

**DEV MANUAL**

**Software Engineering , 2019-2020**

**Project: DataSiren**

**85070 - Daniel Marques**

**84734 - Fábio Alves**

**84804 - Inês Justo**

**67453 - Miguel Neves**

**84891 - Raquel Rainho**





**Document date: 15-Jun-2020**

**Index**

[**1 – Testing** 4](#_Toc43138908)

[**1.1 – Production Tests** 4](#_Toc43138909)

[**1.2 – Integration Tests** 5](#_Toc43138910)

[**1.2 – Conclusions** 6](#_Toc43138911)

[**2 – Implemented System** 7](#_Toc43138912)

[**2.1 – Architecture** 7](#_Toc43138913)

[**2.1 – Data Storage** 8](#_Toc43138914)

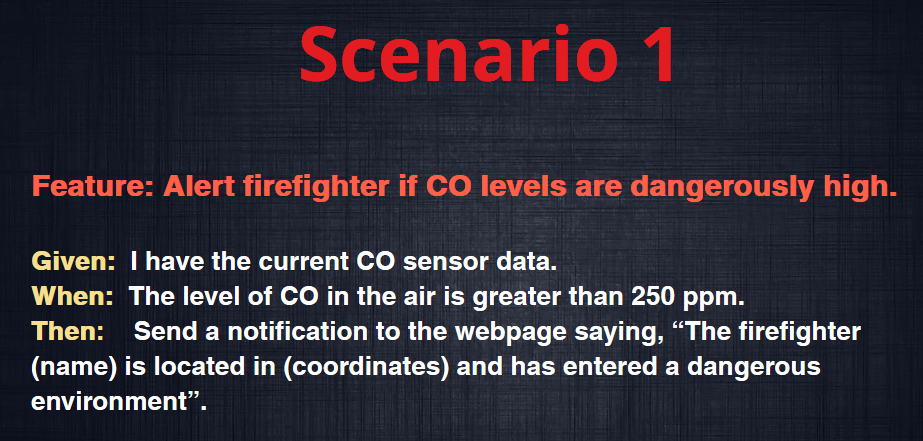
[**2.1 – Functionalities** 8](#_Toc43138915)

[**2.2 – Model** 8](#_Toc43138916)

## 1 - Testing

Tests were elaborated to verify our system’s requirements and its acceptance criteria.

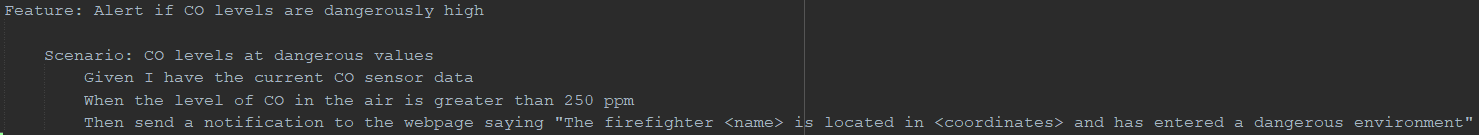
From the scenarios presented in our website, we focused mainly on scenario 1, which aimed to demonstrate our data processing and notification system. The feature evaluated consists in generating notification warnings when there is data received relative to dangerous levels of environment values (CO, temperature, humidity), heart rate or alert when the device’s battery level is low.

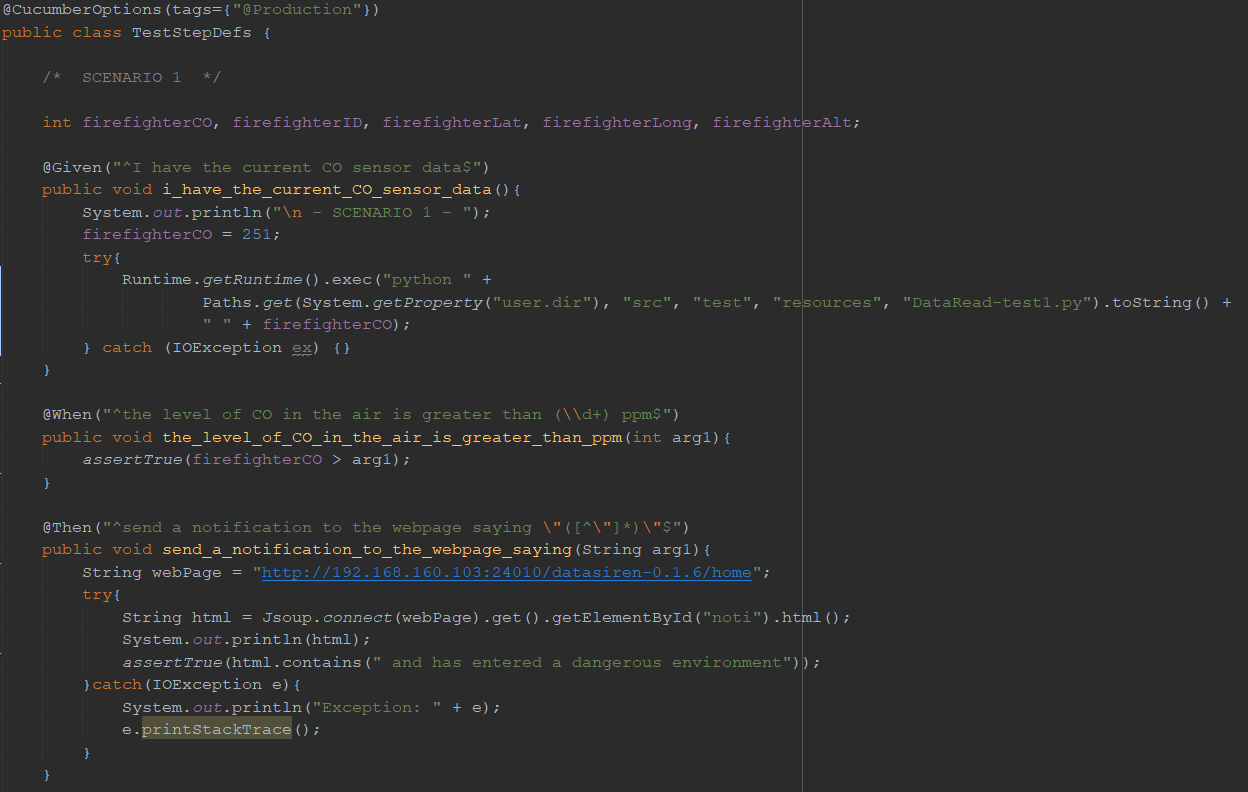
  
Fig. 1 – Scenario 1 description.

## – Production Tests

As mentioned, from our first scenario we created a production test that, with our system running, injected firefighter data with a high CO value (above 250ppm) – using a python script similar to the one used to read/send data in the system’s standard execution – and verified if in our website dashboard presented a notification warning that a firefighter was in a dangerous environment – by accessing the html code of the webpage.

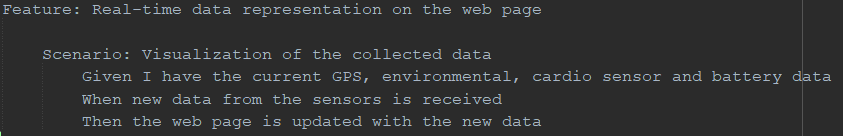
The result was a successful production test using the *Ghernkin* syntax and java files shown below.

  
Fig. 2 - *Ghernkin* syntax file of scenario 1 (“high\_co\_levels.feature”).

  
Fig. 3 – Java cucumber test file of the scenario 1 (“TestStepDefs.java”).

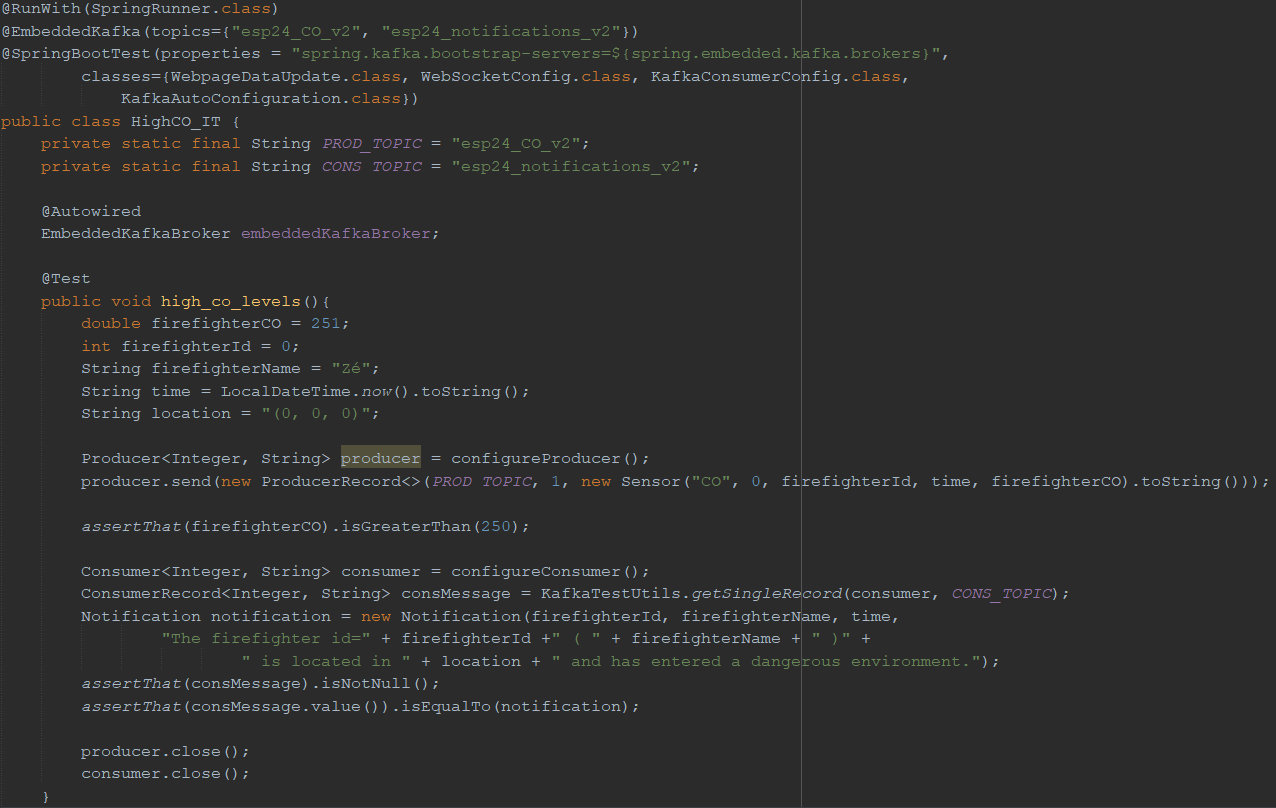
We also started testing our scenario 5, but it was halted due to time limitations and its similarities with the previously mentioned scenario, as this advancement was not too crucial for the software/technological exploration of the subject’s objectives.

Although not fully implemented, its testing also included a *Ghernkin* syntax file and java code similar to the one used in scenario 1.

  
Fig. 4 - *Ghernkin* syntax file of scenario 5 (“data\_visualization.feature”).

## 1.2 – Integration Tests

The integration test developed was also focused on our scenario 1. Its main logic is equivalent to the one within the production test, as the difference to this one is that only the system’s data analysis and notification creation parts are running, instead of the entire project – using the java classes associated with those parts and embedded *kafka*.

  
Fig. 5 – Java file of scenario 5’s integration test (“HighCO\_IT.java”).

## 1.2 – Conclusions

From the success of the implemented tests and the manual assessment of our system, we can guarantee that its main features are working as intended.

The data that is received is presented correctly on the web dashboard and the data analysis is well done as it prompts the correct notifications in the webpage.

As such, the firefighter’s field team can be supervised from our dashboard, using the notifications and map positions to better coordinate, strategize and resolve the mission’s events.

## – Implemented System

## 2.1 – Architecture

Fire Brigade Sensors -> System

A python script sends data from every firefighter to the kafka topic "AllSensorData".

System - DataSeparationService

The data for the kafka topic "AllSensorData" is consumed, separated (each part has only the data corresponding to a sensor or the fireman's name) and sent to the corresponding topic.

System - SensorProcessService

The kafka topic data (sensor topics and name topic) is consumed by a consumer group “SensorProcessing”. Then, each value of each sensor is analyzed to find out if there is any reason to generate one or more notifications. Finally, if a notification has been generated, it is sent to the topic “notifications”.

System - DataStoreService

The kafka topic data (sensor topics) is consumed by a “mainDatabase” consumer group. Then, the data is grouped into a list of firefighters and their sensor data. Finally, the list is sent to the MySQL database.

System - WebpageDataUpdate

The kafka topic data (sensor topics and notification topic) is consumed by a group of consumers “UpdateWeb”. Then, the data is grouped into an Object node. Finally, data is sent to the web page every 5 seconds using web sockets (30 seconds after the system starts so that all services have started, and the initial sensor data is ignored).

ELK

We accept messages from the kafka topic “esp24\_notifications\_v2” as input and send the data to elasticsearch, index esp24\_notifications, passing elasticsearch credentials and host 192.168.160.103:9200.

## 2.2 – Data Storage

The data is being stored in a MySQL database, which is in a pre-existing container on the virtual machine (port 3306).

The data is stored in 3 tables. One with the time/date and an identifier for the fire department. Another with the correspondence between the id of the previously mentioned table (time/date) and the id of each firefighter. And a last one with the data corresponding to each fireman.

At the moment, the only operation to be carried out on the database data is to confirm that the data was stored, by counting the number of entries in the database after each addition.

## 2.3 – Functionalities

The system checks if any sensors have values ​​considered problematic. If so, a notification will be generated.

Creating a new type of notifications is as simple as adding a new case to the “switch” of the “getMessage” function, located in the “SensorProcessService” service.

The addition of new sensors is slightly more complex than the operation mentioned above. For this, it is necessary to add the topic corresponding to the new sensor to the configuration file “KafkaTopicConfig” and, in the separation service, send the new sensor's data to the new topic created.

As for making use of the data of a new topic to generate notifications for the web page, it is only necessary to create a KafkaListener in the service “SensorProcessService” to consume the new topic's data and define the new type of notifications. If a new topic's data is to be stored in the database, it is only necessary to add the sensor to the data model “Firefighter”, updating all services that make use of this model and adding a KafkaListener to the service “DataStoreService”.

The system shows each firefighter's location on the map in real time, being updated every 5 seconds (It only starts to update the site 30 seconds after the system is running, to filter the initial data collection made by the sensors. This data is of zero importance to the system). It also shows all data related to each firefighter (sensor values ​​and name).

Notifications generated by the system are shown on the website.

Finally, using ELK it is possible to check notifications in a more structured way which allows for easier data analysis.

## – Model

"FirefighterGroup" has the creation time as an id and a list of "Firefighter".

“Firefighter” has the id and data for all sensors that are used by each firefighter.

“Location” is used to structure the information sent by kafka regarding this topic. It has the fireman's id to which the data refers, the time at which the data enters the system and the latitude, longitude, and altitude.

“Notification” is used to structure the information sent by kafka regarding this topic. It has the firefighter's id and name to which the data refers, the time at which the data enters the system and the notification it is supposed to send to the site.

“Sensor” is used to structure the information sent by kafka regarding this topic. It has the firefighter's id to which the data refers, the time the data enters the system, the id and the name of the sensor (to allow a firefighter to have several sensors of the same type if necessary) and, finally, the sensor value.

## 3 – Operational Information

LOGSTASH (reason explained on architecture topic in the web page):

Pull the local logstash image:

docker pull docker.elastic.co/logstash/logstash:7.7.1

Build the dockerfile that configures the container's logstash:

cd es\_logstash

docker build -t “logstashdatasiren:dockerfile”

Run logstash:

docker run logstashdatasiren:dockerfile