REST4CEP: RESTful APIs for Complex Event Processing

Ángel Gamaza[1], Guadalupe Ortiz[2], Juan Boubeta-Puig[2], Alfonso Garcia-de-Prado[2]

[2]UCASE Software Engineering Group, [1,2]University of Cádiz, Avda. de la Universidad de Cádiz 10, 11519 Puerto Real, Cádiz, Spain

Abstract. *Complex Event Processing (CEP) is a powerful technology thoroughly used in cutting-edge software architectures to support decision-making in multiple domains. Currently, developing such CEP-enhanced software architectures is not an easy task and there are no general purpose Application Programming Interfaces (APIs) which support programming and software development for CEP-based systems. This paper provides two RESTful APIs which support the management, storage and maintenance of events and patterns of interest both in design time and at runtime. This way, we simplify and speed up the flexible development of any CEP-based software architecture.*

Keywords: *RESTful API; Complex event processing; Event-Driven Service-Oriented Architecture; Decision-making*

1 Introduction

Complex Event Processing (CEP) is a growing and expanding technology which has been, and is currently being used, to improve business decision-making [1]. CEP engines usually require the definition and deployment of the types of events and event patterns to be detected in advance, making it difficult to track the availability of such event types and patterns in the engine. Besides, CEP engines do not usually provide mechanisms to facilitate the monitoring of detected event and complex event instances at runtime.

In the recent past, we have developed several CEP-based software architectures [2–4]; this led us to the realization that we are missing an Application Programming Interface (API) that allows the creation, storage, deployment and query of event types and patterns in the engine, as well as the storage, deployment and monitoring of runtime instances detected in the engine. Thus, the main aim of REST4CEP is to bridge such a gap through the provision of RESTful [5] APIs that allow CEP-based software management and monitoring in a generic way, so that it can be reused in different systems and application domains when the software development of CEP-based applications is required.

2 Problems and Background

Despite all the advantages provided by Event-Driven Service-Oriented Architectures (ED-SOA or SOA 2.0) [6], this type of architecture might not be ideal to analyze and correlate large amounts of data in terms of events. To meet this requirement, the integration of CEP [7] becomes necessary: a technology that allows the capturing, analyzing and correlating of considerable amounts of heterogeneous data with the aim of detecting relevant situations in a particular domain and therefore improving decision-making in such domains [8]. In CEP-based software, event patterns specify the conditions to be met in order to detect the situations of interest; the latter are called complex events. Such simple events, complex events and patterns are managed by a CEP engine, a software which can analyze stream data in real time. Definitely, CEP has gained a significant role in SOA 2.0., facilitating the implementation of the business logic based on the occurrence of events and on the matching of complex event patterns [7].

At present, the problem is that there is no middleware support when developing CEP-based software architectures and applications; in particular, there are no APIs available to facilitate the management, storage and maintenance of events and patterns of interest both in design time and at runtime. It is, therefore, necessary to develop an ad-hoc software artifact for each case study or application domain. There are some libraries and APIs which aim to make such a task easier; as an example, we can name the one from FlinkCEP [9] or FIWARE's CEP Open Restful API [10]. Both are engine-specific and therefore can only interact with software developed for their CEP engines (FlinkCEP and Perseo, respectively). Besides, in the case of FlinkCEP there is no RESTful interface but a Java library is provided and therefore is highly coupled with the system's implementation. In the case of FIWARE, the deprecated API version was very limited and the current one seems to remain under development [11].

This is why we developed two APIs to meet such needs. The first API, from now on the *Design* RESTful API, supports the operations required at design time, which are particularly suited to the definition of the event types that the system is going to receive as well as the event patterns the system should detect. The second API, from now on the *Runtime* RESTful API, supports the operations which tackle the three types of situations detectable at runtime: the occurrence of an event of a previously defined event type, the matching of a previously defined pattern and the detection of an incorrect event type and pattern which cannot therefore be deployed.

3 Software Framework

In the following sub-sections, we first of all present REST4CEP[1] software architecture and, secondly, the software functionalities offered by the APIs.

3.1 Software Architecture

Fig. 1 presents REST4CEP software architecture. As we can see, the Design and Runtime APIs are decoupled and they interact with a SQL and a NoSQL database, respectively, which are set in the configuration module. Note that the APIs are set particularly for MySQL and Mongo databases; the configuration could be updated for other SQL alternatives; noSQL options would require further programming effort since we are currently using Spring Data Mongo in the API. Each API is structured into three layers: the Data layer is composed of the data models and the Data Access Object (DAO); the service layer incorporates the service business logic and the Data Transfer Object (DTO); finally, on top we have the REST Controller which maps the client requests into the service ones.

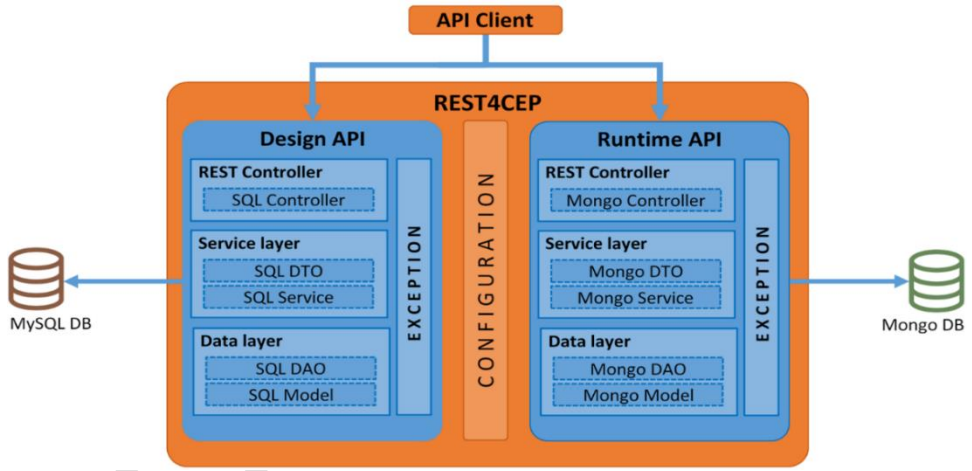

Fig. 1. REST4CEP Software Architecture

3.2 Software Functionalities

REST4CEP APIs permit the following functionalities. For event types and event pattern design, the Design RESTful API interacts with a SQL database, offering the following operations:

- Event types: *get all event types*, *insert event type, get an event type by id, get an event type by name, update an event type, delete an event type, set an event type as ready to deploy, set an event type as unready to deploy, set an event type as deployed, set an event type as undeployed, get all event types ready to deploy, get all event types unready to deploy, get all deployed event types, get all undeployed event types*.
- Event patterns: *get all event patterns, insert event pattern, get an event pattern by id, get an event pattern by name, update an event pattern, delete an event pattern, set a pattern as ready to deploy, set a pattern as unready to deploy, set a pattern as deployed, set a pattern as undeployed, get all event patterns ready to deploy, get all event patterns unready to deploy, get all deployed event patterns, get all undeployed event patterns, link an event type with a pattern , unlink an event type with a pattern*.

[1] REST4CEP is currently deployed and available only for reviewers at https://angelgamaza.es:6443/rest4cep/; username design and runtime (depending on the API) and password REST4CEP.2020$. Please note the server might be busy occasionally. When testing it, note that the deployed API has been moved to HTTPS protocol.

For event instances, complex event instances and incorrect event type and pattern definitions —only detectable at runtime—, the Runtime RESTful API interacts with a noSQL database, offering the following operations:

- Event instances: *get all events, insert new event, get an event by id, get the last five events, delete an event by id, delete all events*.
- Complex event instances: *get all detected complex events, insert a complex event, get a complex event by id, get the last five complex events, delete a complex event by id, delete all detected complex events*.
- Incorrect event type instances: *get all incorrect event types, insert an incorrect event type, get incorrect event type by id, get last five incorrect event types, delete an incorrect event type by id, delete all incorrect event types*.
- Incorrect event pattern instances: *get all incorrect event patterns, insert an incorrect event pattern, get incorrect event pattern by id, get last five incorrect event patterns, delete an incorrect event pattern by id, delete all incorrect event patterns*.

4 Implementation

REST4CEP has been implemented using the Java programming language, based on the Spring framework [12] to help abstraction and using Java Persistence API (JPA) [13] for operations with databases.

The project is divided into two major packages —*java* and *resources*—. The *java* package is used to include all the Java code related to the RESTful APIs. The resources package is used to add resources to the project, such as text files, property files, etc. The *java* package with all the business logic is divided into three packages itself:

- config: inside this package we find all Java files related to project configurations, such as database access sockets, beans for dependency injection and other parameterization aspects of the used libraries.
- design: within this package we find all Java files that deal with the exchange of information with the Design RESTful API for event type and pattern definition.
- runtime: inside this package we find all Java files related to the exchange of information with the Runtime RESTful API of runtime simple event, complex event, incorrect event type and incorrect pattern instances.

Each of these packages is also composed of several packages:

- model: composed of those Java files that represent the entities in the database and that relate such Java objects to the database entities. The entities of the database are converted into objects from these models to be dealt with in Java by performing object-relational mapping.
- dao: this package consists of those Java files, called *repositories,* that allow the CRUD (Create, Read, Update and Delete) operations to be performed on the services that use them. They also permit the performance of new functions using JPA in order to abstract from the SQL language.
- service: this Java package contains all files that implement the service logic, performing all operations internally and mapping the objects of type Model to DTO (and vice versa), which is the final format that the user receives from the API.
- exception: this Java package includes all those files with custom-defined exceptions.
- dto: within this code package, we define the conversion of the database models into the data to be delivered in response to the API users, so as to preserve the integrity of the system data.
- controller: this package contains all the code related to the mapping of RESTful methods to Java ones to permit direct interaction with the user. These classes must not contain application logic, but only call to the corresponding services, which return the desired information to them.

5 Illustrative Example

We implemented a SOA 2.0 to detect complex events in the domain of air quality monitoring. In particular, we linked the databases to an Esper CEP engine [14] and a Mule Enterprise Service Bus (ESB) [15], as shown in Fig. 2.

More specifically, since air quality has become an issue of great interest to governments and citizens nowadays, we implemented the architecture in the scope of an information and alert service about air quality levels dangerous to our health. The high concentrations of various pollutants present in the air we breathe can be very harmful to anyone, and in particular to various at-risk groups , so it is of great interest to have applications that

allow us to easily check air quality status as well as to promptly receive alerts whenever levels become harmful to our health [4]. For that purpose, we used the air quality data provided by the local Andalusian government.

In order to provide such services, a software infrastructure is necessary; in the past we proposed an architecture to facilitate such an infrastructure and the named services [2]. The said architecture is based on CEP and it was during its implementation that became aware of the need for a generic multi-domain RESTful API to facilitate the implementation of this type of architectures and services. Thanks to the novel REST4CEP proposed and implemented in this paper, (1) the creation, storage, deployment and query of air quality event types and patterns according to pollutant levels to be detected (for instance, for different countries' standard levels), as well as (2) the storage, deployment and monitoring of event and pattern runtime occurrences for different location air quality stations, were properly facilitated. Please note that the developer still has to implement the software architecture which will have the business logic of the application with the embedded CEP engine. In our case, such a business logic was implemented by embedding Esper CEP engine in Mule ESB (see Figure 2). The ESB will be of the following: (i) retrieving the event types and patterns to be deployed in the CEP engine, (ii) deploying them, updating their status in the API database, (iii) submitting any existing deployment errors to the runtime API database, (iv) receiving simple events from the data sources, (v) submitting such simple events both to the CEP engine and to the runtime database, (vi) detecting complex events, (vii) submitting the latter to the runtime database, and (ix) acting in accordance with detected complex events. Besides, we defined a web-based management and monitoring console which also interacted with the REST4CEP APIs, which is out of the scope of this paper.

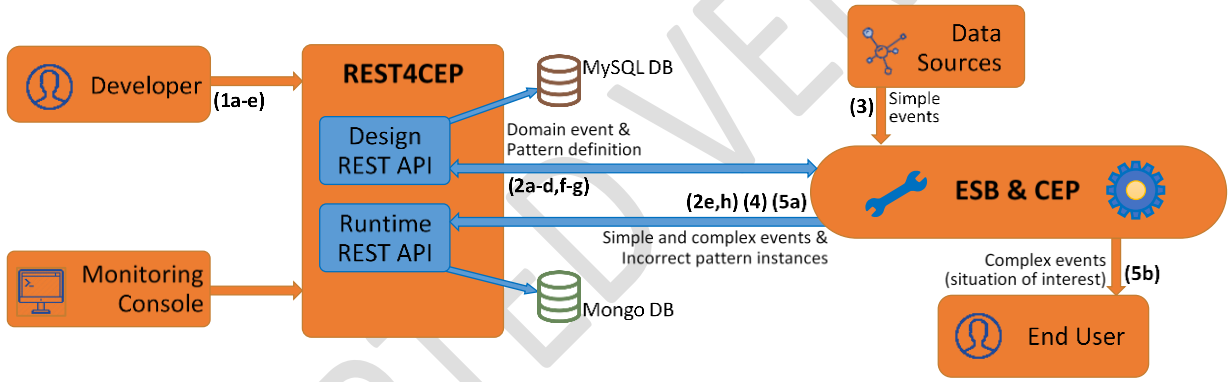

Fig. 2. SOA 2.0 Architecture using REST4CEP

Therefore, for the architecture implementation, assuming that the business logic in the ESB has been implemented, the steps to be followed are explained below and represented in the sequence diagram in Figure 3 (we omitted response messages for clarity purposes). Note that the sequence diagram represents the interaction between the different components and participants, but not the internal operations in the ESB. We also included the numbering of such messages in Figure 2.

(1) First of all, the developer will (1a) insert the event types and (1b) the patterns to be detected in the design database, (1c) link the event types to the corresponding patterns, (1d) set the event types ready for deployment and (1e) set the patterns ready for deployment, as explained in the following paragraphs.
   a. Insert the event types to be detected in the SQL database through the invocation of the Design RESTful API (operation *POST /design/event_type*). By default, event types will be set as *unreadyToDeploy*, so they are not expected to be deployed in the CEP engine yet.
   b. Insert the patterns to be detected in the SQL database through the use of the Design RESTful API (operation *POST /design/event_pattern*). By default, event patterns are set as *unreadyToDeploy*, therefore they are not expected to be deployed in the CEP engine yet.
   c. Then, event types will be linked with the corresponding event patterns through the invocation of the Design RESTful API (operation *POST /design/event_pattern/link*). This is required because we need to ensure that an event type, which is referred in a pattern, is known by the CEP engine in advance; otherwise pattern deployment may fail. The business logic should take that into account

when deploying event types and patterns; if necessary, when we obtain the information of an event pattern, we can see the linked event types.

d. Now, we have to set the event types that we expect to be deployed as ready for deployment by invoking the Design RESTful API (operation *PUT /design/event_type/ready/{id}*). The ESB logic should be programmed to poll the event types to be deployed from the database as explained in step (2a).
e. Finally, we have to set the patterns we expect to be deployed as ready for deployment by invoking the Design RESTful API (operation *PUT /design/event_pattern/ready/{id]*). The ESB logic should also be programmed to poll the patterns to be deployed from the database as explained in step (2b).

(2) Then, the defined code in the ESB should read and deploy (2a) the event types and (2b) patterns ready to be deployed. If deployment was successful, (2c) the ESB will set the successfully deployed event types as deployed; otherwise, if deployment failed, (2d) the ESB will set the event type back as unready to deploy and (2e) save the event type and error in the runtime database (2e). The same would happen with the patterns: if they were successfully deployed, they would be set as deployed by the ESB, but if deployment failed, (2f) the ESB would set the pattern back as unready to deploy and (2g) save the pattern and error in the runtime database, as explained in the following paragraphs:

a. To deploy the ready to deploy event types we have to obtain them by invoking the Design RESTful API (operation *GET /design/event_type/readytodeploy*); then, the business logic should be programmed in accordance with the deployment in the particular CEP engine being used in the implementation.
b. To deploy the ready to deploy patterns we have to obtain them by invoking the Design RESTful API (operation G*ET /design/event_pattern/readytodeploy*); then, the business logic should be programmed in accordance with the deployment in the particular CEP engine being used in the implementation.
c. If the event types are successfully deployed, they have to be set as deployed in the database by invoking the Design RESTful API (operation *PUT /design/event_type/deploy/{id}*).
d. Those event types which were not successfully deployed have to be set as unready to deploy by invoking the Design RESTful API (operation *PUT /design/event_type/unready/{id}*).
e. Also, if the event type deployment failed, the event type and error need to be stored in the runtime database for inspection by the developer. For that purpose, we invoke the Runtime RESTful API (operation *POST /runtime/incorrect_event_type*).
f. If the patterns are successfully deployed, they have to be set as deployed in the database by invoking the Design RESTful API (operation *PUT /design/event_pattern/deploy/{id}*).
g. Those patterns which were not successfully deployed have to be set as unready to deploy by invoking the Design RESTful API (operation *PUT /design/event_pattern/unready/{id}*).
h. Also, if pattern deployment failed, both pattern and error need to be stored in the runtime database for inspection by the developer. For that purpose, we invoke the Runtime RESTful API (operation *POST /runtime/incorrect_event_pattern*).

(3) The ESB business logic is programmed to receive data from the data sources (simple events).

(4) Whenever we receive an event, if we want to save it for further examination, we need to store it in the runtime database through the invocation of the Runtime RESTful API (operation *POST /runtime/event*).

(5) Whenever we detect a complex event, we have to (5a) store it and (5b) notify interested users.

a. To store the complex events detected in the runtime database, for further examination, we have to invoke the Runtime RESTful API (operation *POST /runtime/complex_events_detected*).
b. For user notification about situations of interest, we have to implement the required business logic in the ESB. Even though, for simplicity's sake, we have represented the same user to store information in the database and receive notification of situations of interest in Figure 3, they need not be the same.

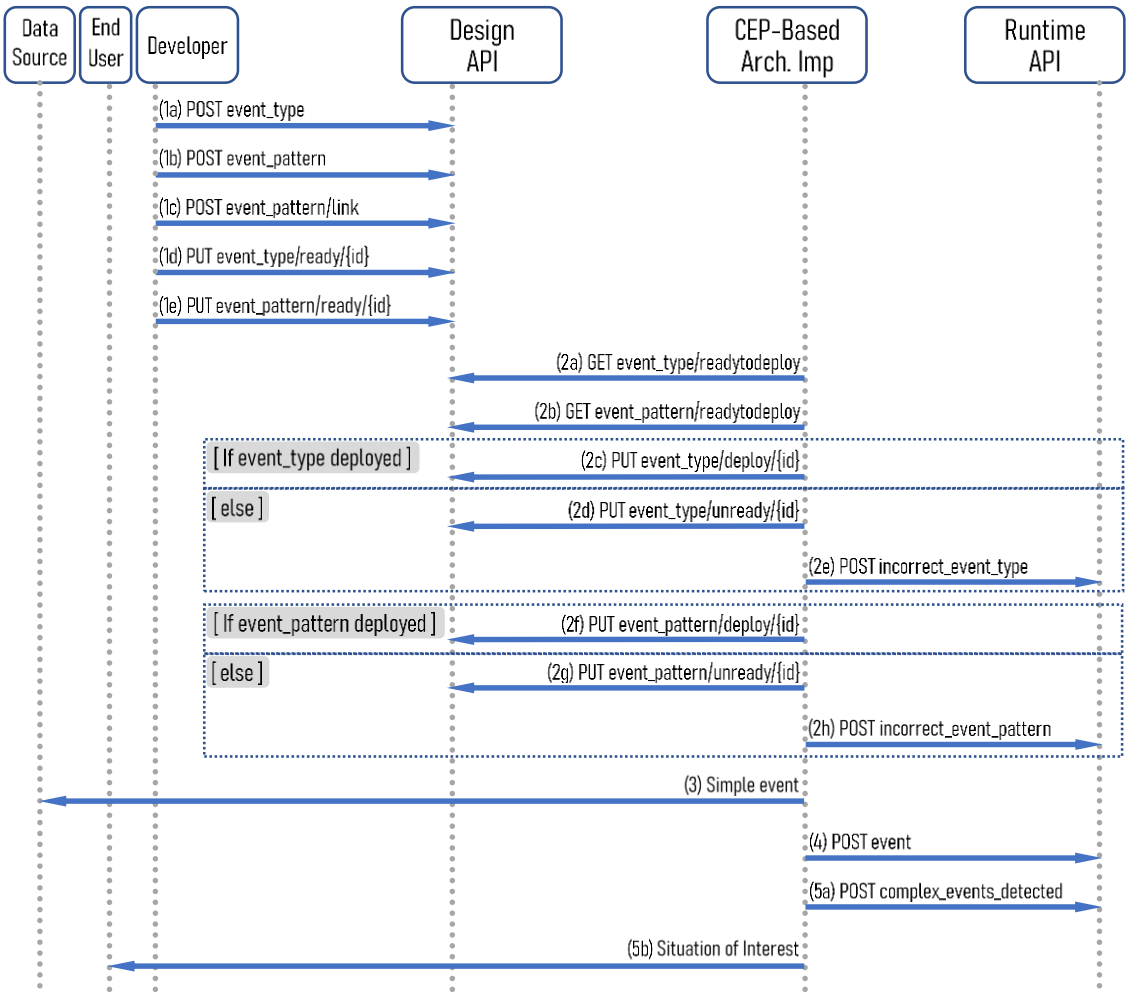

Fig. 3. Sequence diagram representing the messages between the different components and participants from design to runtime in a CEP-based architecture, using REST4CEP.

6 Conclusions

REST4CEP RESTful APIs are fully reusable software artifacts which support programming and software development for CEP-based systems in general and for SOA 2.0 architectures in particular. More specifically, the developer will benefit from the Design RESTful API which facilitates all the management of the event types and patterns in design time, as well as from the Runtime RESTful API, which can manage all event instances, complex events (patterns) and incorrect event types and patterns detected in the domain in question at runtime, through their storage, and update on SQL and NoSQL databases.

**Acknowledgements*

This work was supported by the Spanish Ministry of Science and Innovation and the European Union FEDER Funds [grant number RTI2018-093608-B-C33].

References

[1] J. Boubeta-Puig, G. Ortiz, I. Medina-Bulo, MEdit4CEP: A model-driven solution for real-time decision making in SOA 2.0, Knowl.-Based Syst. 89 (2015) 97–112. https://doi.org/10.1016/j.knosys.2015.06.021.

[2] A. Garcia-de-Prado, G. Ortiz, J. Boubeta-Puig, CARED-SOA: A Context-Aware Event-Driven Service-Oriented Architecture, IEEE Access. 5 (2017) 4646–4663. https://doi.org/10.1109/ACCESS.2017.2679338.
[3] A. Garcia-de-Prado, G. Ortiz, J. Boubeta-Puig, COLLECT: COLLaborativE ConText-aware service oriented architecture for intelligent decision-making in the Internet of Things, Expert Syst. Appl. 85 (2017) 231–248. https://doi.org/10.1016/j.eswa.2017.05.034.
[4] A. Garcia-de-Prado, G.O. Ortiz, J. Boubeta-Puig, D. Corral-Plaza, Air4People: a Smart Air Quality Monitoring and Context-Aware Notification System, J. Univers. Comput. Sci. 24 (2018) 846–863.
[5] F. Roy Thomas, Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, 2000. http://dl.acm.org/citation.cfm?id=932295.
[6] M. Papazoglou, Service-oriented computing: concepts, characteristics and directions, in: Web Inf. Syst. Eng., IEEE Comput. Soc, 2012: pp. 3–12. https://doi.org/10.1109/WISE.2003.1254461.
[7] D.C. Luckham, Event processing for business: organizing the real-time enterprise, John Wiley & Sons, Hoboken, N.J, USA, 2012. https://www.safaribooksonline.com/library/view/event-processing-for/9781118171851/.
[8] C. Inzinger, W. Hummer, B. Satzger, P. Leitner, S. Dustdar, Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems: event-based monitoring and adaptation for distributed systems, Softw. Pract. Exp. 44 (2014) 805–822. https://doi.org/10.1002/spe.2254.
[9] Apache Flink 1.9 Documentation: FlinkCEP - Complex event processing for Flink, (2020). https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html (accessed March 31, 2020).
[10] Complex Event Processing Open RESTful API Specification - FIWARE Forge Wiki, (2014). http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Complex_Event_Processing_Open_RESTful_API_Specification (accessed March 31, 2020).
[11] API - Perseo Context-aware Complex Event Processing, (2019). https://perseo.readthedocs.io/en/latest/API/api/ (accessed March 31, 2020).
[12] spring.io, (2020). https://spring.io/ (accessed March 31, 2020).
[13] Accessing Data with JPA, (2020). https://spring.io/guides/gs/accessing-data-jpa/ (accessed March 31, 2020).
[14] EsperTech, Esper, (2020). http://www.espertech.com/esper/ (accessed March 31, 2020).
[15] MuleSoft, Mule ESB, (2020). http://www.mulesoft.org/ (accessed March 31, 2020).

Appendix A- Required Metadata

A1 Current executable software version

Table A.1 Software metadata

Nr	(executable) Software metadata description	*Please fill in this column*
S1	Current software version	*2.0.0*
S2	Permanent link to executables of this version	*https://gitlab.com/ucase/public/rest4cep*
S3	Legal Software License	*GNU GPL*
S4	Computing platform / Operating System	*Java-compatible platforms*
S5	Installation requirements & dependencies	*JDK 1.11, SQL Database, Mongo DB, Maven*
S6	Link to user manual	*https://gitlab.com/ucase/public/rest4cep/-/wikis/User-Manual*
S6	Support email for questions	*guadalupe.ortiz@uca.es, angel.gamaza@gmail.com*

A2 Current code version

Table A.2 Code metadata

Nr	Code metadata description	*Please fill in this column*
C1	Current Code version	*2.0.0*

C2	Permanent link to code/repository used of this code version	*https://gitlab.com/ucase/public/rest4cep*
C3	Legal Code License	*Apache License 2.0*
C4	Code Versioning system used	*Git*
C5	Software Code Language used	*Java, Spring Framework*
C6	Compilation requirements, Operating environments & dependencies	*Java 1.11 or OpenJDK 1.11, Maven*
C7	Developer documentation	*https://gitlab.com/ucase/public/rest4cep/-/wikis/Developer-Manual*
C8	Support email for questions	*guadalupe.ortiz@uca.es, angel.gamaza@gmail.com*