristics with other sensors but may also have some different characteristics. This can certainly difficult the modulation of a relational database once we would be forced to create several tables to fit each one of the sensor's characteristics. With that in mind, we have decided to use a NoSQL database based on documents - **MongoDB**. The documents will give us the necessary flexibility to store all the sensors in the same collection, without having to create several tables, as a relational database would require.

As the number of registered IoT devices (sensors) increases in our system, the amount of data received which needs to be processed will also increase dramatically. Therefore, we will need to have a robust and scalable system to process all of this data without compromising the availability and integrity of our system. In order to address this difficulty, we will implement an **Event-Driven architecture**. This will allow us to easily create and deploy more processors/workers to process the growing amount of data - **scalability**. This approach will also improve the **availability** of our system once each type of data (temperature, humidity, etc...) will have a large set of available processors meaning that if one of them crashes, there will always be another one to replace it and therefore, maintaining the system available.

The registered sensors will also produce a huge amount of data per unit of time, and although this data may be used to keep our front-end integrations up-to-date in real-time, we don't need **persistence** for all of this data. Consequently, we will use an in-memory database system as a **temporary** and **disposable** caching system (**Redis**) to keep that data while it's needed for our integrations and easily dispose of it when it's not needed anymore. This will avoid a rising overload of our persistent database with useless information and keep our system stable. In order to save a timeline of the events (history of temperature for a given room, for example) we'll utilize a **Batch Processing** unit which will be responsible for saving a snapshot of the sensors data to our persistent data storage every given amount of time.

Both our processors, API, and Batch Processing unit will need to access our databases, to avoid unnecessary usage of connections to our databases and manage concurrency issues, we'll use a **Middleware/Services** layer. For our backend components to access the databases they'll then use the RPC protocol to communicate with the Middleware. The Middleware will then access the databases and provide them with the information they asked for.

The system should be accessible both via a web version and an application for mobile devices. To address this situation we decided to use ReactJS (to the web version) and ReactNative (to the mobile version) once there's a great compatibility between both of them allowing us to easily convert our web app into a mobile app.

Architectural view

To generate the needed data for our system, we'll implement some virtual **agents** written in Python. Each agent will represent and simulate a real-world sensor - we'll have agents for temperature data, motion detection, humidity, etc. The data from those agents will then be posted to a Message Broker (**RabbitMQ**, in our case), which will manage the received data and forward it to the correct **processors/workers** using a **Message Queue**.

Our system will have several **processors** (also written in Python) for each type of data. This will grant a higher **availability** and increase our system **performance** overall by allowing it to recover more easily from eventual failures and crashes.

Each one of those processors will process the received data, apply the needed transformations and store the results in a memory database (**Redis**, in this case) using the **middleware/services** layer as a wrapper.

The **Middleware** layer will also be written in Python using the necessary drivers to access our persistent and in-memory databases.

In order to connect our integrations and clients to the system, we'll use a **REST API** written with **Spring Boot** (Java) which will also be connected to our Middleware in order to access the databases. Our applications/clients will then use HTTP requests to query the API and get the necessary information to present in the front-end.

We'll also have a **Batch Processing** unit written in Java, which will be responsible for performing scheduled operations such as saving, from time to time, the current sensor's data (stored in Redis) to our persistent system storage (MongoDB) in order to build a timeline of some metrics such as temperature and humidity in a given room, for example.

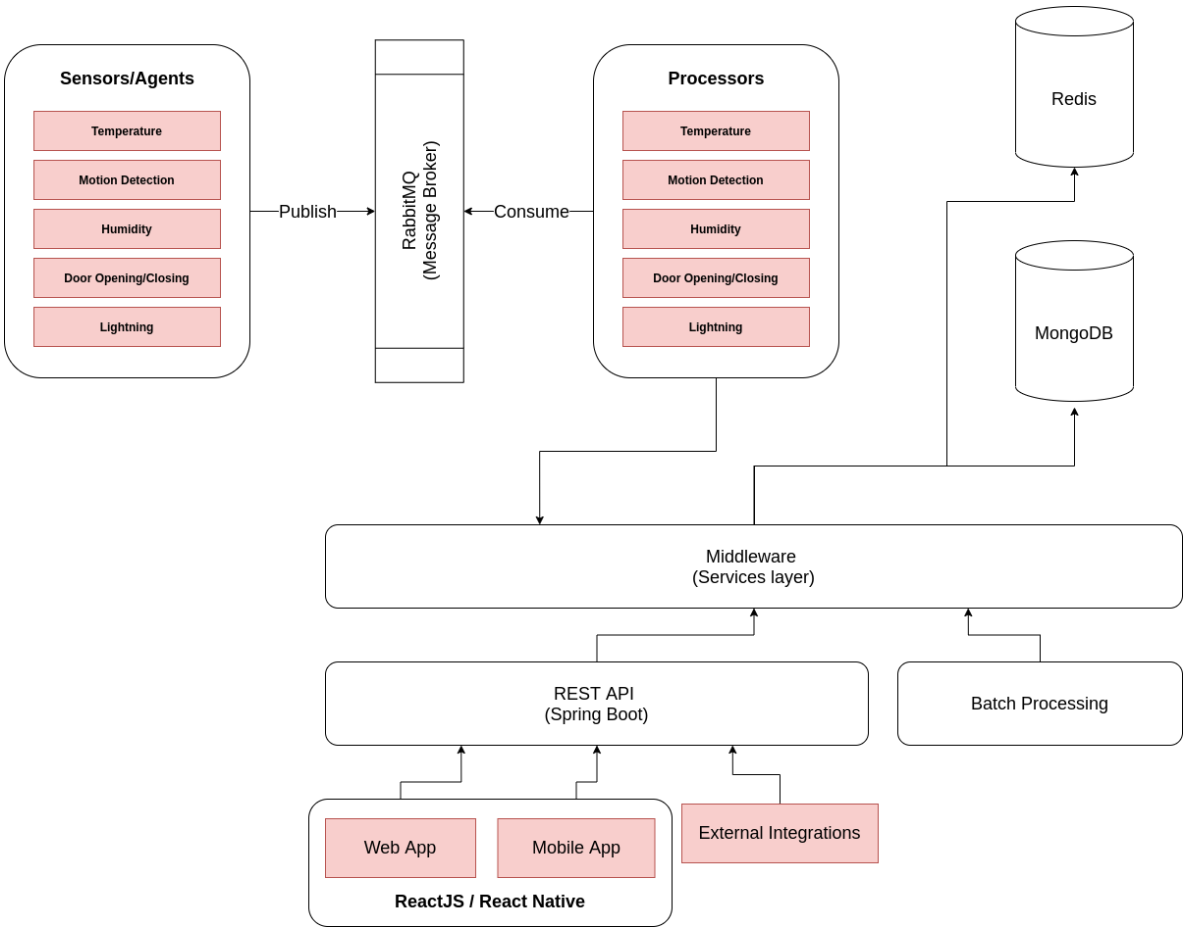
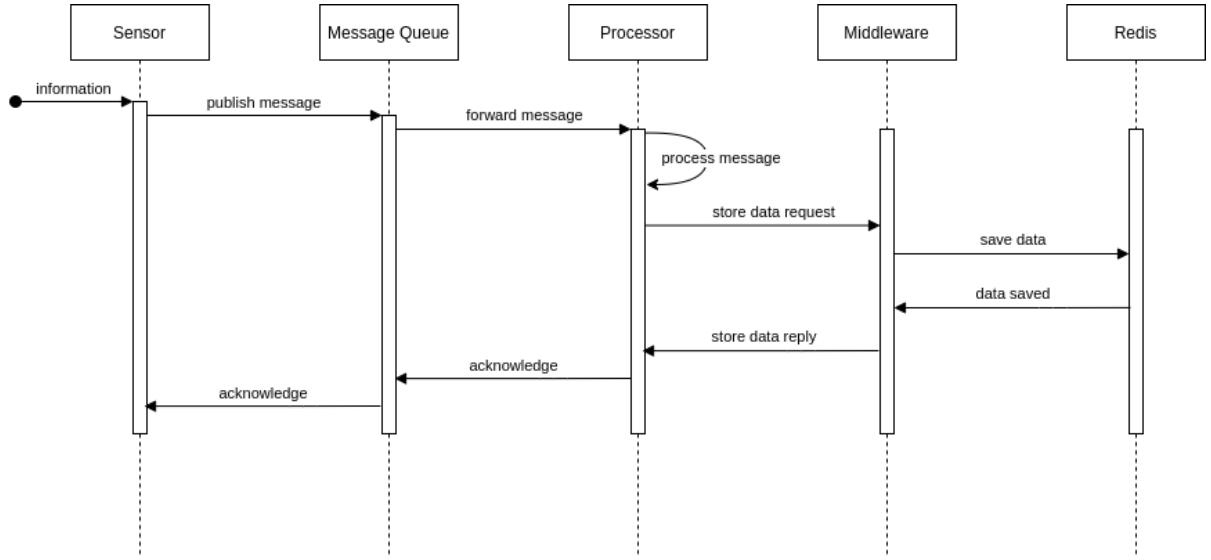


Diagram of the planned architecture.

Module interactions

The **sensors/agents** module will interact with the **processors** module using the **RabbitMQ** library which will implement a **Message Queue** to create an asynchronous channel of communication between these two modules. From time to time, the **Message Queue** will forward the messages to the **processors** which will process the data received, and store it in a **temporary cache** (Redis) using the **Middleware** interface.

These interactions can be represented by the following sequence diagram:



Sequence diagram for data processing.

Using reply messages will allow us to make sure we do not lose any data in the process. If for some reason, the data sending fails then the sender will try again until it was successful or a limit of tries is reached.

As discussed above, all the data processed by our workers will be saved in Redis, once we don't need to save all this data in persistent storage. In order to create a history of the conditions or operations of our IoT devices, we will run a **Batch Processing** unit which will be responsible for saving a snapshot of the data to our persistent storage.

4 Information perspective

<which concepts will be managed in this domain? How are they related?>

<use a logical model (UML classes) to explain the concepts of the domain and their attributes>

5 References and resources

“Internet das Coisas”, Wikipedia, accessed 21 November 2021, <https://pt.wikipedia.org/wiki/Internet_das_coisas>

“What is an Event-Driven Architecture?”, Amazon AWS, accessed 22 November 2021, <https://aws.amazon.com/pt/event-driven-architecture/>

“RabbitMQ vs. Apache Kafka: Key Differences and Use Cases”, Instaclustr, accessed 22 November 2021, <https://aws.amazon.com/pt/event-driven-architecture/>

“Introduction to RabbitMQ”, RabbitMQ, accessed 23 November 2021, <https://www.rabbitmq.com/tutorials/tutorial-one-python.html>

“Introduction to gRPC”, gRPC, accessed 24 November 2021, <https://grpc.io/docs/what-is-grpc/introduction/>

“Pub-Sub vs. Message Queues”, Baeldung, accessed 24 November 2021, <https://www.baeldung.com/pub-sub-vs-message-queues>

“Tutorial: Intro to React”, ReactJS, accessed 29 November 2021, <https://reactjs.org/tutorial/tutorial.html>