POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE
IN INGEGNERIA INFORMATICA

TEMA D'ANNO
IN
ADVANCED SOFTWARE ENGINEERING

FAST ALERT HEALTH MONITORING v2.0

Docente:
Prof.ssa Marina Mongiello
Referenti:
Dott. Francesco Nocera
Dott. Leonardo Avena

Studenti:
Flavio Fanigliulo
Mario Milella

Anno accademico 2017/2018

Contents

INTRODUCTION

Fast Alert Health Monitoring (FAHM) is a web application implemented to optimize remote monitoring of patients by offering health managers a real-time notification service for potentially dangerous clinical events.

In particular, in this project the aforementioned application was the subject of an in-depth analysis with the aim of improving it and adding new features. Then, after a study and analysis of the application and the technologies used such as Complex Event Processing, implementation and architectural decisions were taken such as the adoption of a new CEP technology (*Apache Flink*), different from the previous one (*WSO2 Complex Event Processor*), and the introduction of a security system based on the TLS (*Transport Layer Security*) cryptographic protocol, in order to observe improvements in performance and an increase in overall security and reliability.

1. ARCHITECTURAL DESIGN

The main goal of FAHM application derives from the need to implement remote monitoring systems that have high performance especially in the phase of identifying clinical situations potentially dangerous for the health status of a patient. Moreover, it is essential to have a system that is easy to use and easily accessible from different devices, also guaranteeing access to patients in order to provide them with information on their clinical picture and possibly about the first aid actions to be performed in the case of worsening health conditions.
It is therefore necessary that the application architecture aims to meet these requirements.

1.1. COMPLEX EVENT PROCESSING (CEP)

With regard to the need to develop a system that allows rapid and immediate processing of data flows from acquisition systems, the Complex Event Processing processing model [1] has been selected.

CEP (Complex event processing) is a system formed by real-time information processing techniques and methods used to supervise, guide and optimize activities in real time. Generally, this CEP system is an integrative component of a larger system and is based on an Event Driven Architecture (EDA) based on event detection. The information flows will be "organized" in the form of events occurring in the external world that will then be filtered and combined into higher level events.
Through the elaboration of these events, one can therefore understand what is happening in the external world. These elaborations occur to the match of certain

patterns that generate notifications from the subjects themselves. The origin of this system must be sought in the domain of publish-subscribe systems. Unlike systems related to the publish-subscribe domain, where events are considered separately and filtered according to their relevance for the application scenario, Complex Event Processing systems use a more powerful language from the expressive point of view to define patterns involving occurrences of multiple events related to each other.
Event processing is a process of computation on events. Computation can be: analysis, creation, transformation and deletion.

The introduction of CEP systems in healthcare has made it possible to make patient care better and more complete. This integration was made possible thanks to the evolution of biomedical sensors and the use of smartphones / tablets and advanced data flow extraction techniques, all of which are part of a new generation of patient monitoring systems. These new systems are also designed to be interoperable with existing solutions. The development of these systems was also indispensable for the mobile monitoring of patients before and after surgical treatment. The current health monitoring systems are based on portable recorders that can record health data collected from medical devices and downloaded to an application server.

This model was developed with the assumption of processing data streams in a timely manner exceeding the limits of traditional database management (DBMS), which requires the storage and indexing of data before it can be processed: this constraint, together with the liabilities of DBMS is not suitable for Information Flow Processing. Like we said above, the fundamental aspect of this approach involves the detection of low-level events[1] and their subsequent processing through filtering operations useful for the system to identify complex events to be notified to observers.
So, the three main characteristics of the event processing are:

- Abstraction - The operations that make up the processing logic can be separated from the application logic, thus allowing the modification without involving the applications that produce and consume the events.
- Decoupling - Events detected and produced by a specific application can be used and consumed by totally different applications.
- Centrality of Reality - Event processing often refers to episodes that occur - or should occur - in the real world. In the case in question an event is considered as a warning of a probable danger to the health conditions of the patient under observation.

These characteristics were completely similar to those shown by the application.

[1] An event is something known that happens inside or outside a system; it could represent a problem or an impediment, an opportunity, an unexpected behavior. The term event is often used to refer interchangeably both to the occurrence of a specific set of conditions (complex event) and to the occurrence of a set of single events (composite event)

The components of the FAHM system have been identified starting from the architectural model proposed by the doctors Cugola and Margara [1] (Figure 1).

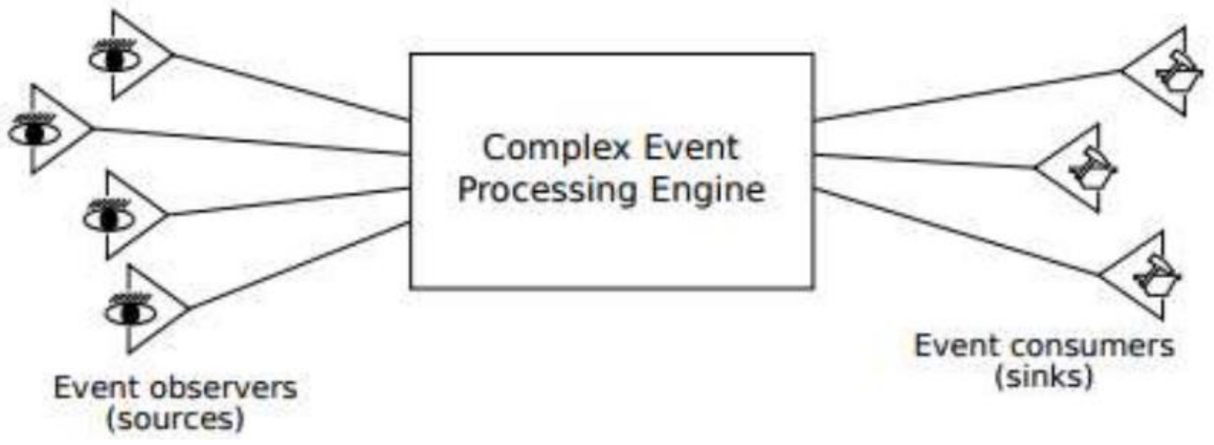


Figure 1- CEP architectural model

In a generic iteration we talk about Event producer to express who will generate events and Event consumer, entities at the edge of the event processing system that receives messages from the producer. Some examples of Event producers are the sensors, which will produce an event based on a particular detection. Event consumers can store events, or display them in a list, or decide on any actions in response. An application can contain one or more event generators, called Event producers.

In the case of FAHM application, the sources of information have been identified in the monitoring systems that acquire patient related data. The observers of the results of the elaboration will instead be the health managers who access this system.

In order to guarantee ease of use and high performance even by accessing the information processed by the system, the application uses a client-server architectural model. In particular, the Server has been assigned the management of users, session, resources and also the storage and forwarding of messages[2] detected by the CEP system. The client represents the consumer of such resources.

1.2. ADVANCED MESSAGE QUEUING PROTOCOL (AMQP)

The application also uses - assuming that the system is to be used by a large number of patients, from which a large amount of data will be detected - a messaging broker

2 In this document, the term message is used to indicate the complex event detected by the system from CEP machine and ready to be managed by the user.

in order to reduce the Application Server's workload. With this in mind, it was decided to use the Advanced Message Queuing Protocol (AMQP), which consists of a message-oriented middleware. AMQP is one of the most important application-level protocols used for the exchange of messages.
It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).
The AMQP protocol provides for the abstraction of the different entities participating in the system and simplifies its management in the communication phases [2].
In particular in the FAHM system the participating entities have been identified in the following components:

- data acquisition systems of vital and non-vital parameters, located at the patient's home;
- server on which the CEP machine resides, which will process the message flows coming from the detection systems, saved on a queue; this server will then publish the events identified on an output queue, which will be accessed by the application server;
- application server that manages interface with end users (doctors and patients).

In this project, *RabbitMQ* was used as an implementation of AMQP protocol. In fact, it is an open source message broker software (sometimes called message-oriented middleware) that implements the Advanced Message Queuing Protocol (AMQP).

1.3. SIDDHI: An open source Complex Event Processor

Siddhi is a 100% open source Complex Event Processor(CEP) offered by WSO2, Inc. It was initially started as a research project at University of Moratuwa, Sri Lanka. The main application of Siddhi is processing data streams in real time. It is written in Java and thoroughly optimized for high performance.

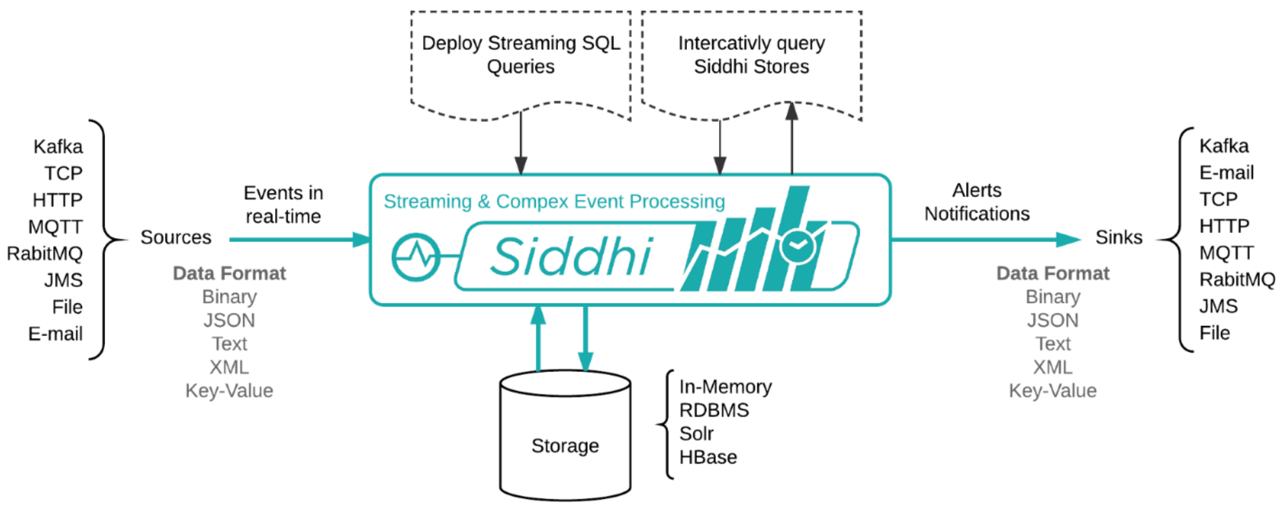


Figure 2- Siddhi architectural model

As you can see above Siddhi,

- can accept event inputs from many different types of sources
- process it to generate insights
- publish them to many types of sinks.

To use Siddhi a user will write a Siddhi query. After writing a Siddhi query and starting a Siddhi application, it will

1. take data event-by-event
2. process the input data in each event
3. add the high level data generated to each event
4. send them to the output stream.

Siddhi Query Language (SiddhiQL) is a rich, compact, easy-to-learn SQL-like language and it has been used in this project work to process streams and identify complex event occurrences.
In particular, Siddhi queries describe how to combine existing event streams to create new event streams. When deployed in the Siddhi runtime, Siddhi queries process incoming event streams, and as specified by the queries, they generate new output event streams if they don't exist.
In the context of this project work, "Siddhi CEP" has been used to enable compatibility between siddhi and apache flink.
"*Siddhi CEP*" is a lightweight and easy-to-use Open Source Complex Event Processing Engine (CEP) released as a Java Library under Apache Software License v2.0. Siddhi CEP processes events which are generated by various event sources, analyses them and notifies appropriate complex events according to the user specified queries.
In particular we used a light-weight library to easily run Siddhi CEP within flink streaming application.
This was achieved by adding flink-siddhi to maven dependency:

```
<dependencies>
        <dependency>
                <groupId>com.github.haoch</groupId>
                <artifactId>flink-siddhi_2.10</artifactId>
                <version>1.2-SNAPSHOT</version>
        </dependency>
</dependencies>

<repositories>
        <repository>
                <id>clojars</id>
                <url>http://clojars.org/repo/</url>
        </repository>
</repositories>
```

In this way it was possible to use siddhi query language for event processing even with Apache Flink, registering Flink DataStream with associating native type information with Siddhi Stream Schema, supporting POJO,Tuple, Primitive Type, etc. (in particular POJO has been used) and connecting with single or multiple Flink DataStreams with Siddhi CEP Execution Plan.

1.4. TRANSPORT LAYER SECURITY (TLS) PROTOCOL

Regarding the security of the system, it was decided to improve it by adopting the TLS protocol as a security layer in the protocol stack to protect the data exchanged between the client and the application server, starting from the access credentials until you get to the content of the messages.
The TLS protocol aims primarily to provide privacy and data integrity between two communicating computer applications. When secured by TLS, connections between a client and a server have one or more of the following properties:

- The connection is *private* (or secure) because symmetric cryptography is used to encrypt the data transmitted.
- The identity of the communicating parties can be *authenticated* using public-key cryptography. This authentication can be made optional but is generally required for at least one of the parties (typically the server).
- The connection is *reliable* because each message transmitted includes a message integrity check using a message authentication code to prevent undetected loss or alteration of the data during transmission.

When the connection starts, the record encapsulates a "control" protocol—the handshake messaging protocol (content type 22). This protocol is used to exchange all the information required by both sides for the exchange of the actual application data by TLS. It defines the format of messages and the order of their exchange. These may vary according to the demands of the client and server—i.e., there are several possible procedures to set up the connection. This initial exchange results in a successful TLS connection (both parties ready to transfer application data with TLS) or an alert message (as specified below).

From the implementation point of view of the adoption of TLS over http, first of all the public certificate and the private key relating to the server have been created (which must be validated by a Certification Authority to work on the network).
Subsequently, on the server side of the app.js (Appendix H), https has been implemented so that all the information sent by patients and doctors to the web application is encrypted.

1.5. THE ARCHITECTURAL MODEL

On the basis of the design choices analyzed in the previous section, the following architecture has been identified (Figure 3), where Apache Flink has been introduced.

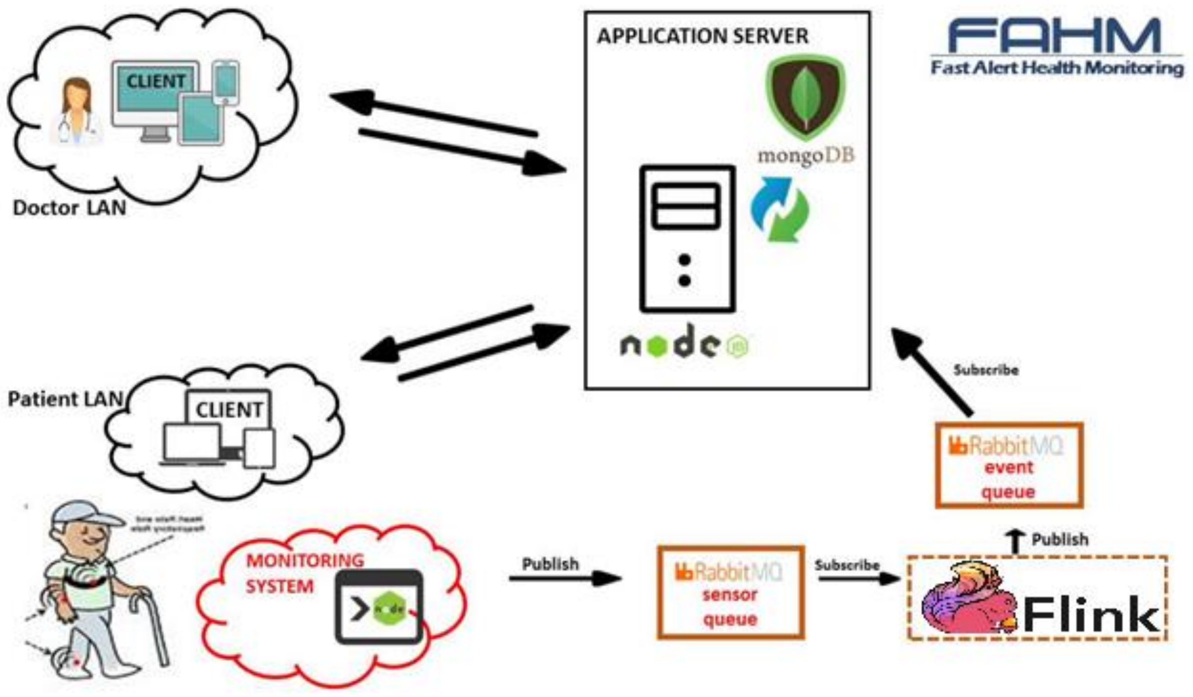


Figure 3 - FAHM architecture

The design solutions used for the construction of each component will be analyzed below. Since the system has been realized following the requirements of scalability and decoupling it is possible to proceed with the analysis of the same considering two sub-systems:

1. Event detection system
2. Interface system

1.5.1. Event detection system

The event detection system is characterized by three components:

1. Clinical data acquisition system: *Node.js*
2. CEP engine: *Apache Flink*
3. Messaging broker: *RabbitMQ*

For design reasons, the *data acquisition system* has been simulated through the implementation of a Node.js server that sends examples of data acquired from the monitoring systems to which the patient is subjected at randomly generated intervals. The data is encapsulated within JSON objects and published on a messaging broker queue; this queue will be the input queue of the CEP engine. The reason why data is encapsulated in JSON objects is to be found in the simplicity of use of this structure in all phases of data processing.

Apache Flink [4] is an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications.
Flink has the following features:

- Provides results that are **accurate**, even in the case of out-of-order or late-arriving data. Flink supports stream processing and windowing with event time semantics. Event time makes it easy to compute accurate results over streams where events arrive out of order and where events may arrive delayed.
- Is **stateful and fault-tolerant** and can seamlessly recover from failures while maintaining exactly-once application state. Flink recovers from failures with zero data loss while the tradeoff between reliability and latency is negligible.
- Performs at **large scale**, running on thousands of nodes with very good throughput and latency characteristics. Apache Flink with its true streaming nature and its capabilities for low latency as well as high throughput stream processing is a natural fit for CEP workloads.

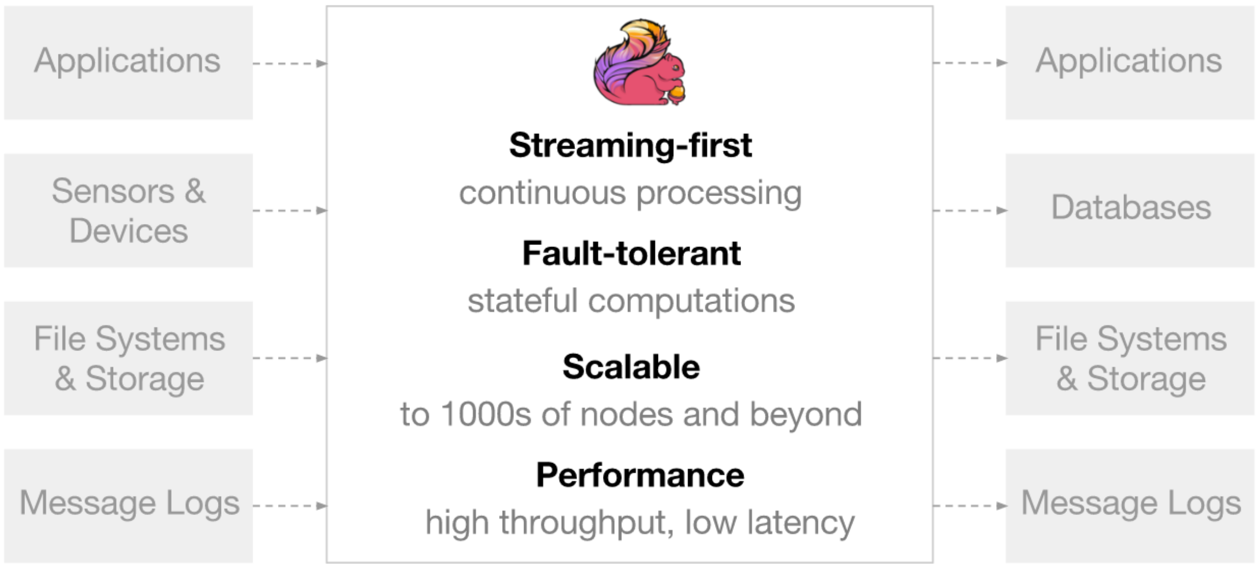


Figure 4- Apache Flink features

So, for the design of the event processing component, it was decided to replace the WSO2 Complex Event Processor with Apache Flink, in order to observe an increase in system performance.
A feature of the WSO2 CEP is the ability to implement flow processing rules using an SQL-Like language. This language, WSO2 Siddhi [3], has been developed to optimize the processing of data flows and is perfectly integrated with WSO2 CEP through a query execution simulator, which simplifies the development and the testing phase.
One of the main difficulties encountered in adopting Apache Flink CEP was that deriving from the lack of the aforementioned native support for this CEP system. As will be explained later, this difficulty has been overcome by adopting a library that allows the use of Siddh Query Language also within Apache Flink.

Among the various messaging brokers who implement AMQP the application uses RabbitMQ [5]. We opted for this product as it is open source, lightweight, easy to configure and usable in on premise or cloud mode. In this project the second mode of use was chosen using the free version for developers (Little Lemur) provided by the CloudAMQP platform.
This version is characterized by the following constraints: - Maximum 100 lines; - Maximum 10,000 messages in the queue; - Time of persistence of unused messages set at 28 days.
Furthermore, RabbitMQ provides for the possibility of installation in distributed and federated systems to guarantee high scalability and reliability. It can run on many operating systems and cloud systems and is supported by a large community of developers. You can use the system either from the command line (only in the on

premise version) or via the GUI. The GUI allows you to visually monitor the parameters related to queue performance, such as the "Message rate" measured on incoming messages and on messages read by subscribers.
In our system two queues have been implemented:

1. *input_queue*, on which the clinical data collected by the data acquisition systems (publishers) are published, consisting of input flows for the CEP (subscriber) machine;
2. *output_queue*, on which are published the complex events detected by the CEP machine (publisher) and from which they are read by the application server (subscriber).

Following is a diagram of this interaction (*Figure 5*).

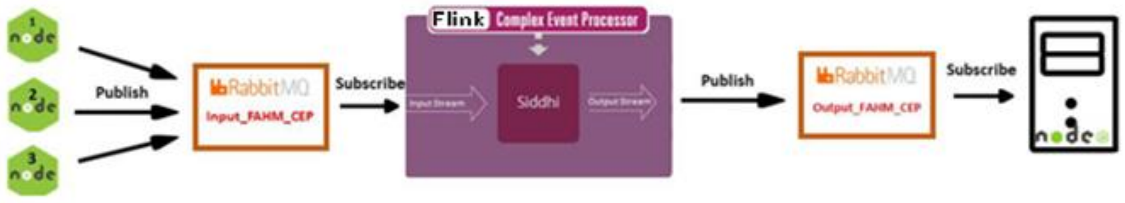


Figure 5- Interaction scheme

1.5.2. Interface system: routing, storage and event processing

The application server is written in Node.js and uses MongoDB for the storage of received events and utilities. In addition, the application uses WebSocket to send and receive real-time notifications of events published on RabbitMQ and the SMTP protocol for the transmission of alert e-mails sent by the doctor to the patient and his possible contacts registered on the platform.

The application uses Node.js [6] for the server side of the application for its event-driven asynchronous paradigm but in particular for its efficiency in cases where the network is saturated by high traffic. Node.js provides high performance by exploiting the asynchronous behavior guaranteed by JavaScript. The application also uses various frameworks and support modules, the main of which are:

- *Express*, designed to simplify the construction of web applications and in particular routing management;
- *Mongoose*, designed to simplify the use and validation of databases in MongoDB, on the basis of data modeling schemes;
- *Nodemailer*, a module useful for sending emails in Node.js.

MongoDB [7] is a non-relational DBMS oriented to JSON-style documents, of the NO-SQL type. It is a highly reliable database, supported by the most popular programming languages and which supports even complex queries.

MongoDB's document model is simple for developers to learn and use, while still providing all the capabilities needed to meet the most complex requirements at any scale.
Following are summarized some of the main features of mongodb:

- MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time;
- The document model maps to the objects in your application code, making data easy to work with;
- Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze your data;
- MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use;
- MongoDB is free and open-source, published under the GNU Affero General Public License.

FAHM uses WebSocket technology to enable full-duplex communication on a single TCP connection, thus overcoming the request-driven limitation of the HTTP protocol. This was necessary for the implementation of sending real-time notifications from the application server to the clients.
The WebSocket protocol enables interaction between a web client (such as a browser) and a web server with *lower overheads*, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the client without being first requested by the client, and allowing messages to be passed back and forth while keeping the connection open. In this way, a two-way ongoing conversation can take place between the client and the server. The communications are done over TCP port number 80 (or 443 in the case of TLS-encrypted connections), which is of benefit for those environments which block non-web Internet connections using a firewall.

2. SCENARIO ANALYSIS AND IMPLEMENTATION DETAILS

The presented scenario is only an example of the potential of this system. The use of the same in the real field would require a careful analysis of the possible use cases assisted by specialists in the health sector. In this situation, the system would easily fit through the configuration of the CEP engine that deals with the processing of flows.

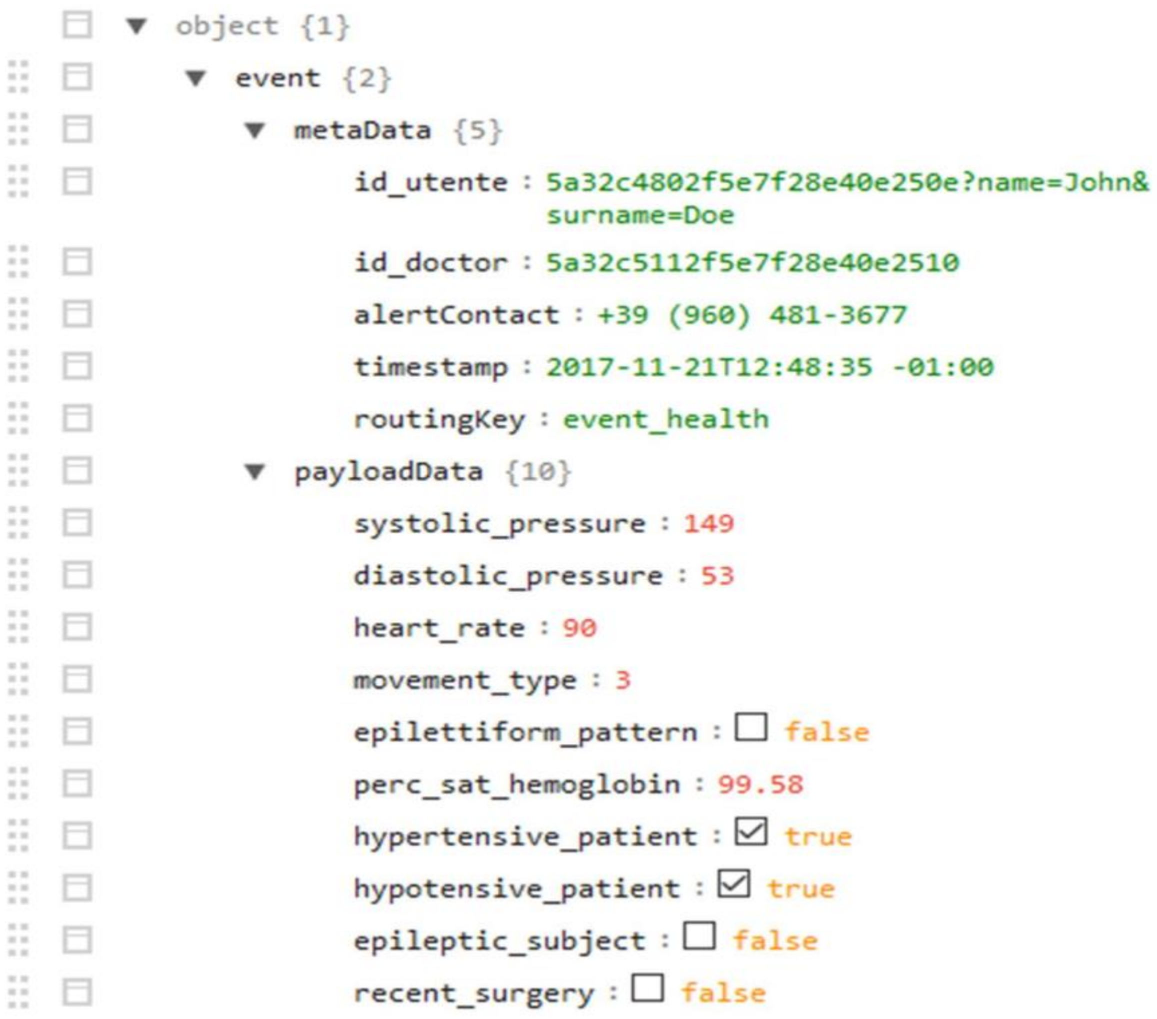
The scenario for which the prototype of the system was tested involves monitoring a limited number of patients with different clinical situations and all connected to the same type of data acquisition system.

In the scenario under consideration, complex events will be evaluated in relation to three possible dangers to the patient's health:

- Epileptic attack;
- Heart attack;
- Hypoxemia.

The procedure for acquiring data from detection systems connected to patients was simulated through the implementation of an application created with Node.js, which publishes a message on the input_queue queue of RabbitMQ at random time intervals. This Node.js application is called “send_data.js” (Appendix A) and then takes care to send the data collected by the sensors (actually only simulated as said) to the input_queue queue of the RabbitMQ instance that is specified in the "send_data.js" file itself. It is good to specify that this instance has been created on the RabbitMQ website, which simultaneously provides all the parameters and the url of the same instance in order to set these parameters correctly in the applications that use them.
When output to the CEP the output events detected by the latter will be published on the output_queue queue of RabbitMQ, and will be delivered to the application server via a code snippet inside the "app.js" (Appendix B), which works in the opposite way to send_data, providing receiving data from a RabbitMQ queue rather than sending it.

Messages published on the queues are JSON objects structured as follows:



```
object {1}
  event {2}
    metaData {5}
      id_utente : 5a32c4802f5e7f28e40e250e?name=John&
                  surname=Doe
      id_doctor : 5a32c5112f5e7f28e40e2510
      alertContact : +39 (960) 481-3677
      timestamp : 2017-11-21T12:48:35 -01:00
      routingKey : event_health
    payloadData {10}
      systolic_pressure : 149
      diastolic_pressure : 53
      heart_rate : 90
      movement_type : 3
      epilettiform_pattern : ☐ false
      perc_sat_hemoglobin : 99.58
      hypertensive_patient : ☑ true
      hypotensive_patient : ☑ true
      epileptic_subject : ☐ false
      recent_surgery : ☐ false
```

Figure 6- Listing

The messages contained within the input_queue are read by the CEP engine as information flows to be processed to detect high-level events to be published later on the output_queue.

The entire flow of operations and components involved in CEP engine part of the system is summarized in the following diagram:

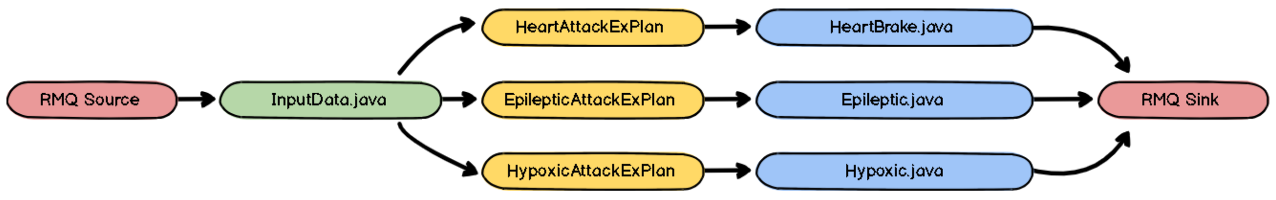


Figure 7- Flow

The individual components shown in the execution flow are described below:

- *Streaming Connectors: RabbitMQ Source and RabbitMQ Sink.*

These two components represent the Source and Sink of the CEP engine.
The basic building blocks of Flink programs are streams and transformations. Conceptually a stream is a (potentially never-ending) flow of data records, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result.
When executed, Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators. Each dataflow starts with one or more ***sources*** and ends in one or more ***sinks***.
A few basic data sources and sinks are built into Flink and are always available.
However, Connectors provide code for interfacing with various third-party systems. Currently these systems are supported:

- Apache Kafka (source/sink)
- Apache Cassandra (sink)
- Amazon Kinesis Streams (source/sink)
- Elasticsearch (sink)
- Hadoop FileSystem (sink)
- RabbitMQ (source/sink)
- Apache NiFi (source/sink)
- Twitter Streaming API (source)

In particular, we used RabbitMQ Connector (Source and Sink) in order to provide access to data streams from RabbitMQ.
To use this connector, it is necessary to add the following dependency to the "pom.xml" file of the project:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-rabbitmq_2.11</artifactId>
  <version>1.4.2</version>
</dependency>
```

The Source Connector provides a RMQSource class to consume messages from a RabbitMQ queue. This source provides three different levels of guarantees, depending on how it is configured with Flink:

1. **Exactly-once**: means that for each message handed to the mechanism exactly one delivery is made to the recipient; the message can neither be lost nor duplicated. In order to achieve exactly-once guarantees with the RabbitMQ source, the following is required:

- Enable checkpointing: With checkpointing enabled, messages are only acknowledged (hence, removed from the RabbitMQ queue) when checkpoints are completed.
- Use correlation ids: The correlation id is used by the source to deduplicate any messages that have been reprocessed when restoring from a checkpoint.
- Non-parallel source: The source must be non-parallel (parallelism set to 1) in order to achieve exactly-once. This limitation is mainly due to RabbitMQ's approach to dispatching messages from a single queue to multiple consumers.

2. **At-least-once**: When checkpointing is enabled, but correlation ids are not used or the source is parallel, the source only provides at-least-once guarantees. It means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; in more casual terms this means that messages may be duplicated but not lost.
3. **No guarantee (at-most-once)**: If checkpointing isn't enabled, the source does not have any strong delivery guarantees. Under this setting, instead of collaborating with Flink's checkpointing, messages will be automatically acknowledged once the source receives and processes them. means that for each message handed to the mechanism, that message is delivered zero or one times; again, in more casual terms it means that messages may be lost.

The third one is the cheapest—highest performance, least implementation overhead—because it can be done in a fire-and-forget fashion without keeping state at the sending end or in the transport mechanism. The second one requires retries to counter transport losses, which means keeping state at the sending end and having an acknowledgement mechanism at the receiving end. The first is most expensive—and has consequently worst performance—because in addition to the second it requires state to be kept at the receiving end in order to filter out duplicate deliveries. In this project, we focused on implementing the third guarantee in StreamingJob.java file (Appendix C). This choice comes from the use of RabbitMQ as a connector. In fact, in this way there is no possibility of data loss due to a problem related to the CEP engine (i.e. packet loss and lack of ack). This happens because it is the CEP engine that requests the data stream from the connector, and not vice versa. Moreover, as mentioned in paragraph 1.2, since AMQP is based on TCP, the resending of packets in case of loss of the same is automatically managed, both in communication to RabbitMQ, and outgoing from it. In addition, choosing the third option maximizes performance.

Finally, the Sink Connector provides a RMQSink class for sending messages to a RabbitMQ queue.

It is good to specify that both the RMQSource class and the RMQSink class make use of an appropriate RMQConnectionConfig object that allows to set the connection to the RabbitMQ instance used to obtain and publish the input and output datastream respectively.

- *InputData.java* (Appendix F).

This class represents a POJO object that collects the various JSON data that are taken from the RabbitMQ queue and that arrive in input to the CEP engine. As you can see from the code of the StreamingJob.java class (Appendix C), the CEP engine will perform a DataStream transformation using a "map" function. In particular, the input stream that contains string (JSON) values will immediately be converted into a stream containing values of type InputFAHM (Appendix G).
Going more specifically in the analysis of the code in question, we can see how this happens. First of all, it is worth pointing out that an instance of the ObjectMapper object is used, which allows us to serialize Java objects into JSON (with the writeValue method) and deserialize JSON into Java objects (with the readValue method). To use this object, however, you must enter the following dependencies within the pom.xml file in the project directory:

```
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>2.9.4</version>
</dependency>
```

After that, we will use the readValue method to transform input JSON strings into objects of type InputData, that is, POJO classes. After that, the object of type InputData will be further converted into another class called InputFAHM, using its constructor. This intermediate class simply serves to obtain a format of the attributes present in the JSON that can be used for processing by the CEP engine.
As a result, once this new DataStream containing items of the InputFAHM type is obtained, the CEP engine can proceed to the next step in the workflow shown in the previous figure.

- *Execution Plans* (Appendix D).

Three different execution plans have been implemented. By execution plan, in the CEP domain, we mean the set of operations to be performed on the data streams. For each event received, the following execution plans will be launched:

- **EpilepticAttackExPlan** (Appendix D.1) This execution plan checks if in the last 10 events related to a user subject to epileptic seizures (connected to an electroencephalograph equipped with an artificial intelligence system for the detection of the epileptiform pattern) 8 or more anomalous measurements have been verified indicating the possibility of a attack in progress.

- **HeartAttackExPlan** (Appendix D.2) This execution plan, for a hypertensive subject, measures the mean value of the last ten systolic blood pressure readings: if this value is greater than a certain limit threshold and is increasing (there is a sequence of at least two measurements with increasing values) then the complex event associated with a possible heart attack is detected. It is important to note that this execution plan is applied to all patients, even to non-hypertensive subjects. In this case, the complex event will be generated only when more critical surveys occur.
- **HypoxicAttackExPlan** (Appendix D.3) This execution plan has been mainly implemented to monitor patients who have undergone surgery and whose remote monitoring is necessary, since even in the presence of a stable clinical picture it is important to check the percentage of oxygen saturated hemoglobin with respect to the total amount of hemoglobin present in the blood, to identify a possible crisis of respiratory failure. From this point of view, it is checked whether in the last 10 seconds there is a drop in the percentage of saturated hemoglobin in the blood and if there is a value that represents an imminent danger for the patient's life. These controls, with more loose alert margins, are also performed on patients on whom a pulse oximeter has been installed and who have not undergone surgery in the last period.

At the implementation level, *Siddhiql* was used, with which it was possible to define the aforementioned queries.
In the *StreamingJob* class, starting from the stream containing InputFAHM objects defined in the previous step, it was possible to logically partition the latter into disjointed partitions thanks to the keyBy function: all records with the same key are assigned to the same partition, where the key in this case will be "meta_id_utente". Then, as already mentioned above, since Siddhi is not natively implemented in Apache Flink, the SiddhiCEP library was used. The latter uses first the "define" method to define the input stream (operation that was possible in WSO2 in the .siddhiql files), after which the "cql" method is used to define the query corresponding to one of the above execution plan files passed as a parameter after being converted into a string. Finally, the returnAsMap method is used to insert the result of the query (defined by the corresponding execution plan) as a Map object into the output datastream.

- *Event Stream: Output*

In the particular scenario, three types of output flows have been designed, one for each execution plan.

- Epileptic.java (Appendix E.1)
- HeartBrake.java (Appendix E.2)

- Hypoxic.java (Appendix E.3)

It is good to underline that all output flows have common characteristics. Each message has two sections:

- metaData: containing information useful to the system;
- payloadData: containing fields specific to the type of stream analyzed and fixed fields related to the monitoring of the cardiovascular system.

From an implementation point of view, starting from the previous step, the opposite operation will be carried out with respect to what has been done on the input datastream.
In particular, again in the StreamingJob class, from the datastream of Map objects obtained previously as a result of the query, a datastream will be obtained (through a datastream transformation) containing objects of type String. Also in this case the ObjectMapper class was used to obtain first a java class (Epileptic.java, Hypoxic.java, HeartBrake.java) and then, using the writeValueAsString method of the ObjectMapper class, a JSON string. This string will then be inserted into the output datastream, and will represent a complex event (processed by the CEP engine thanks to the queries seen above) that will be sent to the sink and then sent to the output_queue queue of rabbitMQ, which in turn will send the data to the Application Server. All potentially dangerous events will then be read by the Application Server which will manage the storage and the possibility of reading by the users registered in the system, as well as prepare the sending of alert mail to the doctors via Nodemailer, a module useful for sending emails in Node.js.

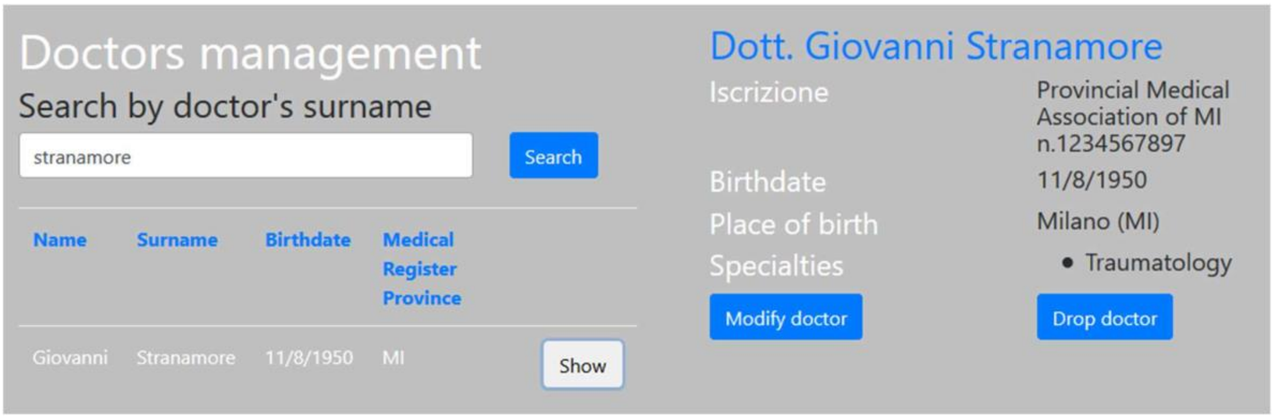
3. USER INTERFACE

The web application provides for the use of three types of users:

1. Administrator
2. Patient
3. Doctor

The *Administrator*, whose profile was created during development, can register the profiles of the doctors and modify the parameters (specifically, specializations and the provincial register of belonging).

The following figures show screenshots of the administration page:



Doctors management

Search by doctor's surname

stranamore

Search

Name	Surname	Birthdate	Medical Register Province	
Giovanni	Stranamore	11/8/1950	MI	Show

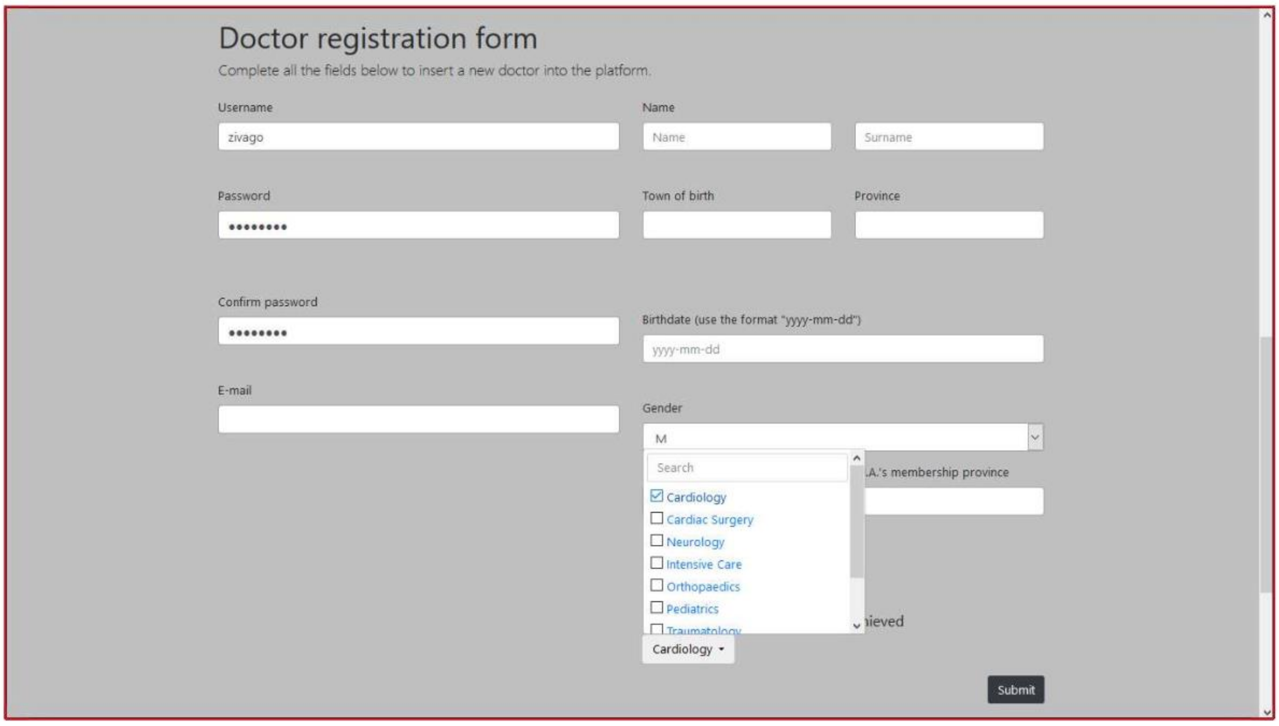
Dott. Giovanni Stranamore

Iscrizione	Provincial Medical Association of MI n.1234567897
Birthdate	11/8/1950
Place of birth	Milano (MI)
Specialties	• Traumatology

Modify doctor

Drop doctor

Figure 8 - [ADMIN] Research function of a doctor



Doctor registration form

Complete all the fields below to insert a new doctor into the platform.

Username: zivago

Name: Name | Surname

Password: ••••••••

Town of birth: | Province:

Confirm password: ••••••••

Birthdate (use the format "yyyy-mm-dd"): yyyy-mm-dd

E-mail:

Gender: M

.A.'s membership province

Search

- [✓] Cardiology
- [ ] Cardiac Surgery
- [ ] Neurology
- [ ] Intensive Care
- [ ] Orthopaedics
- [ ] Pediatrics
- [ ] Traumatology

ieved

Cardiology

Submit

Figure 9 - [ADMIN] Registration function of a doctor

The *Patient*, who, as a guest user, can access the Fast Alert Health Monitoring homepage and register with the platform, in order to have access to the chosen credentials.

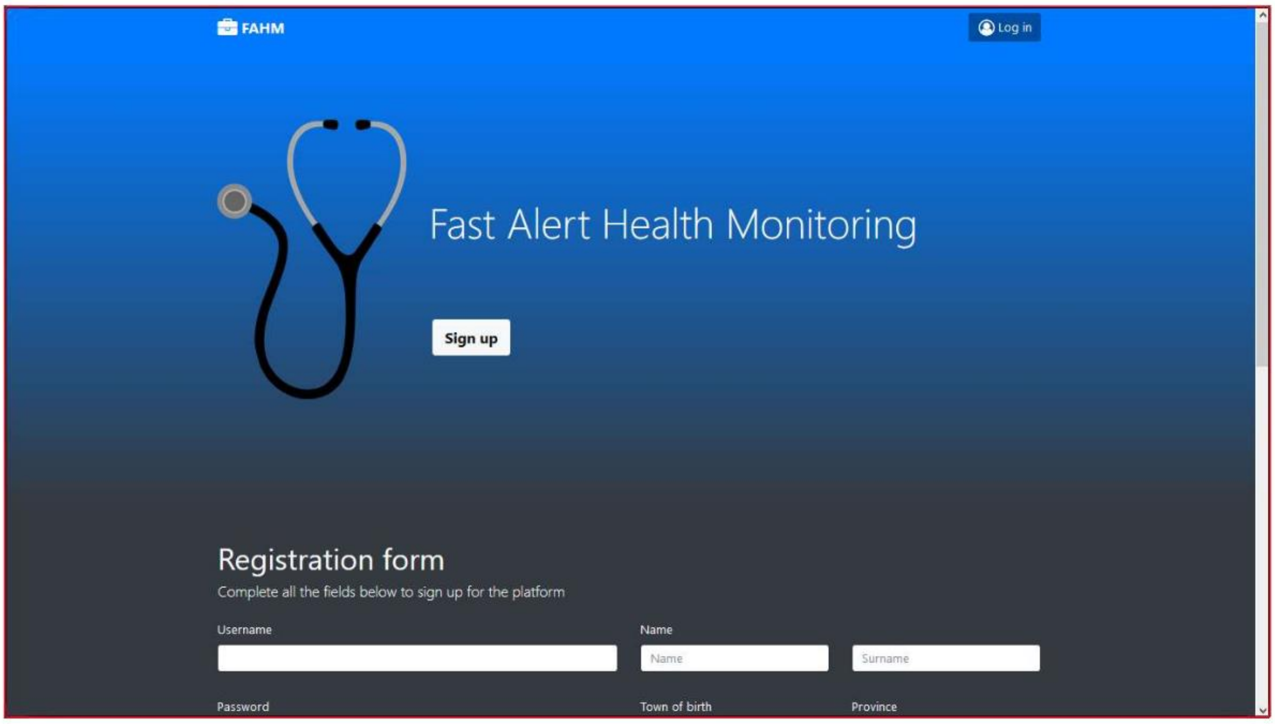


Figure 8- FAHM homepage

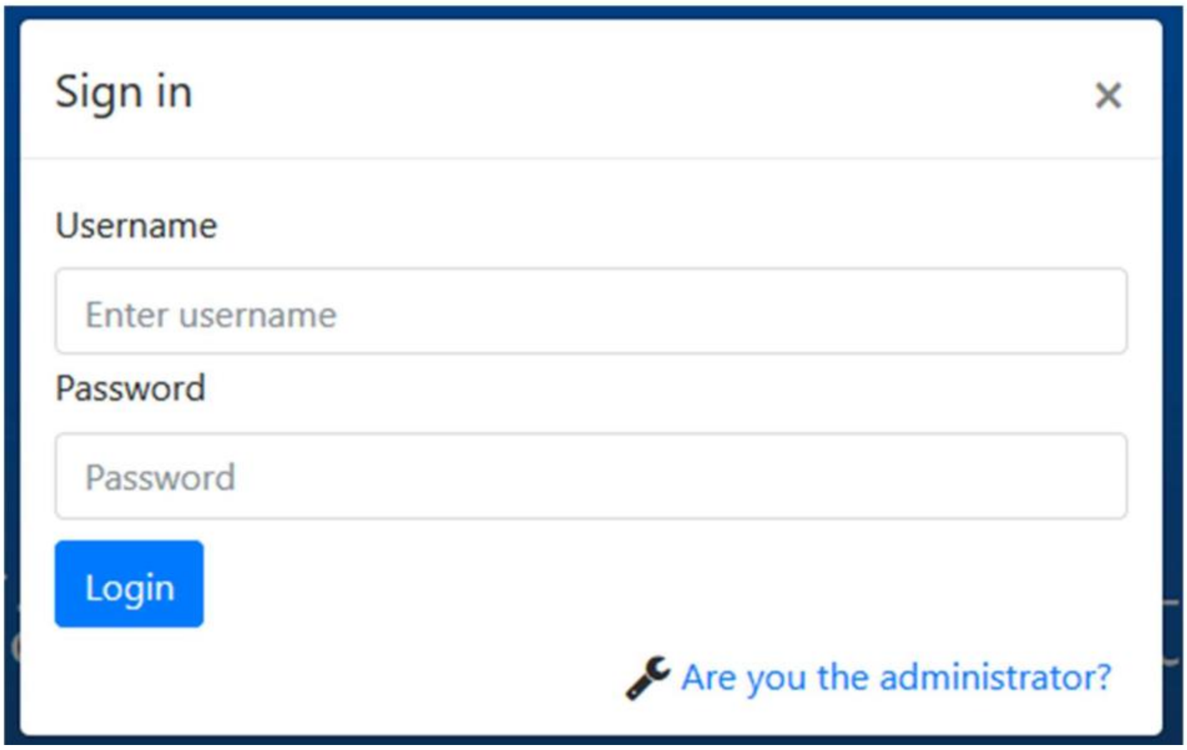
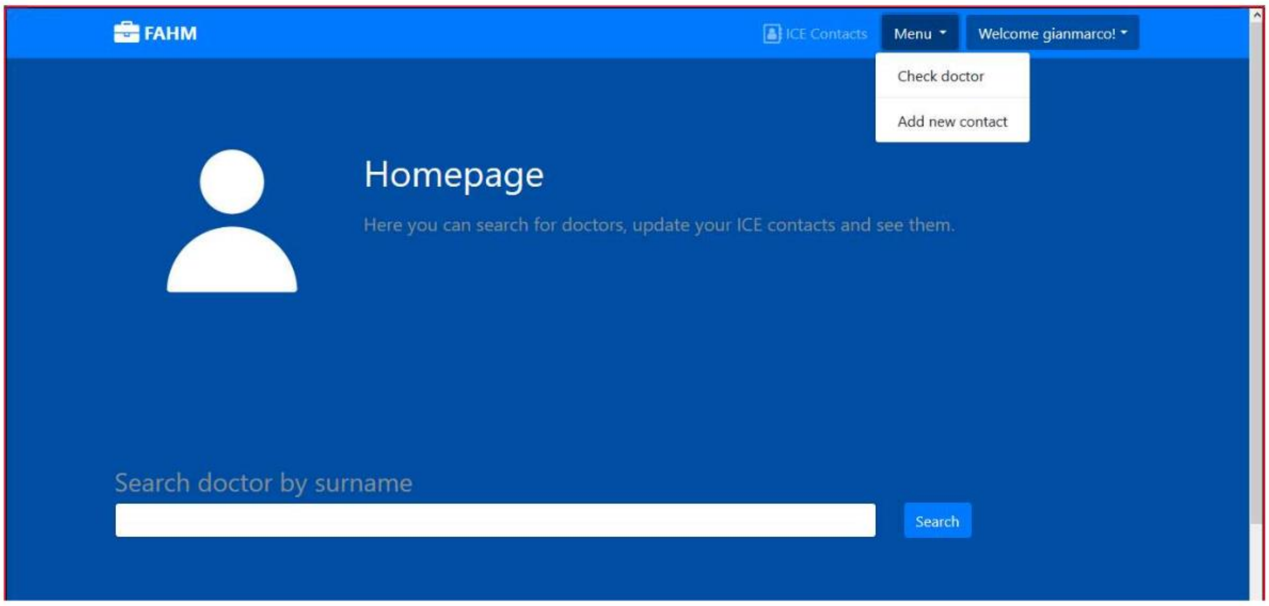


Figure 9- Login form

After authenticating, the "patient" client accesses his main page (Figure 12) from which he can search and view the doctors registered on the platform, view his profile, register his ICE contacts to the platform (In Case of Emergency, Figure 13) and display them.

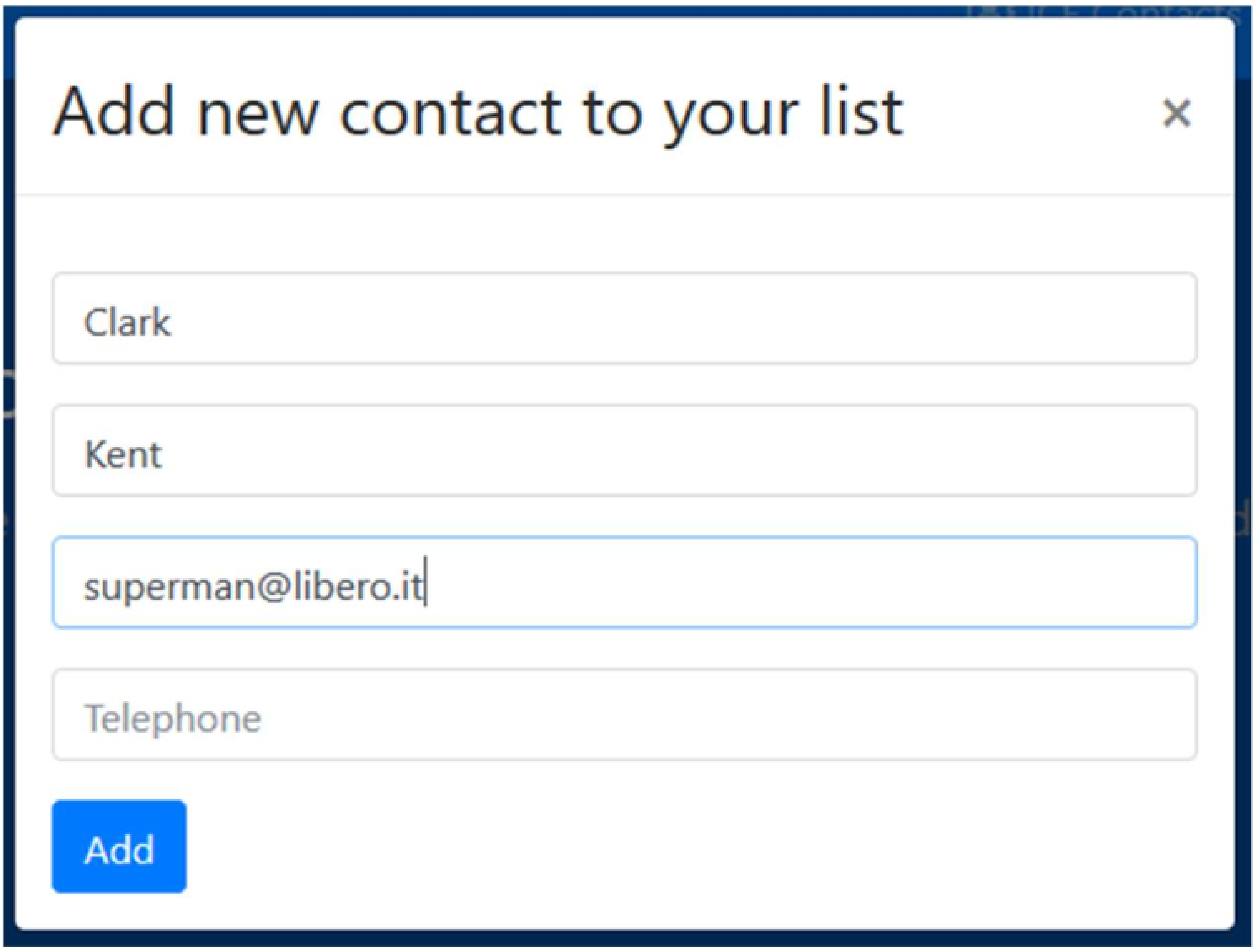


Homepage

Here you can search for doctors, update your ICE contacts and see them.

Search doctor by surname

Figure 11- [PATIENT] Main page



Add new contact to your list

Clark

Kent

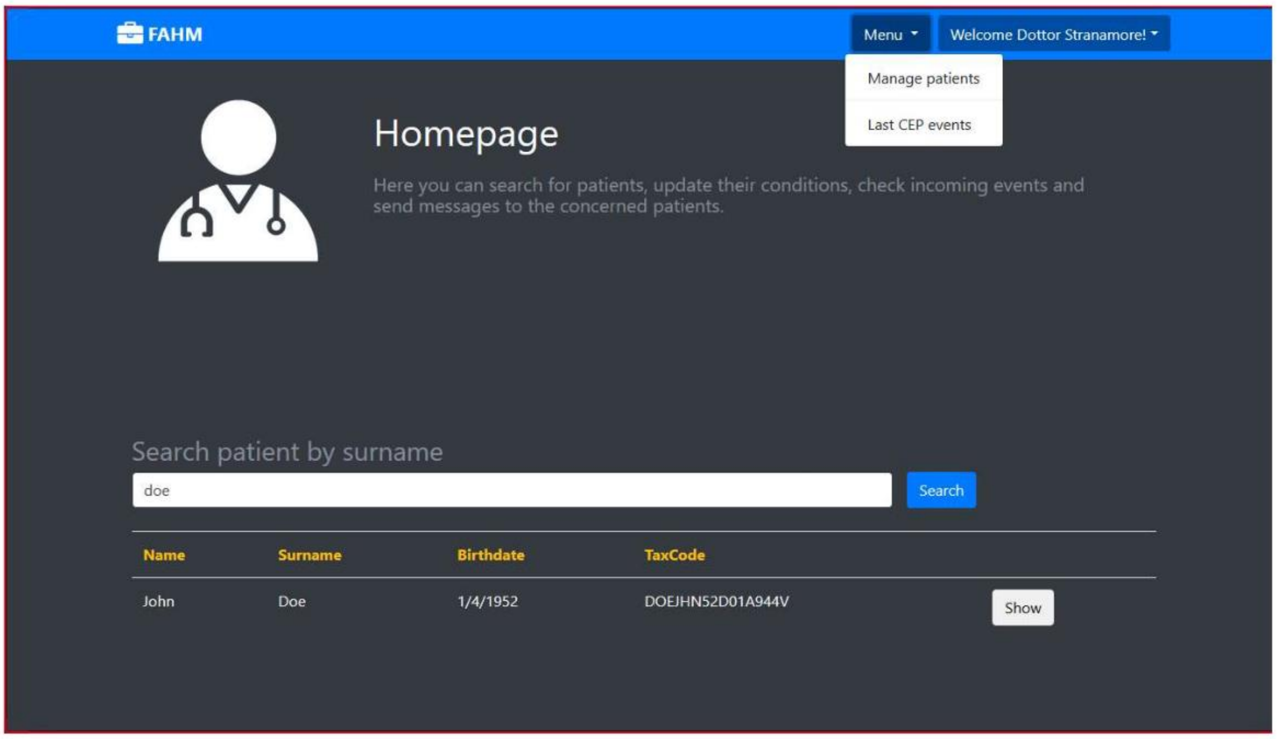
superman@libero.it

Telephone

Figure 10- [PATIENT] ICE contact insertion form

With regard to the third type of user, the *Doctor*, this will be able to log into the platform with the credentials given to him by the system administrator who registered him. Once logged in, this type of client is redirected to its main page, different from that of the "patient" client. Through this page (Figure 14) the user has access to various functions:

- View your personal profile
- logout
- search for patient profile[3] by surname and display of related information (Figure 15)
- modification of this information by adding / deleting pathologies from their medical history (Figure 16).



Homepage

Here you can search for patients, update their conditions, check incoming events and send messages to the concerned patients.

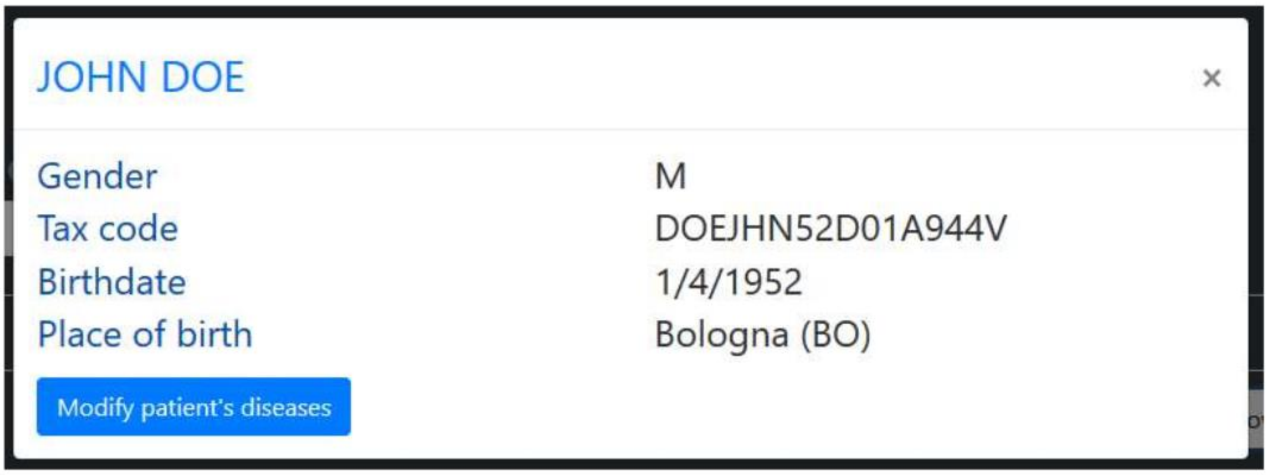
Search patient by surname

doe

Name	Surname	Birthdate	TaxCode	
John	Doe	1/4/1952	DOEJHN52D01A944V	Show

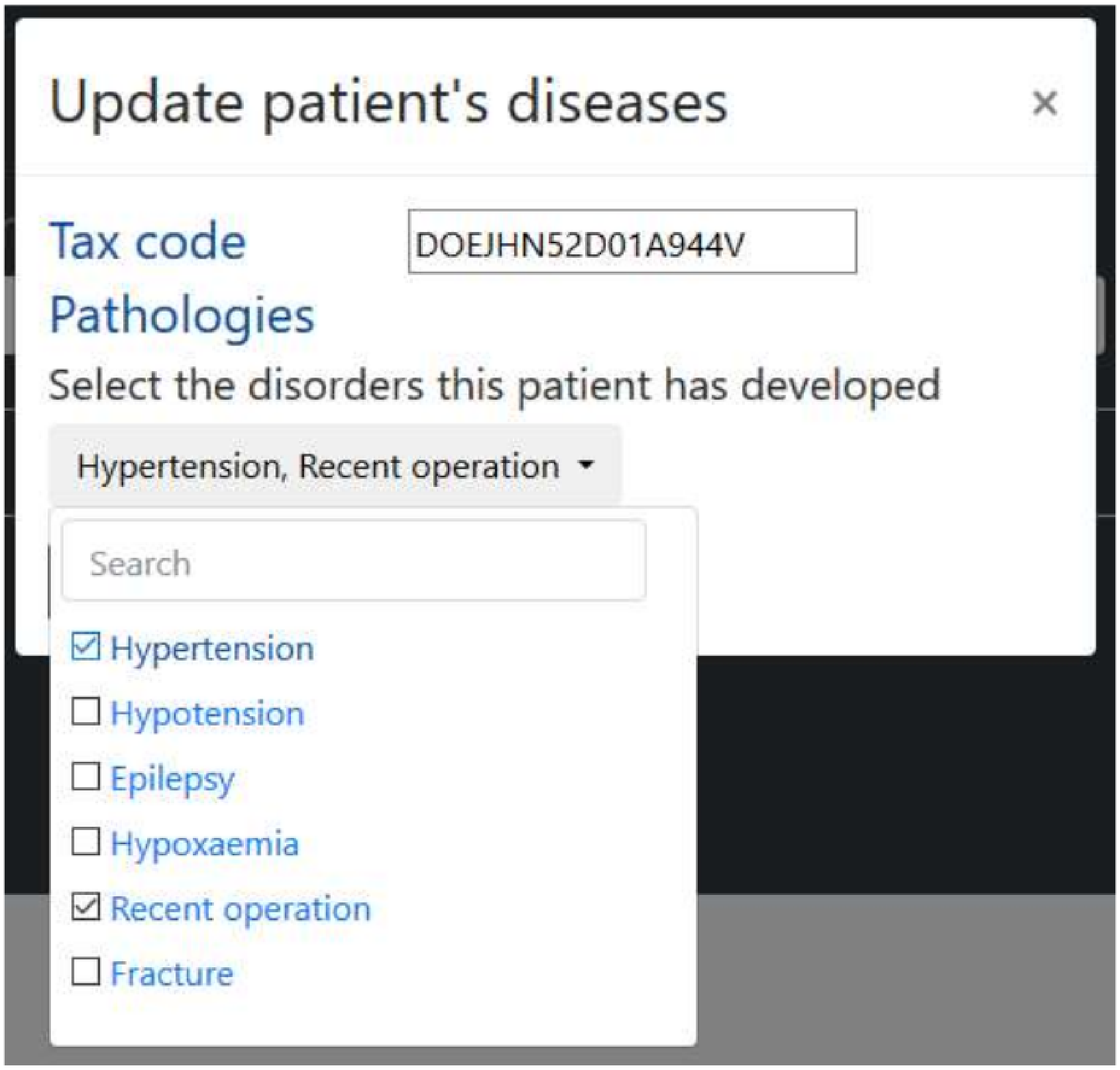
Figure 12- [DOCTOR] Main page

[3] In our current scenario hypothesis, the platform is used by doctors practicing all in the same medical office and patients are the people in the same study. For this reason, doctors can view the information of all patients registered on the platform. However, the implementation of a bond that links doctor and patient is trivial and can easily allow to restrict the possibility of research by a doctor to his own patients


JOHN DOE

Gender	M
Tax code	DOEJHN52D01A944V
Birthdate	1/4/1952
Place of birth	Bologna (BO)

Figure 14 – [DOCTOR] Patient profile display


Update patient's diseases

Tax code DOEJHN52D01A944V

Pathologies

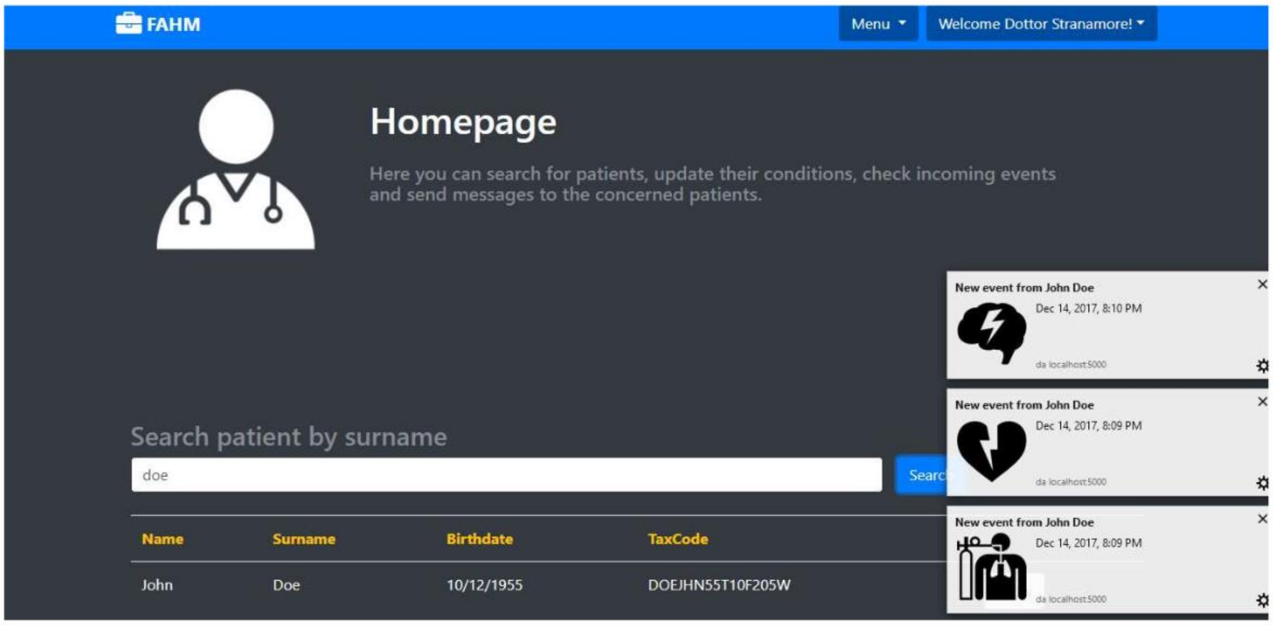
Select the disorders this patient has developed

Hypertension, Recent operation

- [✓ Hypertension]
- Hypotension
- Epilepsy
- Hypoxaemia
- [✓ Recent operation]
- Fracture

Figure 13- [DOCTOR] Patient pathology update

The main function reserved for the doctor is linked to real-time notifications of potentially dangerous events detected by the CEP engine.
These events, once arrived on the application server, are stored in a collection in order to be able to subsequently read them through a page of analysis of the message history, and are also immediately notified on video on the main page of the doctor.
Clicking on the notification (Figure 17) the doctor will be able to interact with it by displaying the data and sending an email that will be sent to the interested patient and his ICE contacts.



Homepage

Here you can search for patients, update their conditions, check incoming events and send messages to the concerned patients.

Search patient by surname

doe

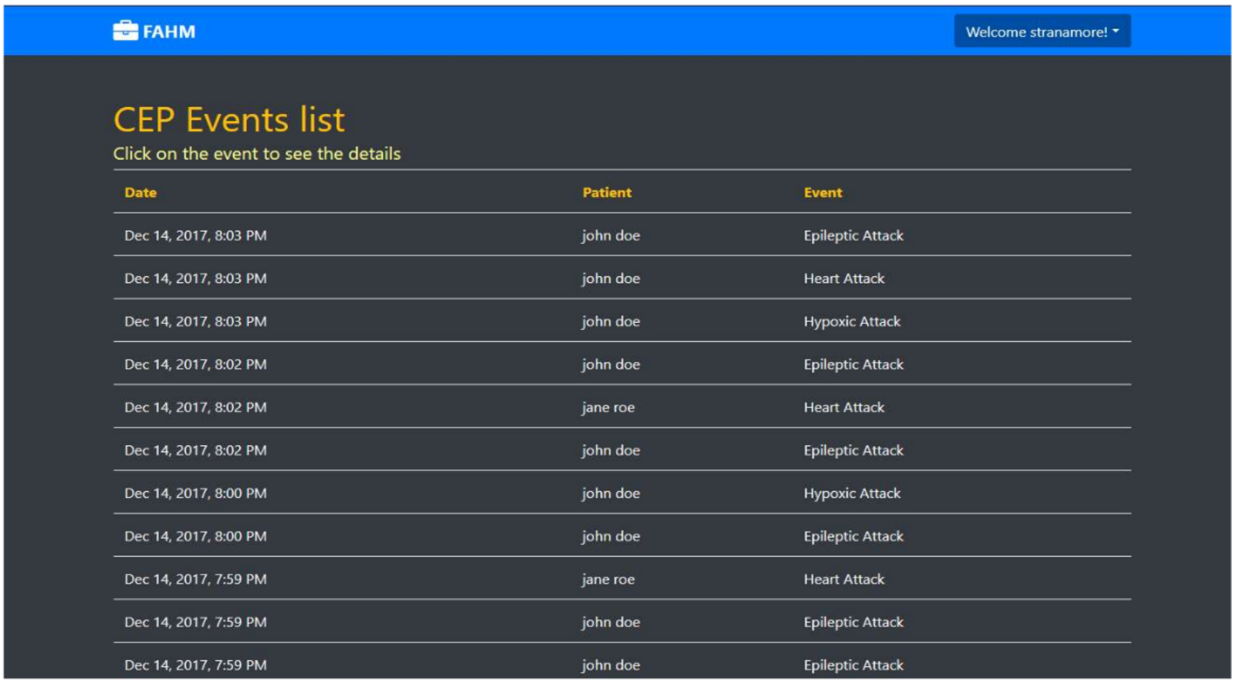
Name	Surname	Birthdate	TaxCode
John	Doe	10/12/1955	DOEJHN55T10F205W

New event from John Doe
Dec 14, 2017, 8:10 PM
da localhost:5000

New event from John Doe
Dec 14, 2017, 8:09 PM
da localhost:5000

New event from John Doe
Dec 14, 2017, 8:09 PM
da localhost:5000

Figure 15 – [DOCTOR] Notifications

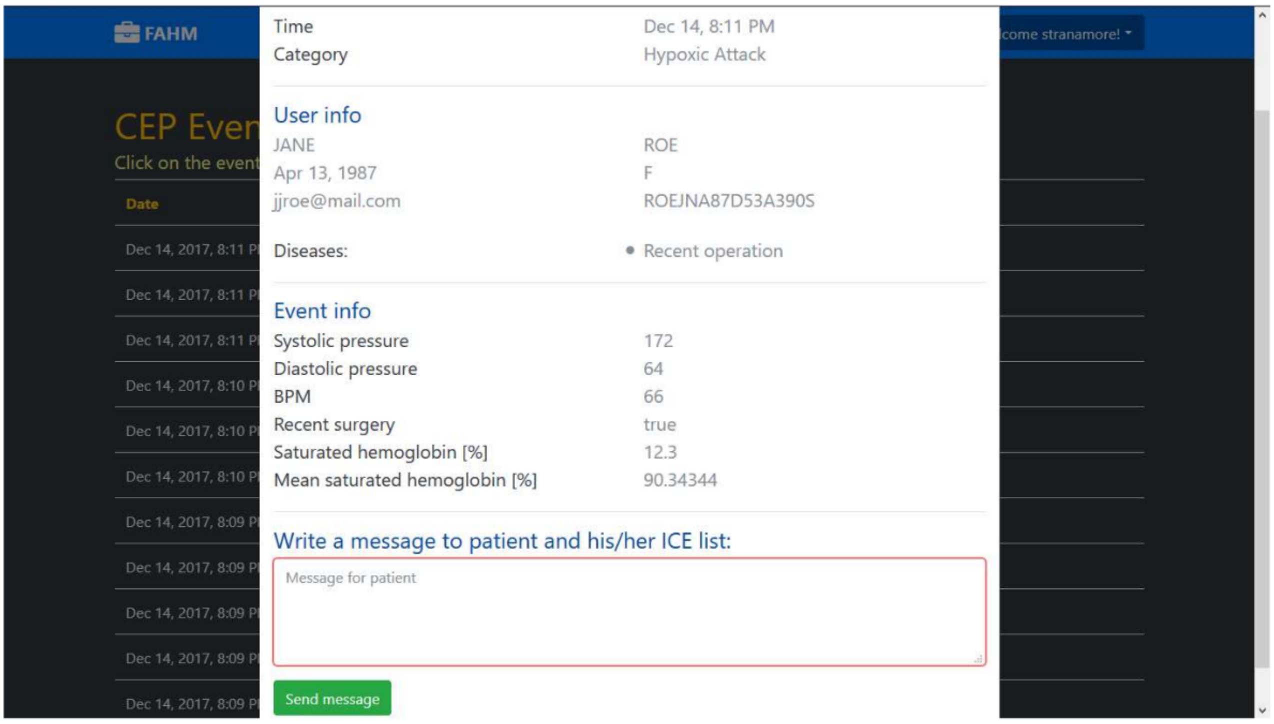


CEP Events list

Click on the event to see the details

Date	Patient	Event
Dec 14, 2017, 8:03 PM	john doe	Epileptic Attack
Dec 14, 2017, 8:03 PM	john doe	Heart Attack
Dec 14, 2017, 8:03 PM	john doe	Hypoxic Attack
Dec 14, 2017, 8:02 PM	john doe	Epileptic Attack
Dec 14, 2017, 8:02 PM	jane roe	Heart Attack
Dec 14, 2017, 8:02 PM	john doe	Epileptic Attack
Dec 14, 2017, 8:00 PM	john doe	Hypoxic Attack
Dec 14, 2017, 8:00 PM	john doe	Epileptic Attack
Dec 14, 2017, 7:59 PM	jane roe	Heart Attack
Dec 14, 2017, 7:59 PM	john doe	Epileptic Attack
Dec 14, 2017, 7:59 PM	john doe	Epileptic Attack

Figure 18 – [DOCTOR] CEP events display



Time	Dec 14, 8:11 PM
Category	Hypoxic Attack

User info

JANE	ROE
Apr 13, 1987	F
jjroe@mail.com	ROEJNA87D53A390S
Diseases:	• Recent operation

Event info

Systolic pressure	172
Diastolic pressure	64
BPM	66
Recent surgery	true
Saturated hemoglobin [%]	12.3
Mean saturated hemoglobin [%]	90.34344

Write a message to patient and his/her ICE list:

Figure 19 - [DOCTOR] Event interaction

CONCLUSIONS AND FUTURE DEVELOPMENTS

The platform, with the changes made, has significantly improved performance in terms of speed: this is mainly due to the replacement of WSO2 with Apache Flink as a CEP engine, which is definitely more performing. At the same time, with the implementation of the TLS protocol, the site now runs on https and consequently a considerable increase in security and reliability in the management of data exchanged with the Application Server has been achieved.

The application currently does not provide a real-time communication between patients and doctors, which may be necessary in case the doctor needs a quick interaction with the patient. In this context it might be necessary to implement a chat within which it would be advisable to introduce a Liquid component to allow the use of the conversation from multiple devices at the same time.
Another fundamental aspect is the verification of operation with real acquisition systems (remember that, at the moment, complex events are only simulated by the CEP engine).
Another possible improvement of the platform has been identified in the enabling of the Apache Flink *checkpointing* function, which would allow to manage a possible fault tolerance strategy: checkpoints allow Flink to recover state and positions in the streams to give the application the <u>same</u> semantics as a failure-free execution.

REFERENCES

[1] G. Cugola, A. Margara , "Processing flows of information: From data stream to complex event processing", ACM Computing Surveys (CSUR) 44 (3), 15

[2] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate and P. Manzoni, "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks," 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, 2015, pp. 931- 936, doi: 0.1109/CCNC.2015.7158101

[3] https://wso2.github.io/siddhi/

[4] https://ci.apache.org/projects/flink/flink-docs-release-1.4/

[5] https://www.rabbitmq.com/documentation.html

[6] https://nodejs.org/it/docs/

[7] https://docs.mongodb.com

Appendix

A. send_data.js

```
var amqp = require('amqplib/callback_api');
var fs = require("fs");
var amqpConn = null;
var q = "input_queue";

var dati = caricaJSON();
function caricaJSON(){
		return JSON.parse(fs.readFileSync("./JSON_EXAMPLE.json"));
}

function start() {
		amqp.connect('amqp://nizuzoyu:R-
9eFbEOoHtTyPKU26fy4rjLMaZgjbgy@duckbill.rmq.cloudamqp.com/nizuzoyu', function(err, conn) {
			if (err) {
				console.error("[AMQP]", err.message);
				return setTimeout(start, 1000);
			}
			conn.on("error", function(err) {
				if (err.message !== "Connection closing") {
					console.error("[AMQP] conn error", err.message);
				}
			});
			conn.on("close", function() {
				console.log("[AMQP] Connection closed");
				setTimeout(function() { process.exit(0) }, 500);
			});

			console.log("[AMQP] connected");
			amqpConn = conn;
			startPublisher();

		});
}

var pubChannel = null;
var offlinePubQueue = [];
function startPublisher() {
    amqpConn.createConfirmChannel(function(err, ch) {


			if (closeOnErr(err)) return;

			ch.on("error", function(err) {
				console.error("[AMQP] channel error", err.message);
			});

			ch.on("close", function() {
				console.log("[AMQP] channel closed");
			});

			pubChannel = ch;

			ch.assertQueue(q, {durable: true});

			while( (m = offlinePubQueue.shift()) !== undefined ) {
```

```
                        publish("", q, m[2]);
                }
   });
}

function publish(exchange, routingKey, content) {
   try {
     var d = new Date();
     pubChannel.publish(exchange, routingKey, content, {
            contentType:"text/plain",
            contentEncoding :"ISO-8859-1",
            priority:1,
            timestamp:d.getTime(),
            userId :"nizuzoyu",
            appId :"applicationId",
            persistent: true },
      function(err, ok) {
            if (err) {
                        console.error("[AMQP] publish", err);
                        offlinePubQueue.push([exchange, routingKey, content]);
                        pubChannel.connection.close();
            }
      });
   } catch (e) {
     console.error("[AMQP] publish", e.message);
     offlinePubQueue.push([exchange, routingKey, content]);
   }
}

function closeOnErr(err) {
   if (!err) return false;
   console.error("[AMQP] error", err);
   amqpConn.close();
   return true;
}

var i = 0, length = dati.length;
setInterval(function() {
            if(i<length){
                        dati[i].event.metaData.timestamp = new Date();
                        publish("", q, Buffer.from(JSON.stringify(dati[i])));
                        console.log("Message " + i + " of "+ length + " time: " + dati[i].event.metaData.timestamp);
                        i++;
            }else{
                        amqpConn.close();
            }
}, (Math.floor(Math.random() * 9) + 1 ) *500);

start();
```

B. app.js (receiving data from the output_queue)

```
. . .

amqp.connect("amqp://nizuzoyu:9eFbEOoHtTyPKU26fy4rjLMaZgjbgy@duckbill.rmq.cloudamqp.com/nizuzoyu").then(
function(conn) {
  process.once('SIGINT', function() { conn.close(); });
  return conn.createChannel().then(function(ch) {

    var ok = ch.assertQueue('output_queue', {durable: true});

    ok = ok.then(function(_qok) {
      return ch.consume('output_queue', function(msg) {
        console.log(" [x] Received '%s'", msg.content.toString());
  var obj = JSON.parse(msg.content.toString());
  console.log(" [x] Received ID '%s'", obj.event.metaData.id_utente);

. . .
```

C. StreamingJob.java

```
public class StreamingJob {

	public static void main(String[] args) throws Exception {

	// set up the streaming execution environment
	final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

	final RMQConnectionConfig connectionConfig = new RMQConnectionConfig.Builder()

	    .setUri("amqp://nizuzoyu:R9eFbEOoHtTyPKU26fy4rjLMaZgjbgy@duckbill.rmq.cloudamqp.com/nizuzoyu")
	    .build();


//DATAINPUT
	DataStream<InputFAHM> input=env.addSource(
		new RMQSource<String>(connectionConfig,"input_queue",true,
		new SimpleStringSchema()))
	.map(new MapFunction<String,InputFAHM>() {
		@Override
		public InputFAHM map(String value) throws Exception {
			ObjectMapper mapper2=new ObjectMapper();
			InputData out=mapper2.readValue(value,InputData.class);
			InputFAHM da=new InputFAHM(out);
			return da;
		}
	}).name("INPUT");
//QUERYS

	InputStream in_epi=new StreamingJob().getClass().getResourceAsStream("/EpilepticAttackExPlan");
	BufferedReader reader_epi = new BufferedReader(new InputStreamReader(in_epi));
	StringBuilder sb_epi = new StringBuilder();
	for (String line; ( line = reader_epi.readLine()) != null;) {
		sb_epi.append(line);
	}
	InputStream in_hea=new StreamingJob().getClass().getResourceAsStream("/HeartAttackExPlan");
	BufferedReader reader_hea = new BufferedReader(new InputStreamReader(in_hea));
```

```
		StringBuilder sb_hea = new StringBuilder();
		for (String line; ( line = reader_hea.readLine()) != null;) {
			sb_hea.append(line);
		}
		InputStream in_hyp=new StreamingJob().getClass().getResourceAsStream("/HypoxicAttackExPlan");
		BufferedReader reader_hyp = new BufferedReader(new InputStreamReader(in_hyp));
		StringBuilder sb_hyp = new StringBuilder();
		for (String line; ( line = reader_hyp.readLine()) != null;) {
			sb_hyp.append(line);
		}

		String EpilepticAttack=sb_epi.toString();
		String HeartAttack=sb_hea.toString();
		String HypoxicAttack=sb_hyp.toString();

		DataStream<InputFAHM> id_input=input.keyBy("meta_id_utente");

//QUERYES

		DataStream<Map<String,Object>>output_epi=SiddhiCEP.define("InputStream",id_input,"meta_id_utente",
"meta_id_doctor","meta_alertContact","meta_timestamp","meta_routingKey","epileptic_subject","recent_surgery",
"systolic_pressure","diastolic_pressure","heart_rate","movement_type",
"epilettiform_pattern","perc_sat_hemoglobin","hypertensive_patient","hypotensive_patient")
			.cql(EpilepticAttack)
			.returnAsMap("EpilepticAttackAlert");

		DataStream<Map<String,Object>>output_hyp=SiddhiCEP.define("InputStream",id_input,"meta_id_utente",
"meta_id_doctor","meta_alertContact","meta_timestamp","meta_routingKey","epileptic_subject",
"recent_surgery","systolic_pressure","diastolic_pressure","heart_rate","movement_type",
"epilettiform_pattern","perc_sat_hemoglobin","hypertensive_patient","hypotensive_patient")
			.cql(HypoxicAttack)
			.returnAsMap("HypoxicStream");
		DataStream<Map<String,Object>>output_hea=SiddhiCEP.define("InputStream",id_input,"meta_id_utente",
"meta_id_doctor","meta_alertContact","meta_timestamp","meta_routingKey","epileptic_subject",
"recent_surgery","systolic_pressure","diastolic_pressure","heart_rate","movement_type",
"epilettiform_pattern","perc_sat_hemoglobin","hypertensive_patient","hypotensive_patient")
			.cql(HeartAttack)
			.returnAsMap("HeartBrake_AttackAlert");
//MAPPINGOUTPUT
		DataStream<String> sink_hyp = output_hyp.map(new MapFunction<Map<String, Object>, String>() {
			@Override
			public String map(Map<String, Object> map) throws Exception {
				ObjectMapper mapper2 = new ObjectMapper();

				BaseOutput base= mapper2.convertValue(map, BaseOutput.class);
				Hypoxic hyp=new Hypoxic(base);
				String string = mapper2.writeValueAsString(hyp);
				return string;
			}
		}).name("QueryMapping");

		DataStream<String> sink_hea = output_hea.map(new MapFunction<Map<String, Object>, String>() {
			@Override
			public String map(Map<String, Object> map) throws Exception {
				ObjectMapper mapper2 = new ObjectMapper();

				BaseOutput base= mapper2.convertValue(map, BaseOutput.class);
				HeartBreak hea=new HeartBreak(base);
				String string = mapper2.writeValueAsString(hea);
				return string;
			}
		}).name("QueryMapping");

		DataStream<String> sink_epi = output_epi.map(new MapFunction<Map<String, Object>, String>() {
			@Override
			public String map(Map<String, Object> map) throws Exception {
				ObjectMapper mapper2 = new ObjectMapper();

				BaseOutput base= mapper2.convertValue(map, BaseOutput.class);
				Epileptic epi=new Epileptic(base);
				String string = mapper2.writeValueAsString(epi);
				return string;
			}
		}).name("QueryMapping");

		sink_hyp.print();
		sink_epi.print();
		sink_hea.print();

//SINK
		sink_hyp.addSink(new RMQSink<String>(
			connectionConfig,		// config for the RabbitMQ connection
			"output_queue",			// name of the RabbitMQ queue to send messages to
			new SimpleStringSchema())).name("HypoxicAttack");	// serialization schema to turn Java objects to
messages
		sink_epi.addSink(new RMQSink<String>(
			connectionConfig,		// config for the RabbitMQ connection
			"output_queue",			// name of the RabbitMQ queue to send messages to
			new SimpleStringSchema())).name("EpilepticAttack");	// serialization schema to turn Java objects to
messages


		sink_hea.addSink(new RMQSink<String>(
			connectionConfig,		// config for the RabbitMQ connection
			"output_queue",			// name of the RabbitMQ queue to send messages to
			new SimpleStringSchema())).name("HeartAttack");	// serialization schema to turn Java objects to
messages

		env.execute("FAHM ");
		}
}
```

D. ExecutionPlans

D.1 EpilepticAttackExPlan

```
@Export('EpilepticEvent_Stream:1.0.0')
define stream EpilepticAttackAlert (meta_routingKey string,meta_id_utente string,meta_id_doctor string,
									meta_operation_type string,
									meta_timestamp string,
									systolic_pressure double,
									diastolic_pressure double,
									heart_rate double,
									epileptic_subject bool,
									number_of_events long);

partition with (meta_id_utente of InputStream)
```

```
begin from InputStream[epileptic_subject == true

                        and (epilettiform_pattern == true
                        or (movement_type == 3
                                    or movement_type == 4))
                                    ]#window.length(10)
            select      meta_routingKey,
                                    meta_id_utente,
                                    meta_id_doctor,
                                    'Epileptic Attack' as meta_operation_type,
                                    meta_timestamp as meta_timestamp,
                                    avg(systolic_pressure) as systolic_pressure,
                                    avg(diastolic_pressure) as diastolic_pressure,
                                    avg(heart_rate) as heart_rate,
                                    epileptic_subject as epileptic_subject ,
                                    count() as number_of_events
   group by meta_id_utente
            having number_of_events > 5
            output last every 2 events

            insert into EpilepticAttackAlert;
end;
```

D.2 HeartAttackExPlan

```
@Export('HeartBrake_EventStream:1.0.0')
define stream HeartBrake_AttackAlert (meta_routingKey string,
                                                                                                              meta_id_utente string,
                                                                                                              meta_id_doctor string,
                                                                                                              meta_operation_type string,
                                                                                                              meta_timestamp string,
                                                                                                              systolic_pressure                    double,
diastolic_pressure double,heart_rate double, hypertensive_patient bool);




partition with (meta_id_utente of InputStream )
begin

            from InputStream[hypertensive_patient == true]#window.length(20)
            select meta_id_utente,
            meta_id_doctor,
            meta_routingKey,
            meta_timestamp as meta_timestamp,
            avg(systolic_pressure) as SysMiddleValue,
            avg(diastolic_pressure) as DisMiddleValue,
            avg(heart_rate) as HRMiddleValue,
            systolic_pressure as CurrentSystolic,
            hypertensive_patient
            having SysMiddleValue > 110
            insert into #InferredStream;

            from every(e1=#InferredStream) ->
            e2=#InferredStream[(e1.SysMiddleValue + 5 ) <= SysMiddleValue]
            select e1.meta_routingKey as meta_routingKey,
                                    e1.meta_id_utente as meta_id_utente,
                                    e1.meta_id_doctor as meta_id_doctor,
                              'Heart Attack' as meta_operation_type,
                              e1.meta_timestamp as meta_timestamp,
                              e2.SysMiddleValue as systolic_pressure,
                              e2.DisMiddleValue as diastolic_pressure,
                              e2.HRMiddleValue as heart_rate,
                              e2.hypertensive_patient as hypertensive_patient
        output last every 2 events
        insert into HeartBrake_AttackAlert;

        from InputStream[hypertensive_patient == false]#window.length(10)
        select meta_id_utente,
        meta_id_doctor,
        meta_routingKey,
        meta_timestamp as meta_timestamp,
        avg(systolic_pressure) as SysMiddleValue,
        avg(diastolic_pressure) as DisMiddleValue,
        avg(heart_rate) as HRMiddleValue,
        systolic_pressure as CurrentSystolic,
        hypertensive_patient
        having SysMiddleValue > 155
        insert into #InferredStreamNoHypertensive;

        from every(e1=#InferredStreamNoHypertensive) ->
        e2=#InferredStreamNoHypertensive[(e1.SysMiddleValue + 5 )
                                                            <= SysMiddleValue]
        select e1.meta_routingKey as meta_routingKey,
                              e1.meta_id_utente as meta_id_utente,
                              e1.meta_id_doctor as meta_id_doctor,
                              'Heart Attack' as meta_operation_type,
                              e1.meta_timestamp as meta_timestamp,
                              e2.SysMiddleValue as systolic_pressure,
                              e2.DisMiddleValue as diastolic_pressure,
                              e2.HRMiddleValue as heart_rate,
                              e2.hypertensive_patient as hypertensive_patient
        output last every 4 events
        insert into HeartBrake_AttackAlert


end;
```

D.3 HypoxicAttackExPlan

```
@Export('HypoxicAttack_Stream:1.0.0')
define stream HypoxicStream (meta_routingKey string,
                                                  meta_id_utente string,
                                                  meta_id_doctor string,
                                                  meta_operation_type string,
                                                  meta_timestamp string, systolic_pressure double,
diastolic_pressure double, heart_rate double, recent_surgery bool, perc_sat_hemoglobin_last float,
perc_sat_hemoglobin_middle double);



partition with (meta_id_utente of InputStream )
begin from InputStream[recent_surgery == true]#window.time(10 sec)
```

```
            select meta_routingKey,
                                    meta_id_utente,
                                    meta_id_doctor,
                                    'Hypoxic Attack'   as meta_operation_type,
                                    meta_timestamp,
                        avg(systolic_pressure) as systolic_pressure,
                        avg(diastolic_pressure) as diastolic_pressure,
                        avg(heart_rate) as heart_rate,
                        recent_surgery,
                        perc_sat_hemoglobin as perc_sat_hemoglobin_last,
                        avg(perc_sat_hemoglobin) as perc_sat_hemoglobin_middle
            group by meta_id_utente
            having perc_sat_hemoglobin_middle < 90.0
            output last every 3 events
            insert into HypoxicStream;

from InputStream[recent_surgery == false]#window.length(20)
            select meta_routingKey,
                                    meta_id_utente,
                                    meta_id_doctor,
                                    'Hypoxic Attack'   as meta_operation_type,
                                    meta_timestamp,
                        avg(systolic_pressure) as systolic_pressure,
                        avg(diastolic_pressure) as diastolic_pressure,
                        avg(heart_rate) as heart_rate,
                        recent_surgery,
                        perc_sat_hemoglobin as perc_sat_hemoglobin_last,
                        avg(perc_sat_hemoglobin) as perc_sat_hemoglobin_middle
            group by meta_id_utente
            having perc_sat_hemoglobin_middle < 90.0
            output last every 10 events
            insert into HypoxicStream;
end;
```

E. Output flows

E.1 Epileptic.java

```
public class Epileptic {
   public class event

   {
      public event(BaseOutput out){metaData=new metaData(out);
                             payloadData=new payloadData(out);}
      private metaData  metaData;
      private payloadData payloadData;
      public class metaData{
         private String id_utente;
         private String id_doctor;
         private String operation_type;
         private String timestamp;
```

```
            private String routingKey;

public  metaData(BaseOutput out){
    setId_utente(out.getMeta_id_utente());
    setOperation_type(out.getMeta_operation_type());
    setRoutingKey(out.getMeta_routingKey());
    setTimestamp(out.getMeta_timestamp());
    setId_doctor(out.getMeta_id_doctor());
}

            public String getId_doctor() {
                return id_doctor;
            }

            public String getId_utente() {
                return id_utente;
            }

            public String getRoutingKey() {
                return routingKey;
            }

            public String getTimestamp() {
                return timestamp;
            }

            public void setId_utente(String id_utente) {
                this.id_utente = id_utente;
            }

            public String getOperation_type() {
                return operation_type;
            }

            public void setOperation_type(String operation_type) {
                this.operation_type = operation_type;
            }

            public void setId_doctor(String id_doctor) {
                this.id_doctor = id_doctor;
            }

            public void setRoutingKey(String routingKey) {
                this.routingKey = routingKey;
            }

            public void setTimestamp(String timestamp) {
                this.timestamp = timestamp;
            }

            @Override
            public String toString() {
                return super.toString();
            }
        }
        public class payloadData{
                private double systolic_pressure;
                private double diastolic_pressure;
                private double heart_rate;
                private boolean epileptic_subject;
                private long number_of_events;
            public payloadData(BaseOutput out){
                setDiastolic_pressure(out.getDiastolic_pressure());
                setSystolic_pressure(out.getSystolic_pressure());
                setHeart_rate(out.getHeart_rate());
                setEpileptic_subject(out.isEpileptic_subject());
                setNumber_of_events(out.getNumber_of_events());

            }           public long getNumber_of_events() {
                return number_of_events;
            }

            public boolean isEpileptic_subject() {
                return epileptic_subject;
            }

            public void setEpileptic_subject(boolean epileptic_subject) {
                this.epileptic_subject = epileptic_subject;
            }

            public void setNumber_of_events(long number_of_events) {
                this.number_of_events = number_of_events;
            }

            public double getDiastolic_pressure() {
                return diastolic_pressure;
                }

                public double getHeart_rate() {
                return heart_rate;
                }

                public double getSystolic_pressure() {
                return systolic_pressure;
                }

                public void setHeart_rate(double heart_rate) {
                this.heart_rate = heart_rate;
                }

                public void setSystolic_pressure(double systolic_pressure) {
                this.systolic_pressure = systolic_pressure;
                }

                public void setDiastolic_pressure(double diastolic_pressure) {
                this.diastolic_pressure = diastolic_pressure;
                }

        };

        public metaData getMetaData() {
            return metaData;
        }

        public void setMetaData(metaData metaData) {
            this.metaData = metaData;
        }
```

```
      public payloadData getPayloadData() {
         return payloadData;
      }

      public void setPayloadData(payloadData payloadData) {
         this.payloadData = payloadData;
      }

      @Override
      public String toString() {
         return super.toString();
      }
   }

   public Epileptic(BaseOutput out){event=new event(out);}
   private event event;

   public void setEvent(event event) {
      this.event = event;
   }

   public event getEvent() {
      return event;
   }

   @Override
   public String toString() {
      return super.toString();
   }
}
```

E.2 HeartBrake.java

```
public class HeartBreak {
   public class event
   {
      public event(BaseOutput out){metaData=new metaData(out);
         payloadData=new payloadData(out);}
      public class metaData{
         public  metaData(BaseOutput out){
            setId_utente(out.getMeta_id_utente());
            setOperation_type(out.getMeta_operation_type());
            setRoutingKey(out.getMeta_routingKey());
            setTimestamp(out.getMeta_timestamp());
            setId_doctor(out.getMeta_id_doctor());
         }
         private String id_utente;
         private String id_doctor;
         private String operation_type;
         private String timestamp;
         private String routingKey;

         public String getId_doctor() {
            return id_doctor;
         }

         public String getId_utente() {
            return id_utente;
```

```
    }

    public String getRoutingKey() {
        return routingKey;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setId_utente(String id_utente) {
        this.id_utente = id_utente;
    }

    public String getOperation_type() {
        return operation_type;
    }

    public void setOperation_type(String operation_type) {
        this.operation_type = operation_type;
    }

    public void setId_doctor(String id_doctor) {
        this.id_doctor = id_doctor;
    }

    public void setRoutingKey(String routingKey) {
        this.routingKey = routingKey;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
public class payloadData{
    public payloadData(BaseOutput out){
        setDiastolic_pressure(out.getDiastolic_pressure());
        setSystolic_pressure(out.getSystolic_pressure());
        setHeart_rate(out.getHeart_rate());
        setHypertensive_patient(out.isHypertensive_patient());

    }
    private double systolic_pressure;
    private double diastolic_pressure;
    private double heart_rate;

   private boolean hypertensive_patient;


    public void setHypertensive_patient(boolean hypertensive_patient) {
        this.hypertensive_patient = hypertensive_patient;
    }

    public boolean isHypertensive_patient() {
            return hypertensive_patient;
        }

        public double getDiastolic_pressure() {
            return diastolic_pressure;
        }

        public double getHeart_rate() {
            return heart_rate;
        }

        public double getSystolic_pressure() {
            return systolic_pressure;
        }

        public void setHeart_rate(double heart_rate) {
            this.heart_rate = heart_rate;
        }

        public void setSystolic_pressure(double systolic_pressure) {
            this.systolic_pressure = systolic_pressure;
        }

        public void setDiastolic_pressure(double diastolic_pressure) {
            this.diastolic_pressure = diastolic_pressure;
        }

    };
    private metaData  metaData;
    private payloadData payloadData;

    public metaData getMetaData() {
        return metaData;
    }

    public void setMetaData(metaData metaData) {
        this.metaData = metaData;
    }

    public payloadData getPayloadData() {
        return payloadData;
    }

    public void setPayloadData(payloadData payloadData) {
        this.payloadData = payloadData;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}



public HeartBreak(BaseOutput out){event=new event(out);}
private event event;

public void setEvent(event event) {
        this.event = event;
    }

    public event getEvent() {
        return event;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
```

E.3 Hypoxic.java

```
public class Hypoxic {
    public class event
    {
        public event(BaseOutput out){metaData=new metaData(out);
            payloadData=new payloadData(out);}
        public class metaData{
            public  metaData(BaseOutput out){
                setId_utente(out.getMeta_id_utente());
                setOperation_type(out.getMeta_operation_type());
                setRoutingKey(out.getMeta_routingKey());
                setTimestamp(out.getMeta_timestamp());
                setId_doctor(out.getMeta_id_doctor());
            }
            private String id_utente;
            private String id_doctor;
            private String operation_type;
            private String timestamp;
            private String routingKey;

            public String getId_doctor() {
                return id_doctor;
            }

            public String getId_utente() {
                return id_utente;
            }

            public String getRoutingKey() {
                return routingKey;
            }

            public String getTimestamp() {
                return timestamp;
            }

            public void setId_utente(String id_utente) {
                this.id_utente = id_utente;
            }

            public String getOperation_type() {
                return operation_type;
            }
    public void setOperation_type(String operation_type) {
        this.operation_type = operation_type;
    }

    public void setId_doctor(String id_doctor) {
        this.id_doctor = id_doctor;
    }

    public void setRoutingKey(String routingKey) {
        this.routingKey = routingKey;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}

public class payloadData{
    public payloadData(BaseOutput out){
        setDiastolic_pressure(out.getDiastolic_pressure());
        setSystolic_pressure(out.getSystolic_pressure());
        setHeart_rate(out.getHeart_rate());
       setRecent_surgery(out.isRecent_surgery());
       setPerc_sat_hemoglobin_last(out.getPerc_sat_hemoglobin_last());
       setPerc_sat_hemoglobin_middle(out.getPerc_sat_hemoglobin_middle());


    }
    private double systolic_pressure;
    private double diastolic_pressure;
    private double heart_rate;
    private boolean recent_surgery;
    private float perc_sat_hemoglobin_last;
    private double perc_sat_hemoglobin_middle;

    public double getPerc_sat_hemoglobin_middle() {
        return perc_sat_hemoglobin_middle;
    }

    public float getPerc_sat_hemoglobin_last() {
        return perc_sat_hemoglobin_last;
    }

    public void setRecent_surgery(boolean recent_surgery) {
        this.recent_surgery = recent_surgery;
    }

    public void setPerc_sat_hemoglobin_last(float perc_sat_hemoglobin_last) {
        this.perc_sat_hemoglobin_last = perc_sat_hemoglobin_last;
    }

    public void setPerc_sat_hemoglobin_middle(double perc_sat_hemoglobin_middle) {
        this.perc_sat_hemoglobin_middle = perc_sat_hemoglobin_middle;
    }
```

```
        public boolean isRecent_surgery() {
            return recent_surgery;
        }

        public double getDiastolic_pressure() {
            return diastolic_pressure;
        }

        public double getHeart_rate() {
            return heart_rate;
        }

        public double getSystolic_pressure() {
            return systolic_pressure;
        }

        public void setHeart_rate(double heart_rate) {
            this.heart_rate = heart_rate;
        }

        public void setSystolic_pressure(double systolic_pressure) {
            this.systolic_pressure = systolic_pressure;
        }

        public void setDiastolic_pressure(double diastolic_pressure) {
            this.diastolic_pressure = diastolic_pressure;
        }
    };

    private metaData  metaData;
    private payloadData payloadData;

    public metaData getMetaData() {
        return metaData;
    }

    public void setMetaData(metaData metaData) {
        this.metaData = metaData;
    }

    public payloadData getPayloadData() {
        return payloadData;
    }

    public void setPayloadData(payloadData payloadData) {
        this.payloadData = payloadData;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}


public Hypoxic(BaseOutput out){event=new event(out);}

private event event;
```

```
    public void setEvent(event event) {
        this.event = event;
    }

    public event getEvent() {
        return event;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
```

F. InputData.java

```
public class InputData {

    class event
    {
        class payloadData
        {
            private boolean epileptic_subject;
            private boolean recent_surgery;
            private int systolic_pressure;
            private int diastolic_pressure;
            private int heart_rate;
            private int movement_type;
            private boolean epilettiform_pattern;
            private float perc_sat_hemoglobin;
            private boolean hypertensive_patient;
            private boolean hypotensive_patient;

            public void setEpileptic_subject(boolean epileptic_subject) {
                this.epileptic_subject = epileptic_subject;
            }

            public void setRecent_surgery(boolean recent_surgery) {
                this.recent_surgery = recent_surgery;
            }

            public boolean isRecent_surgery() {
                return recent_surgery;
            }

            public boolean isEpileptic_subject() {
                return epileptic_subject;
            }

            public boolean isHypertensive_patient() {
                return hypertensive_patient;
            }

            public boolean isHypotensive_patient() {
                return hypotensive_patient;
            }
```

```
public float getPerc_sat_hemoglobin() {
  return perc_sat_hemoglobin;
}

public int getDiastolic_pressure() {
  return diastolic_pressure;
}

public int getHeart_rate() {
  return heart_rate;
}

public int getMovement_type() {
  return movement_type;
}

public int getSystolic_pressure() {
  return systolic_pressure;
}

public void setDiastolic_pressure(int diastolic_pressure) {
  this.diastolic_pressure = diastolic_pressure;
}

public void setSystolic_pressure(int systolic_pressure) {
  this.systolic_pressure = systolic_pressure;
}

public void setEpilettiform_pattern(boolean epilettiform_pattern) {
  this.epilettiform_pattern = epilettiform_pattern;
}

public void setHeart_rate(int heart_rate) {
  this.heart_rate = heart_rate;
}

public void setHypertensive_patient(boolean hypertensive_patient) {
  this.hypertensive_patient = hypertensive_patient;
}

public void setHypotensive_patient(boolean hypotensive_patient) {
  this.hypotensive_patient = hypotensive_patient;
}

public void setMovement_type(int movement_type) {
  this.movement_type = movement_type;
}

public void setPerc_sat_hemoglobin(float perc_sat_hemoglobin) {
  this.perc_sat_hemoglobin = perc_sat_hemoglobin;
}

public boolean isEpilettiform_pattern() {
  return epilettiform_pattern;
}

@Override
public String toString() {
        return super.toString();
    }

}

class metaData{
    private String id_utente;
    private String id_doctor;
    private String alertContact;
    private String timestamp;
    private String routingKey;

    public String getAlertContact() {
        return alertContact;
    }

    public String getId_doctor() {
        return id_doctor;
    }

    public String getId_utente() {
        return id_utente;
    }

    public String getRoutingKey() {
        return routingKey;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setId_utente(String id_utente) {
        this.id_utente = id_utente;
    }

    public void setAlertContact(String alertContact) {
        this.alertContact = alertContact;
    }

    public void setId_doctor(String id_doctor) {
        this.id_doctor = id_doctor;
    }

    public void setRoutingKey(String routingKey) {
        this.routingKey = routingKey;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}

private metaData  metaData;
```

```
        private payloadData payloadData;

        public metaData getMetaData() {
            return metaData;
        }

        public void setMetaData(metaData metaData) {
            this.metaData = metaData;
        }

        public payloadData getPayloadData() {
            return payloadData;
        }

        public void setPayloadData(payloadData payloadData) {
            this.payloadData = payloadData;
        }

        @Override
        public String toString() {
            return super.toString();
        }
    }
    private  event event;

    public void setEvent(event event) {
        this.event = event;
    }

    public event getEvent() {
        return event;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
```

G. InputFAHM.java

```
public class InputFAHM {

    private String meta_id_utente;
    private String meta_id_doctor;
    private String meta_alertContact;
    private String meta_timestamp;
    private String meta_routingKey;
public InputFAHM(){}
public InputFAHM(POJO pojo){
    meta_id_utente=pojo.getEvent().getMetaData().getId_utente();
    meta_id_doctor=pojo.getEvent().getMetaData().getId_doctor();
    meta_timestamp=pojo.getEvent().getMetaData().getTimestamp();
    meta_alertContact=pojo.getEvent().getMetaData().getAlertContact();
    meta_routingKey=pojo.getEvent().getMetaData().getRoutingKey();
    epileptic_subject=pojo.getEvent().getPayloadData().isEpileptic_subject();
    recent_surgery=pojo.getEvent().getPayloadData().isRecent_surgery();
```

```
   systolic_pressure=pojo.getEvent().getPayloadData().getSystolic_pressure();
   diastolic_pressure=pojo.getEvent().getPayloadData().getDiastolic_pressure();
   heart_rate=pojo.getEvent().getPayloadData().getHeart_rate();
   movement_type=pojo.getEvent().getPayloadData().getMovement_type();
   epilettiform_pattern=pojo.getEvent().getPayloadData().isEpilettiform_pattern();
   perc_sat_hemoglobin=pojo.getEvent().getPayloadData().getPerc_sat_hemoglobin();
   hypertensive_patient=pojo.getEvent().getPayloadData().isHypertensive_patient();
   hypotensive_patient=pojo.getEvent().getPayloadData().isHypotensive_patient();
}

   private boolean epileptic_subject;
   private boolean recent_surgery;
   private int systolic_pressure;
   private int diastolic_pressure;
   private int heart_rate;
   private int movement_type;
   private boolean epilettiform_pattern;
   private float perc_sat_hemoglobin;
   private boolean hypertensive_patient;
   private boolean hypotensive_patient;
   public boolean isEpilettiform_pattern() {
      return epilettiform_pattern;
   }
   public void setEpileptic_subject(boolean epileptic_subject) {
      this.epileptic_subject = epileptic_subject;
   }

   public void setRecent_surgery(boolean recent_surgery) {
      this.recent_surgery = recent_surgery;
   }

   public boolean isRecent_surgery() {
      return recent_surgery;
   }

   public boolean isEpileptic_subject() {
      return epileptic_subject;
   }

   public boolean isHypertensive_patient() {
      return hypertensive_patient;
   }

   public boolean isHypotensive_patient() {
      return hypotensive_patient;
   }

   public float getPerc_sat_hemoglobin() {
      return perc_sat_hemoglobin;
   }

   public int getDiastolic_pressure() {
      return diastolic_pressure;
   }

   public int getHeart_rate() {
      return heart_rate;
   }
```

```
public int getMovement_type() {
   return movement_type;
}

public int getSystolic_pressure() {
   return systolic_pressure;
}

public void setDiastolic_pressure(int diastolic_pressure) {
   this.diastolic_pressure = diastolic_pressure;
}

public void setSystolic_pressure(int systolic_pressure) {
   this.systolic_pressure = systolic_pressure;
}

public void setEpilettiform_pattern(boolean epilettiform_pattern) {
   this.epilettiform_pattern = epilettiform_pattern;
}

public void setHeart_rate(int heart_rate) {
   this.heart_rate = heart_rate;
}

public void setHypertensive_patient(boolean hypertensive_patient) {
   this.hypertensive_patient = hypertensive_patient;
}

public void setHypotensive_patient(boolean hypotensive_patient) {
   this.hypotensive_patient = hypotensive_patient;
}

public void setMovement_type(int movement_type) {
   this.movement_type = movement_type;
}

public void setPerc_sat_hemoglobin(float perc_sat_hemoglobin) {
   this.perc_sat_hemoglobin = perc_sat_hemoglobin;
}

public void setMeta_id_utente(String meta_id_utente) {
   this.meta_id_utente = meta_id_utente;
}

public void setMeta_id_doctor(String meta_id_doctor) {
   this.meta_id_doctor = meta_id_doctor;
}

public void setMeta_timestamp(String meta_timestamp) {
   this.meta_timestamp = meta_timestamp;
}

public void setMeta_routingKey(String meta_routingKey) {
   this.meta_routingKey = meta_routingKey;
}

public void setMeta_alertContact(String meta_alertContact) {
   this.meta_alertContact = meta_alertContact;
}
```

```
    public String getMeta_timestamp() {
        return meta_timestamp;
    }

    public String getMeta_routingKey() {
        return meta_routingKey;
    }

    public String getMeta_id_utente() {
        return meta_id_utente;
    }

    public String getMeta_id_doctor() {
        return meta_id_doctor;
    }

    public String getMeta_alertContact() {
        return meta_alertContact;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
```

H. app.js (TLS implementation)

```
. . .

app.use(function(req,res,next) {
  if (!/https/.test(req.protocol)){
     res.redirect("https://" + req.headers.host + req.url);
  } else {
     return next();
  }
});
app.get('*',function (req, res) {
    res.redirect('https://127.0.0.1' + req.url);
});


//redirect http
http.Server(function (req, res) {
    res.writeHead(301, { "Location": "https://" + req.headers['host'].replace(port,https_port) + req.url });
    console.log("http request, will go to >> ");
    console.log("https://" + req.headers['host'].replace(port,https_port) + req.url );
    res.end();
}).listen(port);

//https
https.Server(options, app).listen(https_port, function () {
 console.log('Express app listening on http port ' + port+ '\n' +"Express app listening on Https on port",https_port);

. . .
```