Technical Report - Project specifications

Smartive

Course:	IES - Introdução à Engenharia de Software
Date:	Aveiro, 27/01/2022
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 electronic/IOT smart devices in a house. The user will be able to: connect devices (and sensors); set up triggers to initiate actions (on the same device or on others); group devices together by room and more; activate the devices according to a schedule; see what resources were used by a specific device or a group of them during a specific time; share the "home" and its rooms with other users, etc.

By using our product, the end-user will have a much easier time managing the smart devices that exist in his house and keeping track of the power consumption using the installed sensors.

Personas

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

- See the usages of energy, water, etc. in the house
- Add his family to the "house" (in other words, share this account)

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

- Connect new devices to the application
- Get notifications when a device meets a certain condition (triggers)
- Schedule events for different devices

Main scenarios

Scenario 1

Isabelle lives alone in her 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 with his wife and child. His wife works from home and his kid spends all his free time playing video games, so they spend a lot of electricity and other resources. Since Frank likes to have control over his finances, he utilizes the Smartive app to check 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. However, Frank also wants his wife and kid to also be able to control the devices at home. In order to do this, they create an account in the Smartive app and Frank shares his account with them, giving them access to the devices in each room.

Scenario 4

Isabelle lives alone in her apartment in a big city in the northern area of her country, where it's cold most of the year. As such, she bought a heater for her house. In order to not spend a lot of electricity, yet to not get cold, Isabelle used the Smartive app to get notifications whenever the temperature was not between 15ºC and 22ºC.

Scenario 5

Isabelle has a very busy work schedule, so she does not spend much time at home. In order to have all her devices the way she wants by the time she gets home, she uses the Smartive app to connect all the devices and control them from wherever she wants, by scheduling all kinds of events.

3 Architecture notebook

Key requirements and constraints

Our system data stream will depend 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 ones. This can certainly make the modulation of a relational database difficult since 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. 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, maintain the system available.

Our internal services - processors and generators - will need to perform administrative operations such as modifying the sensor state, removing and adding new available devices, etc. These operations should be handled and made available by our **Middleware**. The Middleware should also be capable of performing aggregation operations and sending notifications to the front-end integrations.

Our API *endpoints* should use **Authorization** and **Authentication** mechanisms to protect the API against unauthorized accesses and operations. Therefore, we will use access tokens to protect the endpoints.

Our application should be accessible from the **web** browser once it allows our users to connect to their accounts whenever and wherever they are. The application will also need to have good performance once it will need to display and change a lot of data in real-time.

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, 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 (**MongoDB**, in this case), using the **middleware/services** layer as a wrapper.

The **Middleware** layer will be written in Java and integrated within the Spring Boot application. The Middleware also performs some aggregation operations such as calculating the statistics of a room. It also has the responsibility to **notify** the front-end - through a **RabbitMQ-WebSocket** connection - when some important change occurs in the data in general. This Middleware exposes a Spring Controller (protected by Authentication and Authorization mechanisms) which allows our services to contact it with ease.

In order to secure our API, we are also using **Authentication** and **Authorization** mechanisms. The Authentication is implemented using access tokens (JWT) that are emitted for a user when he logs in with his username and password. This token is then stored in the local storage of the browser and used as Authentication for the user endpoints. As stated before, the Middleware endpoints are private and should only be used by our internal services. To make sure these endpoints are secure, they will only accept requests from the "admin" user - a user created by us who has admin permissions - Authorization.

In order to protect our users, their sensitive data (i.e. passwords) is also encrypted in our database using *Bcrypt*.

The API also exposes some public methods where no Authorization is required; that is the case of the register and login endpoints, for example.

In order to keep our front-end (ReactJS) updated in real-time - as the data generated changes - we used a WebSocket approach. We've followed this approach as in the real world the data of the sensors may change in an irregular way - some sensors may take 1 minute to get new data, and others may take 5 seconds. Therefore, using *pooling* to get the data from the API may not be the most appropriate approach as we may, for one side, be losing some data (if we define a high time between requests) or, by the other side, overload the API with too many, and unnecessary requests (if we define a low time between requests). With WebSockets, we receive the notifications from the Middleware, we process the notifications and get just the necessary information for that moment and for that specific notification, not more and not less than we need.

The system front-end was implemented using ReactJS once it eases the creation of a SPA (Single-Page application) and is more intuitive and easier to learn than some of the other alternatives (such as Angular, for example). We also use Redux to ease the real-time updates in some parts of the application and implement the toasts service.

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.

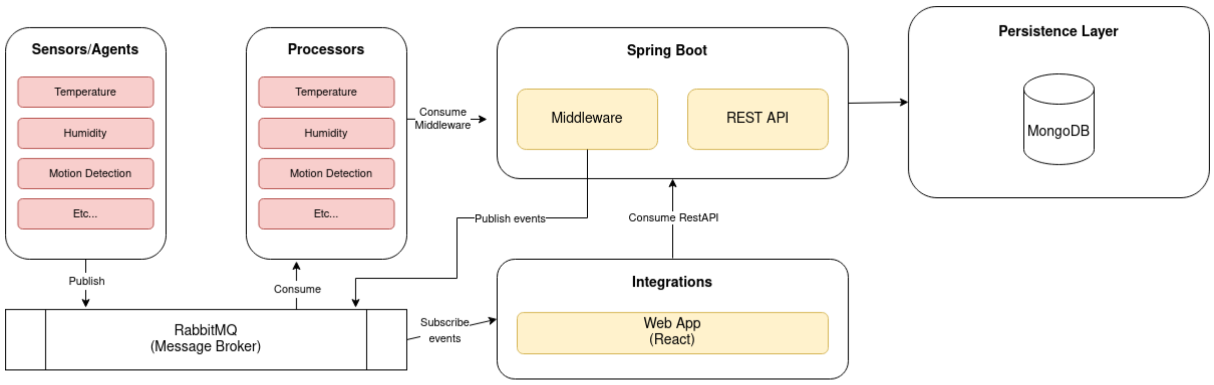
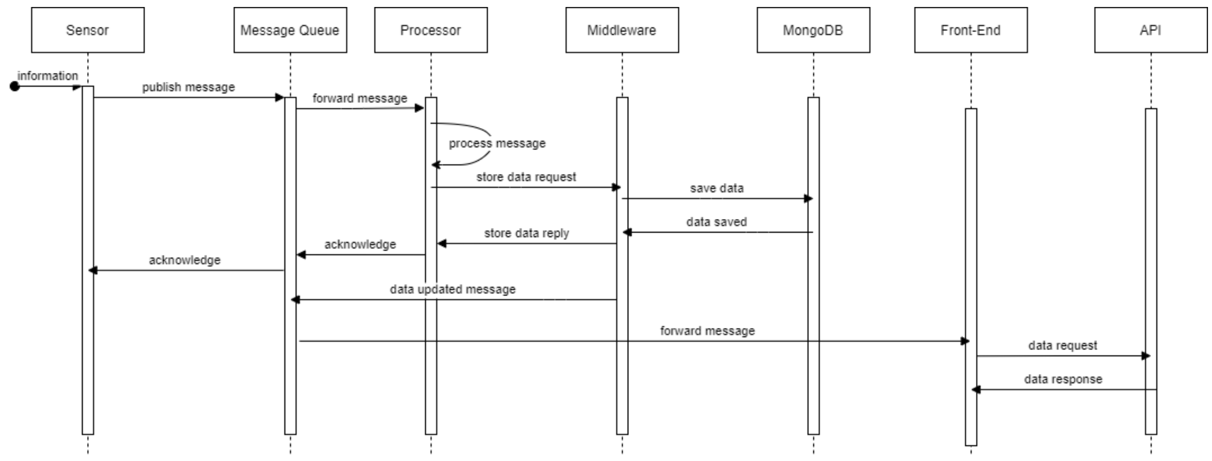


Diagram of the planned architecture.

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 the **database** (MongoDB) using the **Middleware** interface.

These interactions can be represented by the following sequence diagram:

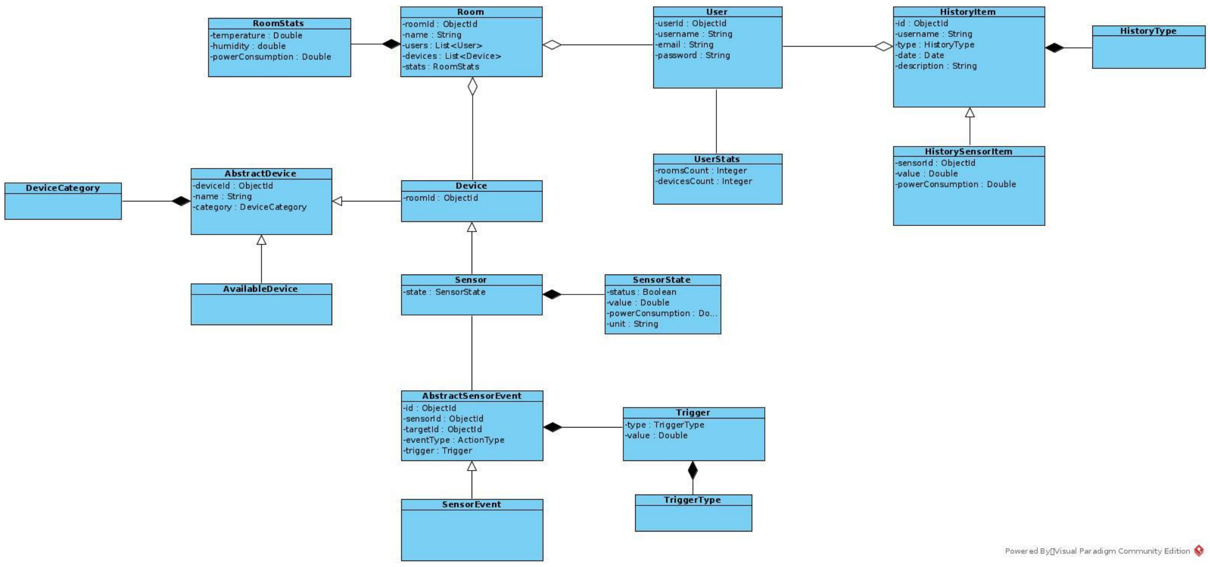


Sequence diagram for data processing.

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.

4 Information perspective

In order to illustrate and explain the concepts of your domain, Smartive, and their attributes, we used a domain model (UML classes). A domain model is a visual representation of conceptual classes or real-situation objects in a domain, along with their different relationships.



Smartive domain diagram.

In our domain, we use the following entities to represent our data:

User

The user is the entity that interacts with the application Smartive. This entity will contain some basic information about our users: username, email, and password. Both the username and email are unique fields.

UserStats

This will store the statistics of a user. Currently, it is used to store the number of rooms the user has access to and the number of devices the user has access to.

Room

This entity represents a room in the application. It has a list of devices associated with that room and a list of users with access to that room.

RoomStats

This entity stores the current statistics for a room, i.e, the current temperature (given by the average temperature value of all sensors), the current humidity, and the power consumption.

HistoryItem

An entity representing an item in the app history. This can represent a different set of actions defined in **HistoryType**.

HistoryType

Defines the different types of history available: history for devices events, rooms events and triggers events.

HistorySensorItem

A subtype of HistoryItem to store history actions related to sensor’s data change. The app creates, and stores a new HistorySensorItem each time the data in some sensor changes.

AbstractDevice

Represents a device in the application.

Device

A subtype of AbstractDevice. Represents a device in the application.

DeviceCategory

Represents the different types of device category. In our application, at this moment, there are only two different categories: temperature and humidity.

AvailableDevice

Extends Device, i.e, it is a subtype of Device and represents an available device. Available devices are devices that are ready to be registered by some user in some room.

Sensor

It is a subtype of Device and represents a sensor in the application.

SensorState

Represents the current state of the sensor. It stores the data generated by the generators and sent to the middleware by the processors.

AbstractSensorEvent

It is an abstract class to represent the sensor events/triggers.

SensorEvent

Implementation of the AbstractSensorEvent. Represents an event/trigger in the application.

Trigger

This entity stores the trigger of a given event. It stores the value at which the event should be triggered and the type of the trigger (instance of TriggerType)

TriggerType

It represents the different types of triggers. In our case, we currently have triggers on a value higher, less, equals, or similar to a given limit.

5 References and resources

API Documentation

https://documenter.getpostman.com/view/16743908/UVR5sVHN

References

"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>