Innovation action

D7.3 – Dynamic data integration, storage and access

<table>
<thead>
<tr>
<th>Author/s</th>
<th>Fausto Giunchiglia, Ronald Chenu Abente, Enrico Bignotti, Alessio Zamboni, Gabor Bella, Mattia Zeni, Andrea Passerini (UniTN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer</td>
<td>Eddy Maddalena (Soton)</td>
</tr>
<tr>
<td>Due date</td>
<td>30.11.2018</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Dissemination level</td>
<td>PU</td>
</tr>
<tr>
<td>Status</td>
<td>Final</td>
</tr>
</tbody>
</table>
Table of contents

Abstract 5

Executive Summary 6

1. Introduction 7

2. Qrowd-DB overview 7

3. The Linked Data Storage 8
   3.1 The Entity Base 9
   3.2 The QROWD Data Model 10
      3.2.1 Entity requirements 10
      3.2.2 The Core Ontology 11
      3.2.3 Citizen 12
      3.2.4 Trip 14
      3.2.5 Modal Split 15
      3.2.6 Parking Infrastructure 17
      3.2.7 Onstreet Parking 17
      3.2.8 Offstreet Parking 18
      3.2.9 Bike Hire Docking Station 19
      3.2.10 Parking Group 20

4. Big Data Storage 20
   4.1 Apache Cassandra: distributed, scalable database system 21
   4.2 Data schema to enable anonymization 21
   4.3 Database physical infrastructure 22
      4.4.1 Data center 22
      4.4.2 Network Configuration 23
      4.4.3 Dedicated Hardware 24

5. The Business Logic 24
   5.1 Linked Data Storage Business Logic 24
      5.1.1 Main call groups 24
      5.1.2. Modal Split 25
   5.2 Big data Storage Business Logic 26
      5.2.1 Architecture 26
      5.2.2 Smartphone-to-server module 27
      5.2.3 Database insert module 27
      5.2.4 Server-to-smartphone module 28

6. Conclusions 30
### Dynamic data integration, storage and access

#### References

<table>
<thead>
<tr>
<th>Annex 1 - Linked data storage API reference</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET Citizen</td>
<td>32</td>
</tr>
<tr>
<td>POST Citizen</td>
<td>33</td>
</tr>
<tr>
<td>GET Trip</td>
<td>34</td>
</tr>
<tr>
<td>POST Trip</td>
<td>36</td>
</tr>
<tr>
<td>GET Modal Split</td>
<td>38</td>
</tr>
<tr>
<td>GET Parking</td>
<td>39</td>
</tr>
<tr>
<td>Emulated API Calls</td>
<td>41</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Annex 2 - Big data storage API reference</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smartphone-to-server module</td>
<td>42</td>
</tr>
<tr>
<td>GET Online</td>
<td>42</td>
</tr>
<tr>
<td>GET Projects</td>
<td>42</td>
</tr>
<tr>
<td>GET Login</td>
<td>43</td>
</tr>
<tr>
<td>GET Refresh Google Cloud Messaging token</td>
<td>43</td>
</tr>
<tr>
<td>GET Check user data</td>
<td>44</td>
</tr>
<tr>
<td>GET Signup</td>
<td>45</td>
</tr>
<tr>
<td>POST Upload</td>
<td>45</td>
</tr>
<tr>
<td>Database insert module</td>
<td>46</td>
</tr>
<tr>
<td>GET Online</td>
<td>46</td>
</tr>
<tr>
<td>GET Process files with errors</td>
<td>46</td>
</tr>
<tr>
<td>Server-to-smartphone module</td>
<td>47</td>
</tr>
<tr>
<td>GET Online</td>
<td>47</td>
</tr>
<tr>
<td>POST Challenge definition</td>
<td>48</td>
</tr>
<tr>
<td>POST Challenge instance</td>
<td>49</td>
</tr>
<tr>
<td>GET Challenge instances</td>
<td>50</td>
</tr>
<tr>
<td>GET Challenge result</td>
<td>50</td>
</tr>
<tr>
<td>GET Publish challenge result</td>
<td>51</td>
</tr>
<tr>
<td>POST Challenge synchronization</td>
<td>52</td>
</tr>
<tr>
<td>POST Reception confirmation</td>
<td>52</td>
</tr>
<tr>
<td>POST Contribution answer</td>
<td>54</td>
</tr>
<tr>
<td>GET Send task</td>
<td>55</td>
</tr>
<tr>
<td>GET Send message</td>
<td>55</td>
</tr>
<tr>
<td>GET Send question</td>
<td>56</td>
</tr>
<tr>
<td>GET Download map</td>
<td>57</td>
</tr>
<tr>
<td>GET Update map</td>
<td>58</td>
</tr>
<tr>
<td>GET Tasks by user</td>
<td>58</td>
</tr>
<tr>
<td>GET Task by id for user</td>
<td>59</td>
</tr>
<tr>
<td>GET Task answer</td>
<td>60</td>
</tr>
<tr>
<td>GET Challenges answers</td>
<td>61</td>
</tr>
</tbody>
</table>
Dynamic data integration, storage and access

Annex 3 - Big data storage technology stack

- Apache Spark
- Container
- Docker
- Kubernetes

Annex 4 - Top level eTypes
ABSTRACT
To succeed in its action, QROWD heavily relies on data of different nature. Generally speaking, there are two types of data: the first one is Big Data, which are too large or complex for traditional data-processing application, and Linked Data, which are data structured in such a way to exploit their interlinking of information. This deliverable introduces the Qrowd-DB, which is a fundamental component of the general QROWD architecture and whose main purpose is to store and manage all the QROWD data, which also includes the general task of data integration. There are two main elements within the Qrowd-DB: the first is the Big Data Storage, which is dedicated to managing the big data from the business cases in Trento, while the second is the Linked Data Storage which manages all the knowledge about the QROWD project within the general domain of mobility. Both the components have dedicated APIs for interacting with them.
EXECUTIVE SUMMARY

The deliverable D7.3 reports on the Qrowd-DB database, and the tasks carried out to enable data access by the third-party applications, in addition to the related data integration tasks. There are two main components in the Qrowd-DB: the first one is the Stream Base, which stores the stream of data collected via iLog, while the second one is the Entity Base, which stores all the knowledge about the Trento business cases.

The audience for this Deliverable are the QROWD team members. D7.3 contains a detailed description of the Qrowd-DB database, focusing on its dedicated components and the APIs for accessing it, providing the QROWD member with all the technical information required.

D7.3 is related to the deliverables of WP8, specifically to D8.1 and D8.3, in terms of architecture, to the business case WPs (WP1, WP2, and WP3) especially with respect to D2.4, and WP4 in terms of the general tasks of data integration in D4.3.

The main outputs of this deliverable is an exhaustive overview of the Qrowd-DB that allows the reader to understand the full process of storing and managing the data collected to make them available within the QROWD project for carrying out the action.
1. INTRODUCTION

The mission of QROWN is to solve mobility issues with data-driven approaches, and as such the QROWN architecture must be able store, manage and exploit a wide variety of data, ranging from mobility to meteorological, and news data. Generally speaking, the data can be divided into two main categories: Big Data and Linked Data. The former are massive amount of data that are heterogeneous in nature, while the latter represent more structured data that are linked to generate knowledge. While being very different, both these data require to have some level of integration as such to make use of them.

This deliverable introduces the Qrowd-DB, a vital element of the overall QROWN architecture which essentially acts as the database for all the QROWN data. There are two main components in the Qrowd-DB: the first one is the Big Data Storage, which is dedicated to store big data and to exploits state-of-the-art technologies to manage the varied nature of such data, while the second one is the Linked Data Storage, which structures the data with a specific ad-hoc model to perform data integration of the various information collected within the project. In addition, the Qrowd-DB comes with a dedicated set of APIs to interact with the two storages.

The remainder of the document is structured as follows. Section 2. QROWN-DB Overview provides an overview of the Qrowd-DB and of its main elements, namely the Linked Data Storage, the Big Data Storage and the APIs. For each element, we dedicate a section, where Section 3. The Linked Data Storage presents the Linked Data Storage, Section 4. Big Data Storage presents the Big Data Storage, and Section 5. The Business Logic presents the APIs for accessing the Qrowd-DB. Finally, Section 6. Conclusions concludes the deliverable.

2. QROWN-DB OVERVIEW

Based on the overall system architecture defined in D8.1 and shown in Figure 1. The QROWN Architecture, Qrowd-DB links with multiple components and Work Packages that deal with heterogeneous types of data. For this reason, it has to include multiple components that are able to collect, store and manage different types of data, namely, linked data but also big unstructured data. The goal of Qrowd-DB is to profile and model datasets and streams, accounting for their features in terms of dynamics, currentness, coverage, and privacy. This will facilitate the discovery and exploitation of data in specific processes and services.

We can define Qrowd-DB as composed by three main elements:

1) **The Linked Data Storage**: its main goal is to store and manage linked data, entities;
2) **The Big Data Storage**: dedicated to the storage of the personal big data generated by mobile phone of citizens;
3) **The Business Logic**: a set of software components previously developed by the University of Trento that allows the partners to access the underlying two systems and databases
3. The Linked Data Storage

Within this deliverable, we refer to Linked Data as data that is interlinked and semantically structured, i.e., that can be retrieved through semantic queries. In general, the data obtained from datasets as detailed in D2.2 and D4.1 can become linked data by interlinking and structuring them. Within the Qrowd-DB, such data are stored in the Linked data storage, which makes them useful by giving them a structure that allows to exploit their semantics, e.g., understanding that a trip ends in a certain location which is not just some coordinated but actually a parking. Section 3.1 The Entity Base presents the Linked Data Storage and the way knowledge is stored within via an entity-centric approach, while Section 3.2 The QROWD Data Model details the QROWD data model, providing its requirements and modelling choices.
3.1 The Entity Base

The Linked Data storage stores concrete information about abstract and physical entities that exist in the real world. Since the information is structured in an entity-centric approach, the Linked Data storage is an Entity Base (EB) represented by the following elements:

- An entity (E) is defined as an abstract or physical object, can be of different types (e.g., person, location, event, etc.) and is described by attributes (e.g., name, birth date, latitude-longitude, size, duration, etc.), which can be different for different types of entities. It is defined by mean of an entity type (ET) and a non-empty set of attributes describing the characteristics of the entity (\{A\}).
- An attribute (A) instantiates an attribute definition AD to represent a particular characteristic of the entity. Some attributes may have multiple values, that can be mapped to a meaning in some knowledge base (i.e., semantic values) or can represent a relation to another entity when the value is a reference to another E (i.e., relational attribute). It is formally defined by mean of an attribute definition (AD) and a set \{V\} of attribute values of the type of the corresponding AD.
- An eType (ET) allows establishing restrictions on the set of attributes \{A\} used to describe a given type of entity, where the meaning is further specified by mapping single elements (i.e., types of entities, the names of attributes and their values) to concepts from the underlying ontology that is also part of the same EB.

Gruber [2] defines ontology as a formal specification of a shared conceptualization. The notion of conceptualization refers to an abstract model of how people theorize (the relevant part of) the world in terms of basic cognitive units called concepts. Explicit specification means that the abstract model is made explicit, for instance by providing terms and definitions for the concepts. In other words the terms and the definition of the concept provide a specification of its meaning in relation with other concepts. The specification is said to be formal when it is written in a language with formal syntax and formal semantics, i.e. in a logic-based language. The conceptualization is shared in the sense that it captures knowledge, which is common to a community of people and therefore represents concretely the level of agreement reached in that community. Accordingly, we attach to each concept in the ontology, a set of terms and a definition.

For instance, the concept of “the date on which a person was born” can be associated in English with the terms “date of birth” and “birthday”. In developing the ontology we also comply with principles of ontological analysis [3] and apply the DERA methodology [1] developed in Trento. The latter draws a clear distinction between ontologies of entity classes, relations and attributes and gives precise principles, borrowed and adapted from the faceted approach [9], to be followed in order to develop robust, and easy to extend, ontologies. For instance, it is essential to make explicit the criteria followed to arrange concepts when appearing at same level of the ontology (siblings) and the semantic relations between them. We also frequently follow the guidelines provided by ISO for thesauri construction [5] that
Dynamic data integration, storage and access provide (among others) concrete suggestions about how to appropriately select terms to lexicalize the concepts.

### 3.2 The QROWD Data Model

The QROWD data model represents the main task of data integration and is a continuation of D4.1 and is related with D4.3. The data model is structured starting from a general analysis of the requirements, detailed in Section 3.2.1 Entity requirements, which leads to a dedicated Core Ontology of eTypes which was modelled starting from building a QROWD ontology to act as a conceptual building block for a set of eTypes, as described in 3.2.2 The Core Ontology.

#### 3.2.1 Entity requirements

The main things that we need to model in the Entity Base are:

1. **Citizens:** Citizens are the participants of the pilots and major stakeholder in improving the city transportation. Citizen provide information not only about their modal split but can also participate in challenges from WP2 to provide more information about transportation and mobility in general.

2. **Trips:** A trip is a movement from a stationary point A to a stationary point B. It is bounded either by the citizen reaching destination, or by changing their mode of transport. Notice that a trip may be composed by multiple sub-trips generated by different transportation modes.

3. **Parkings:** Given their relevance in WP1, parkings are also at the center of the challenges and play a vital role in understanding. Notice that parkings include not only the on-street and off-street parking, but also bike sharing docking station.

4. **Modal split:** Modal split is central in enabling data-driven policies for the Municipality of Trento.

One major requirement is that all the eTypes in the Entity Base have to be compliant with the FIWARE Data Model. FIWARE is a community that makes and shares open source technology for smart solutions to build an open sustainable ecosystem around public, royalty-free and implementation-driven software platform standards. Its final goal is to facilitate the development of new smart applications in multiple sectors including, but not limited, to smart cities. FIWARE produced several data models\(^1\), which are intended to model relevant, harmonized entities to enable data portability among different applications in smart city scenarios such as transportation and parking. More in detail, based on the business cases from WP2, the following entities were treated as a reference:

- **OffStreetParking\(^2\):** An off-street parking site with explicit entries and exits;
- **OnStreetParking\(^3\):** An on-street free entry or metered parking zone contains one or more adjacent parking spots;

---

\(^1\) [https://www.fiware.org/developers/data-models/](https://www.fiware.org/developers/data-models/)


Dynamic data integration, storage and access

**ParkingGroup**: A group of parking spots. The granularity level can vary. It can be a storey on a parking garage, a specific area belonging to a big parking lot or just a group of spots, differentiated for a specific purpose (usage, restrictions, etc.);

**BikeHireDockingStation**: It represents a bike hire docking station where subscribed users can hire and return a bike.

### 3.2.2 The Core Ontology

While FIWARE provided all necessary components for designing almost all transportation elements such as vehicle, parking lot, road, streetlight and even bike hire dock, it did not allow adding general classes in the model which were needed to account for the type of data in QROWD, e.g., citizens. To tackle this issue, we added more general class such as location, event, physical object, mental object etc. In this way, our model became more flexible and adaptive for handling the QROWD data.

![QROWD ontology in Protégé](image)

**Figure 2. QROWD ontology in Protégé**

*Figure 2. QROWD ontology in Protégé* shows the implemented model in Protégé, where the left panel shows the class hierarchy, and the right panel shows interconnection among the classes with various relations, using different colored arrows. Notice that the ontology provides not only the classes of entities and their hierarchy but also their possible attributes and relations (in accordance with the DEAR methodology), e.g., FIWARE-based attributes about parkings. There are four main classes: event, location, structure and person which are aligned to the DOLCE top-level ontology. DOLCE is a standard top level ontology which is at the core of our entity-centric approach⁶. There are two main advantage in relying on this ontology: *i*) it provides a rich vocabulary for describing different domains of knowledge, and *ii*) it

---

⁶ http://www.loa.istc.cnr.it/old/DOLCE.html
Dynamic data integration, storage and access facilitates data interoperability and reuse by providing an abstract schema that can be adapted based on the application requirements.

Below is the core ontology of QROWD, where the first level of the ontology is aligned with the DOLCE top-level, while the eTypes with an asterisks are those related to QROWD.

Entity

- [is-a] Person
  - [is-a] Citizen*
- [is-a] Location
- [is-a] Event
  - [is-a] Trip*
  - [is-a] Modal Split*
- [is-a] Structure
  - [is-a] Parking Infrastructure*
    - [is-a] On Street Parking*
    - [is-a] Off Street Parking*
  - [is-a] Bike Hire Docking Station*

In the next sections, we provide a description and motivation behind the modelling for the QROWD eTypes, while the top-level eTypes are presented in Annex 4 - Top level eTypes. Since eTypes inherit attributes from their parent, e.g., all children of Entity have names because Entity does, the inherited attributes are not visualized for the sake of clarity. The tables describe the name of the attribute, its description, and its datatype. For these last ones, square brackets are used for multivariate attributes, and angle brackets for relational ones.

3.2.3 Citizen

Unlike other entities we developed for QROWD, people are not represented in any FIWARE data model, save for the owner attribute of the Vehicle entity\(^7\), which redirects to schema.org Person type\(^8\). However, the Person eType was developed as preexisting work in compliance with DOLCE, and it acts as the parent of the Citizen eType, shown in Table 1. The Citizen eType.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>occupation</td>
<td>the principal activity in your life that you do to earn money; the concept restriction is {worker, student, other}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>drivingLicense</td>
<td>a license authorizing the bearer to drive a motor vehicle</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>


\(^8\) https://schema.org/Person
Dynamic data integration, storage and access

<table>
<thead>
<tr>
<th>numberCohabitants</th>
<th>number of people that live in the same house</th>
<th>INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointsAwarded</td>
<td>number of points currently awarded to citizen for the challenge</td>
<td>INTEGER</td>
</tr>
<tr>
<td>numberVehicles</td>
<td>total number of vehicles available to all cohabitants</td>
<td>INTEGER</td>
</tr>
<tr>
<td>preferredMode</td>
<td>preferred transportation mode; the concept restriction is {car, bus, train, motorcycle, cable car, bicycle, foot}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>Work sector</td>
<td>Sector of the work address of the citizen</td>
<td>STRING</td>
</tr>
<tr>
<td>Home sector</td>
<td>sector of the home address of the citizen</td>
<td>STRING</td>
</tr>
<tr>
<td>Availability of WiFi</td>
<td>Whether the citizen has a WiFi connection available</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>addressCountry</td>
<td>the country address</td>
<td>STRING</td>
</tr>
<tr>
<td>addressLocality</td>
<td>the locality address</td>
<td>STRING</td>
</tr>
<tr>
<td>citizenId</td>
<td>The id assigned to the citizen</td>
<td>STRING</td>
</tr>
<tr>
<td>age</td>
<td>a time in life (usually defined in years) at which some particular qualification or power arises</td>
<td>INTEGER</td>
</tr>
<tr>
<td>gender</td>
<td>the properties that distinguish organisms on the basis of their reproductive roles; the concept restriction is {male, female}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>Email</td>
<td>the electronic communication address of a person</td>
<td>STRING</td>
</tr>
<tr>
<td>streetAddress</td>
<td>The street address</td>
<td>STRING</td>
</tr>
<tr>
<td>citizenType</td>
<td>The type of citizen in QROWD; the concept restriction is {resident, commuter}</td>
<td>CONCEPT</td>
</tr>
</tbody>
</table>

With respect to other eTypes, citizen had additional requirements needed from the Municipality of Trento, especially in terms of demographics. Figure 3. Demographics from [8], taken from [8], shows the main demographics that are of interest to the Municipality of Trento. There are two types of data: family (in Italian, "livello familiare") and individual (in Italian, "singolo componente"). For the family, what is collected are the number of the following means of transportation owned: i) cars, ii) motorbikes, and iii) bikes; additionally, the distance between a family's home and the closest public transportation stop, e.g., bus stop, is also collected. For the individual, there are only three types of demographics: i) age, ii) gender, and iii) occupation. Similarly to [8], the same demographics are to be collected, although with some important differences. Firstly, the focus is much more on individuals rather than families, which means that there is no need for a dedicated eType. Secondly, additional data that are needed for the use case are as follows: i) domicile and residence, ii) number of people living in the same house, and iii) whether the citizen...
Dynamic data integration, storage and access has a driving license. Thirdly, to compute the official modal split for the whole city a sample size must be identified taking into account two parameters: i) whether the citizen is a resident or a commuter, since Trento is the capital of the Province and many people from neighboring cities commute daily, and ii) the type of occupation for each individual, i.e., worker, student, unemployed, retired, or stay-at-home.

Additional requirements were:

**Identification:** because of privacy reasons, we cannot store a citizen's name, which required us to create an anonymized id, but we can store the email.

**Challenges-specific attributes:** we modelled the points awarded to the citizen, especially since there is the possibility of citizens being recurringly involved in the pilots and challenges.

**Data collection:** by design, iLog uploads the data only over WiFi, which means that we need to understand whether the citizen had it available to devise different strategies for the data upload.

### 3.2.4 Trip

Similarly to Citizen, Trip is not present in FIWARE data models. Nonetheless, it is central in our model. It has been modelled in accordance also with standard modal split questionnaires, thus including i) information on the mode of transportation used, ii) the people involved, and iii) the purpose of the trip. A relevant dimension of trips is time. In fact, it is fundamental to understand i) which day of the week the trip took place, ii) whether it was a working day or not, and iii) the exact time when the trip occurred, e.g., peak hours. In addition, a trip may be constituted of subtrips, i.e., trips on different transportation modes to reach the final destination, each with its
Dynamic data integration, storage and access confidence value of detection. Table 2, The Trip eType shows the Trip eType.

Table 2. The Trip eType

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>means</td>
<td>list of means used to compute the modal split; the concept restriction is {car, bus, train, motorcycle, cable car, bicycle, foot}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>purpose</td>
<td>an anticipated outcome that is intended or that guides your planned actions; the concept restriction is {work, school, accompanying reasons errands, free time, trips made for work purposes during working hours, return trip}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>multirip</td>
<td>Whether the trip is composed of sub-trips</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>confidentValue</td>
<td>confidence value of the transportation mode classification</td>
<td>FLOAT</td>
</tr>
<tr>
<td>subtrips</td>
<td>The subtrips composing the trip</td>
<td>[] STRING</td>
</tr>
<tr>
<td>initDate</td>
<td>timestamp of the start of the trip</td>
<td>DATE</td>
</tr>
<tr>
<td>endDate</td>
<td>timestamp of the end of the trip</td>
<td>DATE</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>Any day of the week; the concept restriction is {monday, tuesday, wednesday, thursday, friday, saturday, sunday}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>partOfDay</td>
<td>The moment of the day; the concept restriction is {car, bus, train, motorcycle, cable car, bicycle, foot}</td>
<td>STRING</td>
</tr>
<tr>
<td>typicalDay</td>
<td>Whether the trip takes place on Wednesday or on a festive day/weekend</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>dayType</td>
<td>one of the portions of the day; the concept restriction is {peak hours, morning, afternoon}</td>
<td>STRING</td>
</tr>
<tr>
<td>tripId</td>
<td>The id of the trip</td>
<td>STRING</td>
</tr>
<tr>
<td>citizenId</td>
<td>The citizen taking part to the trip</td>
<td>STRING</td>
</tr>
<tr>
<td>startCoordinate</td>
<td>starting coordinate of an event</td>
<td>GEOMETRY</td>
</tr>
<tr>
<td>path</td>
<td>a line or route along which something travels or moves</td>
<td>[] GEOMETRY</td>
</tr>
<tr>
<td>endCoordinate</td>
<td>ending coordinate of an event</td>
<td>GEOMETRY</td>
</tr>
<tr>
<td>districtIdO</td>
<td>The origin of the trip</td>
<td>STRING</td>
</tr>
<tr>
<td>districtIdD</td>
<td>The destination district</td>
<td>STRING</td>
</tr>
</tbody>
</table>
3.2.5 Modal Split
Similarly to Citizen and Trip, Modal split is not represented in the FIWARE data models. According to [6], there is a lack of a unified approach for the modal split calculation in the scientific community. Generally speaking, modal split represents the ratio of different transport modes in the total journey from the origin to the destination. Notice that this is different from simply vehicles counting, where only information about traffic volume in a certain area is provided. In addition to modelling these notions, we accounted also for statistical elements of the modal split, such as its weight, value, and quality, together with trip-related information, specifically on its timing, citizen, and transportation mean(s) used; Table Table 3, The Modal Split eType presents the Modal Split eType.

Table 3. The Modal Split eType

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>districtIdO</td>
<td>the id of the origin district</td>
<td>STRING</td>
</tr>
<tr>
<td>means</td>
<td>list of means used to compute the modal split; the concept restriction is {car, bus, train, motorcycle, cable car, bicycle, foot}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>occupation</td>
<td>the principal activity in your life that you do to earn money; the concept restriction is {worker, student, other}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>districtIdD</td>
<td>the id of the destination district</td>
<td>STRING</td>
</tr>
<tr>
<td>quality</td>
<td>The quality of the measure of the modal split</td>
<td>INTEGER</td>
</tr>
<tr>
<td>value</td>
<td>a numerical quantity measured or assigned or computed</td>
<td>INTEGER</td>
</tr>
<tr>
<td>weight</td>
<td>the coefficient assigned to the modal split</td>
<td>FLOAT</td>
</tr>
<tr>
<td>citizenType</td>
<td>The type of citizen in QROWD; the concept restriction is {resident, commuter}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>partOfDay</td>
<td>The moment of the day; the concept restriction is {car, bus, train, motorcycle, cable car, bicycle, foot}</td>
<td>STRING</td>
</tr>
<tr>
<td>typicalDay</td>
<td>Whether the trip takes place on Wednesday or on a festive day/weekend</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>dayType</td>
<td>one of the portions of the day; the concept restriction is {peak hours, morning, afternoon}</td>
<td>STRING</td>
</tr>
</tbody>
</table>
Dynamic data integration, storage and access

dayofWeek | Any day of the week; the concept restriction is {monday, tuesday, wednesday, thursday, friday, saturday, sunday} | CONCEPT
--- | --- | ---
citizenId | The citizen taking part to the trip | STRING

3.2.6 Parking Infrastructure
The Parking Infrastructure eType represents all the types of parking spots available in the city of Trento, i.e., off-street parking sites with explicit entries and exits, on-street, free-entry (but might be metered) parking zone which contains at least one or more adjacent parking spots. Notice that the main reason to create this eType was to manage the shared attributes between the two types of parking and reduce repetition in the data. As such, this eType accounts for the shared attributes between the two parking, in addition to last occupancy measurement time, i.e., the last time the occupancy of the parking was calculated, and it is presented in Table 4. The Parking Infrastructure eType.

### Table 4. The Parking Infrastructure eType

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowedVehicleType</td>
<td>allowed type of vehicle; the concept restriction is {bicycle, bus, car, caravan, carWithCaravan, carWithTrailer, constructionOrMaintenanceVehicle, lorry, moped, motorcycle, motorcycleWithSideCar, motorscooter, tanker, trailer, van, anyVehicle}</td>
<td>[] CONCEPT</td>
</tr>
<tr>
<td>addressCountry</td>
<td>country of the parking</td>
<td>STRING</td>
</tr>
<tr>
<td>image</td>
<td>image of the parking infrastructure</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>availableSpotNumber</td>
<td>the number of spots available (including all vehicle types or reserved spaces)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>totalSpotNumber</td>
<td>the total number of spots in a bike rack</td>
<td>INTEGER</td>
</tr>
<tr>
<td>districtId</td>
<td>id of district</td>
<td>STRING</td>
</tr>
<tr>
<td>streetAddress</td>
<td>address of the street</td>
<td>STRING</td>
</tr>
<tr>
<td>addressLocality</td>
<td>locality of the parking</td>
<td>STRING</td>
</tr>
<tr>
<td>parkingZoneId</td>
<td>id of the parking zone</td>
<td>STRING</td>
</tr>
<tr>
<td>lastOccupancyMeasur</td>
<td>last occupancy measurement time</td>
<td>DATE</td>
</tr>
<tr>
<td>ementTime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>detectionType</td>
<td>type of detection; the concept restriction is</td>
<td>CONCEPT</td>
</tr>
</tbody>
</table>
Dynamic data integration, storage and access

{None,Balancing,Manual,modelBased,SingleSpaceDetection,CrowdSensed}

### 3.2.7 Onstreet Parking
According to the corresponding FIWARE data model, an on street parking is a site, open space zone, on street, (metered or not) with direct access from a road, intended to park vehicles. With respect to the actual FIWARE data model, this eType is a subset of attributes, shown in Table 5. The Onstreet Parking eType.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>addressCountry</td>
<td>country of the parking</td>
<td>STRING</td>
</tr>
<tr>
<td>image</td>
<td>image of the parking infrastructure</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>availableSpotNumber</td>
<td>the number of spots available (including all vehicle types or reserved spaces)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>totalSpotNumber</td>
<td>the total number of spots in a bike rack</td>
<td>INTEGER</td>
</tr>
<tr>
<td>districtId</td>
<td>id of district</td>
<td>STRING</td>
</tr>
<tr>
<td>streetAddress</td>
<td>address of the street</td>
<td>STRING</td>
</tr>
<tr>
<td>addressLocality</td>
<td>locality of the parking</td>
<td>STRING</td>
</tr>
<tr>
<td>parkingZoneId</td>
<td>id of the parking zone</td>
<td>STRING</td>
</tr>
<tr>
<td>onStreetTypeI</td>
<td>type I of on street parking; the concept restriction is {FAorDisabled, forResidents, forLoadUnload, onlyWithPermit, forELectricalCharging}</td>
<td>CONCEPT</td>
</tr>
<tr>
<td>onStreetTypeII</td>
<td>type II of the on street parking category; the concept restriction is {ForDisabled, forResidents, forLoadUnLoad, onlyWithPermit, forELectricalCharging}</td>
<td>CONCEPT</td>
</tr>
</tbody>
</table>

### 3.2.8 Offstreet Parking
According to the corresponding FIWARE data model, an off street parking is a site, off street, intended to park vehicles, managed independently and with suitable and clearly marked access points (entrances and exits). Table 6. The Offstreet eType shows the Offstreet Parking eType, which consists only of a subset of attributes.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>addressCountry</td>
<td>country of the parking</td>
<td>STRING</td>
</tr>
</tbody>
</table>
### 3.2.9 Bike Hire Docking Station

According to the corresponding FIWARE data model, a bike hire docking station is where subscribed users can hire and return a bike. Similarly to the other parking, its attributes, shown in Table 7. The Bike Hire Docking Station eType are a subset of the FIWARE data model entity.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>availableSpotNumber</td>
<td>the number of spots available (including all vehicle types or reserved spaces)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>totalSpotNumber</td>
<td>the total number of spots in a bike rack</td>
<td>INTEGER</td>
</tr>
<tr>
<td>districtId</td>
<td>id of district</td>
<td>STRING</td>
</tr>
<tr>
<td>streetAddress</td>
<td>address of the street</td>
<td>STRING</td>
</tr>
<tr>
<td>addressLocality</td>
<td>locality of the parking</td>
<td>STRING</td>
</tr>
<tr>
<td>parkingZoneId</td>
<td>id of the parking zone</td>
<td>STRING</td>
</tr>
<tr>
<td>security</td>
<td>security of the parking</td>
<td>[] STRING</td>
</tr>
<tr>
<td>offStreetTypeI</td>
<td>type of offstreet parking; the concept restriction is {public, private, publicPrivate, urbanDeterrentParking, parkingGarage, parkingLot, shortTerm, mediumTerm, longTerm, free, feeCharged, guarded, barrierAccess, gateAccess, freeAccess, forElectricalCharging, onlyResidents, onlyWithPermit, forEmployees, forVisitors, forCustomers, forStudents, forMembers, forDisabled, forResidents, underground, ground}</td>
<td>STRING</td>
</tr>
<tr>
<td>totalSlotNumber</td>
<td>the total number of slots</td>
<td>INTEGER</td>
</tr>
<tr>
<td>addressCountry</td>
<td>country of the parking</td>
<td>STRING</td>
</tr>
<tr>
<td>freeSlotNumber</td>
<td>number of free slots</td>
<td>INTEGER</td>
</tr>
<tr>
<td>image</td>
<td>image of the parking infrastructure</td>
<td>STRING</td>
</tr>
<tr>
<td>dateModified</td>
<td>date of the modification</td>
<td>DATE</td>
</tr>
<tr>
<td>districtId</td>
<td>id of district</td>
<td>STRING</td>
</tr>
<tr>
<td>availableBikeNumber</td>
<td>number of available bikes</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
3.2.10 Parking Group
The Parking Group eType, shown in Table 8. The Parking Group eType, represents a group of parking spots, whose granularity level can vary. It can be a storey on a parking garage, a specific area belonging to a big parking lot, etc. or just a group of spots, differentiated for a specific purpose (usage, restrictions, etc.).

Table 8. The Parking Group eType

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>the parking group identifier</td>
<td>UUID</td>
</tr>
<tr>
<td>Required permit</td>
<td>the type of required permit to access; the concept restriction is {residentPermit, noPermitNeeded, disabledPermit}</td>
<td>Concept</td>
</tr>
<tr>
<td>Permit Active Hours</td>
<td>the time slots during which the permit is required to access the parking group</td>
<td>Date</td>
</tr>
<tr>
<td>Total Spot Number</td>
<td>the total number of parking spots in the parking infrastructure</td>
<td>Integer</td>
</tr>
<tr>
<td>Available Spot Number</td>
<td>the currently available number of parking spots in the parking infrastructure</td>
<td>Integer</td>
</tr>
<tr>
<td>Last Occupancy Measure Time</td>
<td>the timestamp of the last time the occupancy was measured</td>
<td>Date</td>
</tr>
<tr>
<td>Image</td>
<td>an image of the parking infrastructure, which is to be collected by citizens</td>
<td>varchar</td>
</tr>
</tbody>
</table>

4. BIG DATA STORAGE
As Big Data is a central element of QROWD, the database technology required to deal with them is clustered under the umbrella terms of NoSQL databases [4], also called “Not Only SQL”, which is an approach for large and distributed data management and database design. While it is true that some NoSQL systems are entirely non-relational, others avoid selected relational functionality such as fixed table schemas and join operations. The mainstream Big Data platforms adopt NoSQL to break and transcend the rigidity of normalized RDBMS schemas. To store and manage unstructured data or non-relational data, NoSQL employs some specific approaches. Firstly, data storage and management are separated into two
Dynamic data integration, storage and access independent parts, which is the opposite of relational databases, which try to meet the concerns in the two sides simultaneously. This design gives NoSQL databases systems a lot of advantages. In the storage part which is also called key-value storage, NoSQL focuses on the scalability of data storage with high-performance. In the management part, NoSQL provides low-level access mechanism in which data management tasks can be implemented in the application layer rather than having data management logic spread across in SQL or DB-specific stored procedure languages. Therefore, NoSQL systems are very flexible for data modeling and easy to update application developments and deployments [4]. Most NoSQL databases have a valuable property, i.e., they are commonly schema-free. Indeed, the most significant advantage of schema-free databases is that it enables applications to modify the structure of data quickly and does not need to rewrite tables. Additionally, it possesses greater flexibility when the structured data is heterogeneously stored. In the data management layer, the data is enforced to be integrated and valid. The most popular NoSQL database is Apache Cassandra and is the one we are currently using in Qrowd-DB.

4.1 Apache Cassandra: distributed, scalable database system
Apache Cassandra\(^9\) is a free and open-source, distributed, wide column store, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple data centers, with asynchronous masterless replication allowing low latency operations for all clients. It was created at Facebook to power its inbox search feature. Facebook released Cassandra as an open-source project on Google code in July 2008. In March 2009 it became a project of the Apache foundation. The Apache Cassandra database is the right choice when you need scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra’s support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages. We found that Apache Cassandra was a perfect fit for the needs of the big data storage used in QROWD for three main reasons:

1. As said above, it is very popular and it is the best database [10] for large critical sensor applications;
2. Its eventual consistency model allows us to tune the consistency level depending on the required performances, ideally at the query level;
3. The fact that it is a distributed database allows the system to scale linearly with respect to the number of nodes in the cluster. When the number of users (and data) increases, there is the possibility to add nodes to the cluster to increase the capacity and maintain the performances;
4. It allows us to create replicas of the nodes for backups easily.

4.2 Data schema to enable anonymization
All the advantages of Cassandra come at a price, i.e., the data schema must be

\(^9\) http://cassandra.apache.org/
Dynamic data integration, storage and access
designed in a very fine-grained way. Cassandra's eventual consistency model allows
for high performances only on a limited set of queries, e.g., joins among tables are
not allowed. In terms of the design, this means that the data schema must be
created so that the application can reply to a query with a single database read
operation, otherwise, it gets slow. The database architect must design the schema
according to a query-based rationale. Data duplication is advisable in these
situations if, from the same data, there is the need to reply to multiple queries: disk
space is cheap, while fast reads are challenging to obtain.
In the system, the data schema was designed according to these considerations:

1. A Cassandra keyspace is associated with one single user. This allows us to
have different consistency strategies for different users and, most importantly,
will enable us to isolate the data for privacy concerns. If every user's data is
saved in a separate keyspace is easier to deal with the data, e.g., delete them
if the user wants to uninstall the application. The anonymization is granted at
this level since the name of the keyspace is a 160bit salt string generated
randomly using the Secure Hash Algorithm\(^\text{10}\) (SHA-1) when the user
subscribes to the system through the i-Log mobile application (D2.4). All the
data exchange, be it analysis in the system or with the partners (data
processors), occurs through this anonymized identifier. Thus, the users'
personal data (the email address) in never used in this regard. Both data, the
salt, and the email are stored in a disambiguation table that is accessible only
by the data controllers (University of Trento and Municipality of Trento). The
email can be used to contact the users in some situations they previously
agreed on, e.g., collect the prices of the pilots or help by providing feedback.

2. There is one table per query we need to reply per sensor. Since we are
dealing with time series, we chose to allow querying the data by time and in
some limited cases also by value. In time series most of the time a client
application needs to have the values in a time interval, e.g., the accelerometer
data to understand if the user is moving from 08:00 AM to 10:00 AM. In less
common situations, we would like to query by value, e.g., to understand is the
user previously visited a specific location.

3. We defined a replication factor for the Cassandra cluster of three, that is,
every piece of data is replicated on three different nodes.

4.3 Database physical infrastructure
As explained before, Cassandra is a distributed database system which performs
better when multiple nodes are part of the cluster. If the goal is to store a small
amount of data that can eventually be stored on a single machine, then Cassandra
may be not the best solution. Having it distributed on multiple nodes allows to exploit
all its potential, namely the eventual consistency model, the linear scalability, among
others. If then, its distributed fashion is an advantage, at the same time it makes it
harder to install, configure and maintain it since many components and machines
come into play (See Annex 3 - Big data storage technology stack for an overview of
them), and additional aspects have to be considered and tuned with respect to a
non-distributed system. Among them, we have to consider the data center where the

\(^{10}\) https://en.wikipedia.org/wiki/SHA-1
Dynamic data integration, storage and access
physical machines composing the cluster are located, the connections between the machines and those between the database and the final user and finally the configuration of the database itself.

4.4.1 Data center
The cluster that the University of Trento has allocated to the QROWD project is located inside the newly created datacenter\textsuperscript{11}. It is a Tier 3 data center, with a total of 55 racks distributed across 225 square meters of surface area. The power supply is guaranteed by a ~1MW line available plus a redundant system of two UPS that can provide 125KW each for a total of 30 minutes of uptime. The free cooling air-conditioning system in the rooms is supplied by two chiller units assisting the eight machines in the racks room with chilling water. Each machine draws in warm air from the environment, cools it and then forces it beneath the raised floor (80 cm), which supplies the conditioned air into the separate zones of the building. The goal is to store cool air in the central aisle so that the units could draw in conditioned air and cool their internal components, and finally expel it from the isolated zones. The infrastructure has all the necessary elements to guarantee the security both from physical but also virtual intruders, in accordance with the latest regulations. With such infrastructure, the University of Trento guarantees a 99.982% uptime, with up to 1.6 hours of downtime a year.

4.4.2 Network Configuration
In a distributed system the network component is a core element that has to work properly. Otherwise, every piece of the system cannot communicate efficiently. In Cassandra, a non-working network can create delays and trigger timeouts that will cause the writing/reading operations failures. The most critical network component is the one that connects the nodes of the Cassandra cluster, which means, the one inside the datacenter. In the data center of the University of Trento\textsuperscript{12}, there are:

- 960 x 1Gbit ports
- 192 x 10Gbit ports
- 64 x 40Gbit ports
- Uplink 80/160Gbit

All the connections are made with 12km of UTP cat 6A cable and 2km of optical fibers. The machines dedicated to the QROWD project are linked with 1Gbit links, eventually upgradable to 10Gbit links, if necessary.

At the same time, also the external network is vital if the clients and systems that use the database are not physically located in the University of Trento, like in QROWD. The main flow of data in the project occurs between the cluster and the partners in Leipzig, while minimal exchanges also happen to the others in Southampton and Madrid. The link with these institutions is guaranteed by the university network\textsuperscript{13}, that connects the different building with up to 1Gbit/s links and a redundant wireless link that works up to 700Mbit/s. Then, the whole university is connected to the main

\textsuperscript{11} https://icts.unitn.it/en/datacenter
\textsuperscript{12} https://icts.unitn.it/en/man-the-university-network
\textsuperscript{13} https://icts.unitn.it/en/man-the-university-network
Dynamic data integration, storage and access public networks thanks to a link with the GARR\textsuperscript{14} consortium at 10Gbit/s. GARR is the national ultra broadband network dedicated to research institutions that is also part of the european network GEANT\textsuperscript{15} and is linked with other international networks, such as IETF\textsuperscript{16}, Internet Society\textsuperscript{17} e e-IRG\textsuperscript{18}.

4.4.3 Dedicated Hardware
For QROWD we created a dedicated Cassandra cluster composed of twelve nodes, hosted on three physical machines stored in the datacenter of the University of Trento. The characteristics of the machine are as follows:
- Dell PowerEdge R640
- Intel Xeon 2x6126 CPU with 48 total threads
- 192GB of RAM
- 2x150GB SSD
- 16TB HDD storage

On each machine we installed Kubernetes\textsuperscript{19} (more details provided in Annex 3 - Big data storage technology stack) so that to be able to orchestrate the containers containing the 12 Cassandra nodes automatically. Kubernetes uses the three physical servers as a unique entity and balances the load among them by moving the nodes from one to the other if needed. This solution also facilitates the situation in which we would need to add additional nodes to this cluster due to increases in the load and/or amount of data.

5. The Business Logic
This component in Qrowd-DB is the one that “facilitates the discovery and exploitation of data in specific processes and services” as per the GA. It consists of a set of software API that communicates with the underlying systems, the linked data, and the big data storage, and provide the desired information to the QROWD partners. Even if we present it as a single component from the outside, each storage has a specific business logic since the type of data have very different characteristics and need different solutions and technologies. In the next Sections 5.1 Linked Data Storage Business Logic and 5.2 Big data Storage Business Logic we present each API.

5.1 Linked Data Storage Business Logic
This section provides the description of the main API\textsuperscript{20} calls that are used to update and obtain information about the different entities stored in Qrowd-DB. This section will first explain the main groups of calls and their purpose to then focus on the Modal Split call, as an example, while the remaining calls are listed in Annex 1 - Linked data storage API reference.

\textsuperscript{14} https://www.garr.it/it/
\textsuperscript{15} https://www.geant.org/
\textsuperscript{16} https://www.ietf.org/
\textsuperscript{17} http://www.internetsociety.org/
\textsuperscript{18} http://e-irg.eu/
\textsuperscript{19} https://kubernetes.io/
\textsuperscript{20} http://streambase1.disi.unitn.it:3000/api-docs
5.1.1 Main call groups
The following are considered the main groups of calls in the API:

- **Citizen**: creation and retrieval of citizen entities. These contain information about the persons participating in the experiment (e.g., gender, possession of a driving licence, occupation, and awarded points).
- **Trip**: creation and retrieval of trip entities and execution of the ModalSplit. Trip entities contain information about the trip that citizens perform (e.g. origin, destination and type of day) and it also includes GPS coordinates for it.
- **Parking**: retrieval of parking entities. Parking entities contain information about places for parking of vehicles (e.g. position or number of free slots for parking).
- **Emulated**: renamed APIs that are functionally the same as the previous ones but with names and parameter configurations that better match the uses that will be given by the partners using them. As such, these could be considered merely "an alias" of the previous calls done to facilitate integration.

5.1.2 Modal Split
We explain the modal split call in the main text of the deliverable as an example and because we believe it to be one of the key calls in the API. The rest of the API calls are described in the Annex I.

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/modalSplit</td>
<td>Performs the Modal Split analysis over a period of time and/or for a specific citizen.</td>
</tr>
</tbody>
</table>

**Call Example**


**Response Example**

```json
{
  "quality": 0,
  "factors": [
    {
      "value": 0,
      "type": "car",
      "weight": 0
    }
  ]
}
```

**Response General**

```json
{
  "description": internal representation id
  "quality": number
}
```
Dynamic data integration, storage and access

```json
factors: [
  {value: number, type: string, title: 'Transportation',
   Enum: ['car', 'bus', 'train', 'motorcycle', 'cablecar', 'bicycle', 'foot'],
   weight: number},
]}
```

5.2 Big data Storage Business Logic

5.2.1 Architecture

A simplified version of the architecture of the business logic for the big data storage component is presented in Figure 4. Big Data Storage business logic architecture.

![Figure 4. Big Data Storage business logic architecture](image)

It is composed by multiple API modules, each one acting as a container in the Kubernetes cluster running on top of three physical machines. Each API provides specific functionalities to the final users, i.e., through the i-Log mobile application or reply to queries by the partners in QROWD, or fulfill some tasks needed by the system itself. More in details, the Smartphone-to-server module (Section 5.2.2 Smartphone-to-server module) is responsible for handling the communication from the smartphone to the server, which is, user registration, login, data collection, among others. The Database Insert Module (Section 5.2.3 Database insert module) instead, has no communication to and from outside the system. Its main role is to receive the data logs from the Data API Module and load them into the internal
Dynamic data integration, storage and access
Cassandra database. Finally, there is the Server-to-smartphone module (Section 5.2.4 Server-to-smartphone module) that mainly manages the communications from the server to the smartphones based on events in the Cassandra database or external triggers, triggered by the partners in the project after some data processing task, for example. Each module in the architecture is a container orchestrated by Kubernetes, that can automatically duplicate and rebalance in presence of high load to efficiently fulfil the incoming requests.
There are other components in the system, but for the sake of clarity, we decided to omit them in this document since they fulfill various internal tasks that are not relevant in this context.

5.2.2 Smartphone-to-server module
The main goal of this API module is to enable the communication from smartphones to the backend system. It handles all the calls triggered by the user on the i-Log mobile application with GET and POST that will be presented in Annex 2 - Big data storage API reference.
Every call made by the users has to be authenticated so that we know precisely who is calling and with which information we have to reply. After different iterations and versions of the authentication mechanism, we decided to use a service provided by Google called Google Sign-In\(^{21}\) provided as part of the Google Identity Platform. This service allows the secure authentication of a user on a backend server. This is guaranteed by the Google Sign-In module that generates a token on the device, associated with the specific user, device, and application, and then this token can be freely sent to the backend server in every call and decoded at the other side to detect the user identity. The token can be decoded only by the application linked with the mobile application which generated it. In this way, there is no need to exchange personal identifiers of the users at every call from the smartphone to the server, like the email or the anonymized identifier. We can even not store on the device any user information, which is a process that can lead to possible privacy issues if the phone is lost or stolen. The usage of this method also simplifies the way the user subscribes and registers on the smartphone, using her already configured Google account (more details about this can be found in D2.4). Once the token is received on the server the system checks if it is a valid token (which means, signed by the i-Log mobile application) and if yes, it retrieves the anonymized identifier and satisfies the API call.
Among the different endpoints available, in this module, we have the one that collects the logs containing the raw data obtained from the sensors embedded in the mobile devices of the users. Once received, the log is passed to the database insert module that processes it and inserts the values in the Cassandra database.

5.2.3 Database insert module
The main purposes of this component is to accept the logs of data collected by the Smartphone-to-server module, process them and store them in the Cassandra database system so that to made them available.
In the i-Log mobile application the data is collected in logs files, which are .csv

\(^{21}\) https://developers.google.com/identity/
Dynamic data integration, storage and access format compressed using the .bzip2 algorithm (more details about this are provided in D2.4). Once the logs are sent to the server, some operations need to be performed on them before their storing:

- **Uncompress**: the first operation to be performed is to uncompress the data, in memory. The data was previously compressed to save space on the device but, most importantly, to have fewer data to be transferred over the network, which is an energy consuming task and that could also impact the user’s data plan by draining the available traffic;

- **Pre-processing**: after the uncompression, some of the data need to be pre-processed. This is mainly because on the smartphone we need to reduce the computational power required and reduce the battery consumption and at the same time to save space. An example of pre-processing operation is the one performed on the data about the user motion, e.g., the accelerometer. These data are generated with up to 10 decimals, something that is useless during the analysis phases. In fact, two decimals are more than enough. By removing the remaining eight decimals, we can save a considerable amount of disk space considering that these sensors generate up to 100 values per second. Thus, on the smartphone, we need to remove those digits efficiently: the most efficient way is to multiply the decimal value by a multiple of then based on how many digits we want to save and trim this to an integer. For example, 9,8189829839 * 100 gives us 981,89829839 and taking the integer part returns 981. Then, on the server, we need to convert this value back to a float, dividing by 100. This is only one example of the pre-processing operation that is performed on the server.

- **Schema matching**: an additional step to be done is to convert each line of the csv resulting from the previous step and match it with the schema of the Cassandra tables.

- **Insert**: finally, the data can be pushed to the database and finally persisted. Multiple threads pushes one single line, in parallel, to speed up the operations. We use up to 1000 threads that with the current configuration for the cluster allow us to insert up to ~110000 values per second. To put this into context, considering an average of 250 values per second per user, at the current time we can manage the load of 400 users. This process could be eventually sped up using alternative technologies such as Apache Spark (explained in Annex 2 - Apache Spark). With it, we can parallelize even further the insert operation and make it faster. We tested this solution previously, and the results were promising since we could reach ~80000 operations per second with a smaller Cassandra cluster.

### 5.2.4 Server-to-smartphone module

This module is used by the other components of the system to send contents to the smartphone. Among the many different possibilities available to do so we decided to leverage on an existing service called Firebase Cloud Messaging\(^{22}\), provided by Google as part of Google Firebase\(^{23}\). The main reason that led us to choose it is the

---

\(^{22}\) https://firebase.google.com/docs/cloud-messaging/

\(^{23}\) https://firebase.google.com/
Dynamic data integration, storage and access

fact that it is embedded in all the Android device since it is something native in the operating system. Its benefits are multiple:

- Google manages all the routing from their servers to the smartphone;
- The delivery process is more efficient for the smartphone, meaning that if the phone is in sleep mode, it is not disturbed by the message but instead the message is delivered once it gets back in active mode;
- Is more efficient because multiple messages from multiple applications are delivered by Google at the same time;
- If the device is not connected when the message is sent, Google delivers it as soon as it comes back online;
- It is privacy compliant, we do not need any identifier of the user to send them messages. Instead, we need an identifier Google creates for that instance of i-Log installed (if the same user installs i-Log more then once in different moments in time, she will receive different tokens, meaning that the address is not used to identify the person).

With Google Firebase it is much easier and efficient to send data from the server to the smartphone. In QROWD we use this component to send messages to remotely configure i-Log and mainly to push to the users the questions for the hybrid workflows (as described in deliverable D3.2), especially the tasks about modal split (more details in deliverable D2.4). A message can be sent based on two situations:

- A script analyses the data and reacts based on the result (feature not used in QROWD);
- External triggers through the API embedded in the module. This is the main way this component is used, especially in the hybrid workflows, where the other QROWD members access the raw sensor data in Cassandra directly, compute the results with their scripts, and finally send the results (questions) to the user’s smartphones. Details about the available API are presented in Annex 2 - Server-to-smartphone module.
6. CONCLUSIONS

In this deliverable, we described the QROWDDB, its components and how they manage and store the data collected throughout the QROWD project. The two main components of the QROWDDB are the Big Data Storage and the Linked Data Storage. The former is the database dedicated to managing the sensor streams and the crowdsourcing data from the WP2 business cases involving the citizens, while the latter manages all the knowledge about the QROWD project in terms of entities involved in the pilots, be they citizens or mobility-related entities, such as parking areas. Both the components can be accessed via dedicated APIs, which were described in detail. Overall, the QROWDDB represents a pivotal element of the overarching QROWD architecture for carrying out the action, and in general one of the main contributions of the project itself.
REFERENCES


ANNEX 1 - LINKED DATA STORAGE API REFERENCE

This annex describes the main calls from the Qrowd-DB API implemented by UNITN.

GET Citizen

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/citizen</td>
<td>Search citizen by id, when id is not provided it returns all citizens</td>
</tr>
</tbody>
</table>

Call Example

http://streambase1.disi.unitn.it:3000/api/v1/citizen/1

Response Example

```json
[
  {
    "citizenId": "test2",
    "age": 99,
    "gender": "male",
    "occupation": "student",
    "citizenType": "commuter",
    "drivingLicense": true,
    "numberCohabitants": 1,
    "numberVehicles": 3,
    "preferredMode": "bus",
    "pointsAwarded": 5,
    "address": {
      "addressCountry": "italy",
      "addressLocality": "trento",
      "streetAddress": "mia strada"
    }
  }
]
```

Response General

```json

<table>
<thead>
<tr>
<th>Citizen</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>citizenId</td>
<td>string</td>
</tr>
<tr>
<td>age</td>
<td>number</td>
</tr>
<tr>
<td>gender</td>
<td>stringEnum: [ male, female ]</td>
</tr>
<tr>
<td>occupation</td>
<td>stringEnum: [ student ]</td>
</tr>
<tr>
<td>citizenType</td>
<td>stringEnum: [ commuter ]</td>
</tr>
<tr>
<td>drivingLicense</td>
<td>boolean</td>
</tr>
<tr>
<td>numberCohabitants</td>
<td>number</td>
</tr>
<tr>
<td>numberVehicles</td>
<td>number</td>
</tr>
<tr>
<td>preferredMode</td>
<td>stringEnum: [ bus ]</td>
</tr>
</tbody>
</table>
```
Dynamic data integration, storage and access

```json
pointsAwarded: number
address: {
    addressCountry: string
    addressLocality: string
    streetAddress: string
}
}
```

**POST Citizen**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/v1/citizen</td>
<td>Create a new citizen</td>
</tr>
</tbody>
</table>

**Call Example**

http://streambase1.disi.unitn.it:3000/api/v1/citizen

**Request Body Example**

```json
{
    "citizenId": "test2",
    "age": 99,
    "gender": "male",
    "occupation": "student",
    "citizenType": "commuter",
    "drivingLicense": true,
    "numberCohabitants": 1,
    "numberVehicles": 3,
    "preferredMode": "bus",
    "pointsAwarded": 5,
    "address": {
        "addressCountry": "italy",
        "addressLocality": "trento",
        "streetAddress": "mia strada"
    }
}
```

**Request Body General**

```json
Citizen{
    citizenId: string
    age: number
    gender: string
    occupation: string
    citizenType: string
    drivingLicense: boolean
    numberCohabitants: number
    numberVehicles: number
}
```
Dynamic data integration, storage and access

```
preferredMode: string
  enum: [bus]

pointsAwarded: number

address:
  addressCountry: string
  addressLocality: string
  streetAddress: string
```

**GET Trip**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/trip</td>
<td>Get Trips filtering by attributes</td>
</tr>
</tbody>
</table>

**Call Example**

```
```

**Response Example**

```
[
  {
    "tripId": "testTrip1",
    "citizenId": "test1",
    "typicalDay": false,
    "dayType": "working",
    "partOfDay": "morning",
    "startCoordinate": {
      "type": "Point",
      "coordinates": [
        12.333,
        12.55
      ]
    },
    "districtIdO": "string",
    "endCoordinate": {
      "type": "Point",
      "coordinates": [
        12.34,
        12.55
      ]
    },
    "districtIdD": "string",
    "initDate": "1970-01-16T04:45:23.123Z",
    "endDate": "1970-01-15T06:00:31.231Z",
    "meansL": "bus",
    "confidenceValue": 12.2,
    "path": {
      "type": "LineString",
      "coordinates": [
        [D7.3]
      ]
    }
  }
]`
```
Dynamic data integration, storage and access

```
12.34,
12.55
],
[ 12.35,
12.57
],
[ 12.34,
12.55
]
},
"multitrip": false,
"subtrips": []
}
```

### Response General

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trip</td>
<td></td>
</tr>
<tr>
<td>tripId</td>
<td>string</td>
</tr>
<tr>
<td>citizenId</td>
<td>string</td>
</tr>
<tr>
<td>required: true</td>
<td></td>
</tr>
<tr>
<td>typicalDay</td>
<td>boolean</td>
</tr>
<tr>
<td>dayType</td>
<td>string</td>
</tr>
<tr>
<td>Enum: [ ]</td>
<td></td>
</tr>
<tr>
<td>partOfDay</td>
<td>string</td>
</tr>
<tr>
<td>Enum: [ ]</td>
<td></td>
</tr>
<tr>
<td>startCoordinate</td>
<td>Point</td>
</tr>
<tr>
<td>type</td>
<td>string</td>
</tr>
<tr>
<td>required: true</td>
<td></td>
</tr>
<tr>
<td>example: POINT</td>
<td></td>
</tr>
<tr>
<td>coordinates</td>
<td>[...]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>districtIdD</td>
<td>string</td>
</tr>
<tr>
<td>endCoordinate</td>
<td>Point</td>
</tr>
<tr>
<td>type</td>
<td>string</td>
</tr>
<tr>
<td>required: true</td>
<td></td>
</tr>
<tr>
<td>example: POINT</td>
<td></td>
</tr>
<tr>
<td>coordinates</td>
<td>[...]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>districtIdD</td>
<td>string</td>
</tr>
<tr>
<td>initDate</td>
<td>string</td>
</tr>
<tr>
<td>($date-time)</td>
<td></td>
</tr>
<tr>
<td>endDate</td>
<td>string</td>
</tr>
<tr>
<td>($date-time)</td>
<td></td>
</tr>
<tr>
<td>meansL</td>
<td>Transportationstring</td>
</tr>
<tr>
<td>title: TransportationEnum: [ car, bus, train, motorcycle, cablecar, bicycle, foot ]</td>
<td></td>
</tr>
<tr>
<td>confidenceValue</td>
<td>numeric</td>
</tr>
<tr>
<td>multitrip</td>
<td>boolean</td>
</tr>
<tr>
<td>subtrips</td>
<td>[string]</td>
</tr>
</tbody>
</table>
```
Dynamic data integration, storage and access

```json
path

  { oneOf ->

    Linestring{ type string
      required: true
      example: LINESTRING
      coordinates [...]
    }

    Linestring{ type string
      required: true
      example: LINESTRING
      coordinates [...]
    }
  }
```

**POST Trip**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/v1/trip</td>
<td>Create new trip.</td>
</tr>
</tbody>
</table>

**Call Example**

http://streambase1.disi.unitn.it:3000/api/v1/trip

**Request Body Example**

```json
[
  {
    "tripId": "testTrip1",
    "citizenId": "test1",
    "typicalDay": false,
    "dayType": "working",
    "partOfDay": "morning",
    "startCoordinate": {
      "type": "Point",
      "coordinates": [
        12.333,
        12.55
      ]
    },
    "districtIdO": "string",
    "endCoordinate": {
      "type": "Point",
      "coordinates": [
```
Dynamic data integration, storage and access

```json
{
  "tripId": "string",
  "citizenId": "string",
  "required": true,
  "typicalDay": boolean,
  "dayType": stringEnum:
    [ ]
  "partOfDay": stringEnum:
    [ ]
  "startCoordinate": Point{
    "type": string
    "required": true
    "example": POINT
    "coordinates": [ ]
    "required": true
    "minItems": 2
    "maxItems": 3
    "string"
  }
  "districtIdD": "string",
  "initDate": "1970-01-16T04:45:23.123Z",
  "endDate": "1970-01-15T06:00:31.231Z",
  "meansL": "bus",
  "confidenceValue": 12.2,
  "path": {
    "type": "LineString",
    "coordinates": [
      12.34,
      12.55,
      12.35,
      12.57,
      12.34,
      12.55
    ],
    "multitrip": false,
    "subtrips": []
  }
}
```

### Request Body General

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tripId</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>citizenId</td>
<td>string</td>
<td>required: true</td>
</tr>
<tr>
<td>typicalDay</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>dayType</td>
<td>stringEnum</td>
<td>[ ]</td>
</tr>
<tr>
<td>partOfDay</td>
<td>stringEnum</td>
<td>[ ]</td>
</tr>
<tr>
<td>startCoordinate</td>
<td>Point</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>string</td>
<td>required: true</td>
</tr>
<tr>
<td>example</td>
<td>POINT</td>
<td></td>
</tr>
<tr>
<td>coordinates</td>
<td>[ ]</td>
<td></td>
</tr>
<tr>
<td>districtIdD</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic data integration, storage and access

```json
endCoordinate: Point{
  type: string
  required: true
  example: POINT
  coordinates: [
    required: true
    minItems: 2
    maxItems: 3
    string
  ]
}
districtIdD: string
initDate: string($date-time)
endDate: string($date-time)
meansL: Transportation
  title: Transportation
  Enum:
    [ car, bus, train, motorcycle, cablecar, bicycle, foot ]
confidenceValue: numeric
multitrip: boolean
subtrips: [string]
path: }
  oneOf ->
    { maxItems: 1
      Linestring{
        type: string
        required: true
        example: LINESTRING
        coordinates: [...] }]
Linestring{
  type: string
  required: true
  example: LINESTRING
  coordinates: [...] ]}
}
```

**GET Modal Split**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/modalSplit</td>
<td>Performs the Modal Split analysis over a period of time and/or for a specific citizen.</td>
</tr>
</tbody>
</table>

**Call Example**

```
```

**Response Example**

```
{
  "quality": 0,
  "factors": [
```
Dynamic data integration, storage and access

```json
{
   "value": 0,
   "type": "car",
   "weight": 0
}
}
```

**Response General**

```json
{
   description: internal representation id
   quality
   factors
   [{
      value
      number
      type Transportation
      title: TransportationEnum:
      [car, bus, train, motorcycle, cablecar, bicycle, foot]
      weight
      number
   }]
}

**GET Parking**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/parking</td>
<td>Get parking filtering by ID</td>
</tr>
</tbody>
</table>

**Call Example**

http://streambase1.disi.unitn.it:3000/api/v1/parking/1

**Response Example**

```json
[
   {
      "id": 0,
      "totalSlotNumber": 0,
      "address": {
         "addressCountry": "string",
         "addressStreet": "string",
         "addressLocality": "string"
      },
      "freeSlotNumber": 0,
      "allowedVehicleType": "bicycle",
      "image": "string",
      "districtId": 0,
      "security": "string",
      "availableBikeNumber": 0,
      "availableSpotNumber": 0,
      "totalSpotNumber": 0,
      "offStreetTypeI": "string",
      "onStreetTypeI": "string"
   }
]```
Dynamic data integration, storage and access

```json
"onStreetTypeII": "string",
"detectionType": "None",
"lastOccupancyMeasurementTime": "2018-10-19T09:05:14.676Z",
"dateModified": "2018-10-19T09:05:14.676Z",
"location": {
"type": "POINT",
"coordinates": [
"string"
]
},
"type": "OnStreetParking"
}
```

### Response General

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>number</td>
<td>Read Only: true</td>
</tr>
<tr>
<td>totalSlotNumber</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>address</td>
<td>addressCountry</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>addressStreet</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>addressLocality</td>
<td>String</td>
</tr>
<tr>
<td>freeSlotNumber</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>allowedVehicleType</td>
<td>string</td>
<td>Vehicles Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>title: Vehicles Type: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enum: [ bicycle, bus, car, caravan, carWithCaravan, carWithTrailer,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>constructionOrMaintenanceVehicle, lorry, moped, motorcycle,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>motorcycleWithSideCar, motorscooter, tanker, trailer, van,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>anyVehicle ]</td>
</tr>
<tr>
<td>image</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>districtId</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>security</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>availableBikeNumber</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>availableSpotNumber</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>totalSpotNumber</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>offStreetTypeI</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>onStreetTypeI</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>onStreetTypeII</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>detectionType</td>
<td>string</td>
<td>Enum: [ None, Balancing, Manual, ModelBased, SingleSpaceDetection,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CrowdSensed ]</td>
</tr>
<tr>
<td>lastOccupancyMeasurementTime</td>
<td>string</td>
<td>(date-time)</td>
</tr>
<tr>
<td>dateModified</td>
<td>string</td>
<td>(date-time)</td>
</tr>
<tr>
<td>location</td>
<td>Point</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>required: true</td>
<td></td>
</tr>
<tr>
<td></td>
<td>example: POINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>coordinates</td>
<td></td>
</tr>
<tr>
<td></td>
<td>required: true</td>
<td></td>
</tr>
<tr>
<td></td>
<td>minItems: 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>maxItems: 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>
```
Dynamic data integration, storage and access

```yaml
  type: string
  required: true
  enum:
    - OnStreetParking
    - OffStreetParking
    - BikeHireDockingStation
```

**Emulated API Calls**
The following calls are also supported by the API to aid to the better integration of Qrowd-DB with other component but do not offer significant changes from the ones previously described:

- GET basicModalSplit
- GET getBikeHireDockingStation
- GET getOffStreetParking
- GET getOnStreetParking
- GET getPersonalTrips
- GET personalModalSplit
- GET temporalModalSplit
Dynamic data integration, storage and access

**ANNEX 2 - BIG DATA STORAGE API REFERENCE**

In this annex we present the complete list of API calls of the three different modules currently used by the business logic of the big data storage of Qrowd-DB.

**Smartphone-to-server module**

At the time of writing this document this container and its API are served on port 8090.

**GET Online**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/online</td>
<td>Returns a positive message if the system is online and able to receive incoming traffic. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/online

**Parameters**

```json
{
  "email": string
  required: true

  "password": string
  required: true
}
```

**Response General**

```json
{
  "done_message": string
}
```

**GET Projects**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/projects</td>
<td>Method that returns the information about a project defined by its code, if it exists.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/projects

**Parameters**

```json
{
  "code": string
  required: true
}
```
### Dynamic data integration, storage and access

#### Response General

<table>
<thead>
<tr>
<th>Project</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>id</strong></td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>title</strong></td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>configuration</strong></td>
<td>JSONObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>description</strong></td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>logo</strong></td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>informedconsent</strong></td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>permissions</strong></td>
<td>JSONObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>profile</strong></td>
<td>JSONObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sensors</strong></td>
<td>JSONObject</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### GET Login

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/login</td>
<td>Method that manages the login operation on the smartphone. The token is the Google Sign-In one and the firebasetoken is the one used to send messages to the device.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/login

**Parameters**

```
{
  token: string
  required: true
  firebasetoken: string
  required: true
}
```

**Response General**

```
{
  status: string
  values: done_message,
  error_message
  payload: string
}
```
Dynamic data integration, storage and access

**GET Refresh Google Cloud Messaging token**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/refreshgcmtoken</td>
<td>Method called when the Google services on the smartphone update the firebasetoken.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/refreshgcmtoken

**Parameters**

```
{
    token                    string
                      required: true
    firebasetoken           string
                      required: true
}
```

**Response General**

```
{
    status                   string
                      values: done_message,
                      error_message
    payload                  string
}
```

**GET Check user data**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/checkuserdata</td>
<td>Method that returns the presence of user data for every user enrolled in a project on a specific date. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/checkuserdata

**Parameters**

```
{
    email                    string
                      required: true
    password                 string
                      required: true
    dates                    string
                      required: true
    project                 string
                      required: true
}
```
Dynamic data integration, storage and access

**Response General**

| status string  
| values: done_message, error_message  
| payload JSONObject |

**GET Signup**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/signup</td>
<td>Method called by a smartphone if the user is not registered yet.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/signup

**Parameters**

| token string  
| required: true  
| firebasetoken string  
| required: true  
| project string  
| required: true  
| sensors JSONObject  
| required: true |

**Response General**

| status string  
| values: done_message, error_message  
| payload string |

**POST Upload**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/upload</td>
<td>Method called to upload the log files with the raw sensor data. The token is the Google Sign-In one.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/upload

**Parameters**
Dynamic data integration, storage and access

```json
{
  "token" : string
      required: true

  "file" : MultipartHttpServletRequest
      required: true
}
```

### Response General

```json
{
  "status" : string
      values: done_message,
      error_message

  "payload" : string
}
```

---

**Database insert module**

At the time of writing this document this container and its API are served on port 8090 inside the Kubernetes Flannel network.

---

**GET Online**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/online</td>
<td>Returns a positive message if the system is online and able to receive incoming traffic. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/online

**Parameters**

```json
{
  "email" : string
      required: true

  "password" : string
      required: true
}
```

**Response General**

```json
{
  "done_message" : string
}
```

---

**GET Process files with errors**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/processerrors</td>
<td>When a log file cannot be processed completely and not</td>
</tr>
</tbody>
</table>
Dynamic data integration, storage and access

all its entries are insert into Cassandra, it is stored in a tmp folder as an error file. This endpoint allows to try again to insert these logs in the database. Only the administrator can call this method and has to authenticate with his user/password.

Call Address

http://streambase1.disi.unitn.it:port/processerrors

Parameters

```
{
    email: string, required: true
    password: string, required: true
}
```

Response General

```
{
    done_message: string
}
```

Server-to-smartphone module

At the time of writing this document this container and its API are served on port 8095.

GET Online

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/online</td>
<td>Returns a positive message if the system is online and able to receive incoming traffic. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

Call Address

http://streambase1.disi.unitn.it:port/online

Parameters

```
{
    email: string, required: true
    password: string, required: true
}
```

Response General

```
{
    done_message: string
}
```
## POST Challenge definition

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/createchallengedefinition</td>
<td>Method to post a challenge definition object. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

### Call Address

http://streambase1.disi.unitn.it:port/createchallengedefinition

### Parameters

```json
{
    email: string
    required: true

    password: string
    required: true

    object: ChallengeDefinition{
        type: string
        required: true
        values: freeroam, static

        name: string
        required: true

        description: string
        required: true

        instructions: string
        required: true
        values: a url

        pointsawarded: int
        required: true

        pointpercontribution: int
        required: true

        constraints: JSONObject
        required: true

        content: JSONArray
        required: true
    }
}
```

### Response General

```json
{
}
```
## Dynamic data integration, storage and access

```json
status: string
  values: done_message, error_message

payload: string
```

### POST Challenge instance

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/createchallengeinstance</td>
<td>Method to post a challenge instance object from a challenge definition. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

#### Call Address

```
http://streambase1.disi.unitn.it:port/createchallengeinstance
```

#### Parameters

```
{  
  email: string required: true  
  password: string required: true  
  object: ChallengeInstance{  
    definitionid: string required: true  
    startdate: string required: true  
    enddate: string required: true  
    location: string required: true  
    constraints: JSONObject required: true  
    project: string required: true  
  }  
}
```

#### Response General

```
{  
  status: string  
  values: done_message,  
}
```
Dynamic data integration, storage and access

```javascript
    error_message
    payload    string

```  

**GET Challenge instances**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET</strong></td>
<td>/getchallengeinstances</td>
<td>Method that returns the available challenges for the specific user. The token is the one that identifies the user with Google Sign-In. Return a list of challenge instances.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/getchallengeinstances

**Parameters**

```javascript

{ 
    token    string
    required: true

```

**Response General**

```javascript


{ 
    status    string
    payload    [ChallengeInstance{
        definitionid    string
        startdate    string
        enddate    string
        location    string
        constraints    JSONObject
        project    string
    }]}}

```  

**GET Challenge result**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET</strong></td>
<td>/getchallengeresult</td>
<td>Allows the user to get the results of her completed challenges. It accepts as parameters the token which is the one that identifies the user with Google Sign-In and the instanceid of the challenge to check. Return a list of challenge instances.</td>
</tr>
</tbody>
</table>

**Call Address**
Dynamic data integration, storage and access

http://streambase1.disi.unitn.it:port/getchallengeresult

Parameters

```json
{
    "token": "string",
    "required": true,
    "instanceid": "string",
    "required": true
}
```

Response General

```json
{
    "status": "string",
    "payload": {"ChallengeResult": {"result": "string", "points": "int"}}
}
```

**GET Publish challenge result**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/publishchallengeresult</td>
<td>Method that allows to publish the result of a specific challenge. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

Call Address

http://streambase1.disi.unitn.it:port/publishchallengeresult

Parameters

```json
{
    "email": "string",
    "required": true,
    "password": "string",
    "required": true,
    "userid": "string",
    "required": true,
    "instanceid": "string",
    "required": true,
    "status": "string",
    "required": true,
    "pointsawarded": "int",
    "required": false
}
```

Response General
Dynamic data integration, storage and access

```json
{
    status string
    values: done_message, error_message

    payload string
}
```

**POST Challenge synchronization**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/challengessynchronization</td>
<td>Method called by the smartphone to confirm the status of the current challenge, either ONGOING, COMPLETED or EXPIRED. The token is the one that identifies the user with Google Sign-In.</td>
</tr>
</tbody>
</table>

**Call Address**

```
http://streambase1.disi.unitn.it:port/challengessynchronization
```

**Parameters**

```
{
    token string required: true

    challenge ChallengeInstance{
        instanceid string required: true

        participationtime string required: false

        status string required: true

        completiontime string required: false
    } required: true
}
```

**Response General**

```
{
    status string
    values: done_message, error_message

    payload string
}
```

**POST Reception confirmation**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D7.3
Method that allows to keep track of the status of each user's contribution on the smartphone, namely, tasks, time diaries or messages. Every time the smartphone receives one of those contributions, this method is called to confirm that it was received. The token is the one that identifies the user with Google Sign-In.

**Call Address**

http://streambase1.disi.unitn.it:port/receptionconfirmation

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Response General
Dynamic data integration, storage and access

```json
{
  status: string,
  values: done_message, error_message
  payload: string
}
```

**POST Contribution answer**

<table>
<thead>
<tr>
<th>Method</th>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/contributionsanswer</td>
<td>Method used to collect the answers of the contributions from the user's smartphone. The token is the one that identifies the user with Google Sign-In.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/contributionsanswer

**Parameters**

```json
{
  token: string required: true
  answer: Answer{
    instanceid: string required: true
    answer: string required: true
    payload: string required: true
    day: long required: true
    instancetimestamp: long required: true
    notificationtimestamp: long required: true
    answertimestamp: long required: true
    delta: long required: true
    answereduration: long required: true
  }
  required: true
}
```
Dynamic data integration, storage and access

<table>
<thead>
<tr>
<th>GET Send task</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Method</strong></td>
</tr>
<tr>
<td>GET</td>
</tr>
</tbody>
</table>

Method user to send a task with the specific content to the user with userid usersalt. Only the administrator can call this method and has to authenticate with his user/password.

**Call Address**

http://streambase1.disi.unitn.it:port/sendtask

**Parameters**

```json
{
   "email": string,
   "password": string,
   "content": string,
   "usersalt": string,
   "t_title": string,
   "t_until": int,
}
```

**Response General**

```json
{
   "status": string,
   "values": done_message,
   "error_message": error_message,
   "payload": string
}
```

---

**GET Send message**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/sendmessage</td>
</tr>
</tbody>
</table>

**Description**

Method to send a message to a specific user with the content of the message. Only the administrator can call this method and has to authenticate with his user/password.

**Call Address**

http://streambase1.disi.unitn.it:port/sendmessage

**Parameters**

```json
{
   "email": string,
   "password": string,
   "content": string,
   "usersalt": string,
   "t_title": string,
   "t_until": int,
}
```

**Response General**

```json
{
   "status": string,
   "values": done_message,
   "error_message": error_message,
   "payload": string
}
```
### GET /sendmessage

Method user to send a message with the specific content to the user with userid usersalt. Only the administrator can call this method and has to authenticate with his user/password.

#### Call Address

http://streambase1.disi.unitn.it:port/sendmessage

#### Parameters

```json
{
  "email" : string
  required: true

  "password" : string
  required: true

  "content" : string
  required: true

  "usersalt" : string
  required: true

  "t_title" : string
  required: true

  "t_until" : int
  required: false
}
```

#### Response General

```json
{
  "status" : string
  values: done_message,
  error_message

  "payload" : string
}
```

### GET Send question

Method user to send a question with the specific content to the user with userid usersalt. Only the administrator can call this method and has to authenticate with his user/password.

#### Call Address

http://streambase1.disi.unitn.it:port/sendquestion

#### Parameters

```json
{
  "email" : string
  required: true

  "password" : string
  required: true

  "content" : string
  required: true

  "usersalt" : string
  required: true

  "t_title" : string
  required: true

  "t_until" : int
  required: false
}
```
Dynamic data integration, storage and access

```
content    string    required: true
usersalt   string    required: true
t_title    string    required: true
t_until    int       required: false
```

Response General

```
{
    status     string
    values: done_message,
              error_message
    payload    string
}
```

**GET Download map**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/downloadmap</td>
<td>Method called from the smartphone, there is no need to identify the user,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>that downloads the map for the challenge (of type validation) specified by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the challengeid.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/downloadmap

**Parameters**

```
{
    challengeid    string
    required: true
}
```

Response General

```
{
    status     string
    values: done_message,
              error_message
    payload    Map{
                    mapid    string
                    points   [{
                                lat: double
                                long: double
                            }]
                }
}
```
Dynamic data integration, storage and access

**GET Update map**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/updatemap</td>
<td>Method called to update the map of a “validation” challenge. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/updatemap

**Parameters**

```json
{
  "email": string,
  "password": string,
  "challengeid": string,
  "map": Map{
    "mapid": string,
    "points": [{
      "lat": double,
      "long": double
    }],
    required: true
  }
}
```

**Response General**

```json
{
  "status": string,
  "values": done_message, error_message
  "payload": string
}
```

**GET Tasks by user**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/gettasksbyuser</td>
<td>Method that returns all the tasks sent to a user. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/gettasksbyuser

**Parameters**

```json
{
  "email": string
}
```
Dynamic data integration, storage and access

```

```{language=java}
password
  string
  required: true

userid
  string
  required: true

```

**Response General**

```

```{language=java}

```

**GET Task by id for user**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/gettaskbyidbyuser</td>
<td>Method that returns a single task sent to a user. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambasel.disi.unitn.it:port/gettaskbyidbyuser

**Parameters**

```

```{language=java}
email
  string
  required: true

password
  string
  required: true

```
Dynamic data integration, storage and access

```
userid: string required: true
taskid: string required: true
```

Response General

```
{
    status: string
    values: done_message,
    error_message

    payload: Task{
        instanceid: string
        required: true
        day: string
        required: true
        question: string
        required: true
        status: string
        required: true
        instancetimestamp: string
        required: true
        title: string
        required: true
    }
}
```

**GET Task answer**

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/gettaskanswer</td>
<td>Method that returns the answer to a specific task (instanceid) sent to the user. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

**Call Address**

http://streambase1.disi.unitn.it:port/gettaskanswer

**Parameters**

```
{
    email: string required: true
    password: string required: true
}
```
Dynamic data integration, storage and access

```
userid string
required: true

instanceid string
required: true

{
  status string
  values: done_message, error_message
  payload Answer{
    instanceid string
    required true

    answer string
    required true

    payload string
    required true

    day string
    required true

   instancetimestamp string
    required true

    notificationtimestamp string
    required true

    answertimestamp string
    required true

    delta long
    required true

    ansverduration long
    required true

  }
}
```

### Response General

GET Challenges answers

<table>
<thead>
<tr>
<th>Method</th>
<th>Api</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/getchallengesanswers</td>
<td>Returns the answers given by all the users to a specific challenge. Only the administrator can call this method and has to authenticate with his user/password.</td>
</tr>
</tbody>
</table>

Call Address
Dynamic data integration, storage and access

http://streambasel.disi.unitn.it:port/getchallengesanswers

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>email</td>
</tr>
<tr>
<td>password</td>
</tr>
<tr>
<td>pilot</td>
</tr>
<tr>
<td>instanceid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response General</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>status</td>
</tr>
<tr>
<td>payload</td>
</tr>
<tr>
<td>instanceid</td>
</tr>
<tr>
<td>answer</td>
</tr>
<tr>
<td>payload</td>
</tr>
<tr>
<td>day</td>
</tr>
<tr>
<td>instancetimestamp</td>
</tr>
<tr>
<td>notificationtimestamp</td>
</tr>
<tr>
<td>answertimestamp</td>
</tr>
<tr>
<td>delta</td>
</tr>
<tr>
<td>answerduration</td>
</tr>
</tbody>
</table>


Dynamic data integration, storage and access

ANNEX 3 - BIG DATA STORAGE TECHNOLOGY STACK

This annex describes more in details the technologies used in the stack of the Big Data Storage of Qrowd-DB.

Apache Spark
Spark was started in the UC Berkeley AMPLab and open-sourced in 2010\(^{24}\). It is designed for efficient iterative computation and starting with early releases has been packaged with example machine learning algorithms. Spark simple programming model can capture streaming, batch, and interactive workloads and enable new applications that combine them, and it has grown to 1,000 contributors and thousands of deployments in recent years \([12]\). The main distinguishing feature of Spark is Resilient Distributed Datasets (RDD), i.e., data-sharing abstraction allowing for the storing of data on memory and its persistence as per requirements \([11]\). More in detail, as the name suggests, they are i) resilient, i.e., fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures, ii) distributed with data residing on multiple nodes in a cluster, and iii) datasets, i.e., a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects. RDDs lead to a massive increase in batch processing job performance.

One of the significant benefits of Spark lies in its generality. Firstly, it makes developing applications easier thanks to a unified API. Secondly, it is more efficient since, unlike other systems, it can run diverse functions over the same data, often in memory. Moreover, Apache Spark provides the following advantages:

1. It is a data parallel general purpose batch processing engine
2. It is implemented in Scala, which is powerful object-oriented language and with several resources running on Java Vector Machines
3. Several Open source contributors, Cloudera, Databricks, IBM, and Intel, publicly committed to support and fund the standardization of Apache Spark as a general purpose engine for big data analysis
4. It can join datasets across multiple disparate data sources.
5. It caches the data in memory, which is especially beneficial for iterative machine learning algorithms.
6. It can to perform stream processing with large input data and deal with only a chunk of data on the fly, which is extremely relevant for use cases requiring real time analysis (as in could be in QROWD)

Container
Containers implement isolation of processes at the operating system level of the host machine, thus, avoiding the overhead due to virtualized hardware and virtual device drivers. A container can be considered a tiny and isolated virtual environment, which includes a set of specific dependencies needed to run a particular application. By containerizing the application platform and its dependencies, differences in OS distributions and underlying infrastructure are abstracted away.

\(^{24}\) http://spark.apache.org/
Dynamic data integration, storage and access
A significant benefit that containers provide is that, rather than run an entire complex application inside a single container, the application can be split into modules (such as the database, the application front end, and so on). This is the so-called microservices approach. Applications built in this way are easier to manage because each module is relatively simple, and changes can be made to modules without having to rebuild the entire application. Because containers are so lightweight, individual modules (or microservices) can be instantiated only when they are needed and are available almost immediately. The business logic of Qrowd-DB is based on the multiple containers and in particular runs thanks to two container technologies: Docker and Kubernetes.

Docker
Docker\(^{25}\) is the most adopted container platform, and it consists of an underlying container engine, together with a functional API that allows quickly building, managing, and removing a containerized application. Because of the small overhead produced, multiple containers can run even in devices with limited computation resources such as Single Board Computer platforms. These lightweight and versatile characteristics have facilitated the use of containers in different contexts ranging from Cloud Computing to the Internet of Things (IoT) scenarios [Linux].

The main features of Docker are the following:

- **Linux Containers (LXC)**: it is a user-space control package for Linux that uses kernel-level namespaces to isolate the container from the host. The user namespace separates the container's and the host's user database, thus ensuring that the container's root user does not have root privileges on the host. The process namespace is responsible for displaying and managing only processes running in the container, not the host. And, the network namespace provides the container with its own network device and virtual IP address.

- **Control Groups (cgroups)**: While namespaces are responsible for isolation between host and container, control groups implement resource accounting and limiting. While allowing Docker to limit the resources being consumed by a container, such as memory, disk space and I/O, cgroups also output lots of metrics about these resources. These metrics allow Docker to monitor the resource consumption of the various processes within the containers and make sure that each gets only its fair share of the available resources.

- **AuFS (Advanced Multi-Layered Unification Filesystem)**: it is the filesystem for containers. AuFS is a layered filesystem that can transparently overlay one or more existing filesystems. When a process needs to modify a file, AuFS creates a copy of that file. AuFS is capable of merging multiple layers into a single representation of a filesystem. This process is called copy-on-write. An added benefit of using AuFS is Docker's ability to version container images.

Kubernetes
Kubernetes\(^{26}\) is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an

\(^{25}\) https://www.docker.com/
\(^{26}\) https://kubernetes.io/
Dynamic data integration, storage and access application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

Kubernetes uses a declarative approach to get the desired state for the applications mentioned by the user. When an application is deployed on a Kubernetes cluster, the Kubernetes master node decides in which underlying host the application has to be deployed. Kubernetes scheduler does the job of application deployment. Kubernetes can replicate containers to improve the availability of applications. When a container fails, Kubernetes recreates it from a predefined image. However, the state of the failed container is not restored. Applications can use external volumes to maintain their state, but it is necessary to protect the volumes against failures. Furthermore, when providing state replication, applications have to handle concurrency when accessing the volume.

Kubernetes offers a wide range of advantages: first, it poses high scalability, easier container management and helps to reduce the delay in communication. Moreover, Kubernetes self-healing, auto restarting, replication and rescheduling mechanisms make it more robust and suitable for container-based applications.

In Qrowd-DB, Kubernetes manages the orchestration, which is very tricky to set up but also very powerful because allows the system to scale up and down automatically. One of the main features of Kubernetes is the load balancer. It allows us to duplicate the containers depending on the load they are facing, in real-time. This adapts very well to the opportunistic nature of the sensor data upload phase. Finally, the twelve nodes of the Apache Spark database are also managed by Kubernetes.
ANNEX 4 - TOP LEVEL eTypes

This annex presents the top-level eTypes for the QROWN data model, namely Entity in Table 9, The Entity eType, which is the parent of Location, Event, Person, and Structure, respectively in Table 10, The Location eType, Table 11, The Event eType, Table 12, The Person eType, and Table 13, The Structure eType. The tables describe the name of the attribute, its description, and its datatype. For these last ones, [] means that the attribute is multivalue, and <> means that the attribute is relational. The inherited attributes are not repeated for the sake of clarity.

Table 9. The Entity eType

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name by which an entity is known</td>
<td>STRING []</td>
</tr>
<tr>
<td>Identifier</td>
<td>a symbol that establishes the identity of the one bearing it</td>
<td>STRING []</td>
</tr>
<tr>
<td>Duration</td>
<td>The duration of existence of an entity</td>
<td>DURATION</td>
</tr>
<tr>
<td>Start</td>
<td>The moment in time an entity started to exist</td>
<td>DATE</td>
</tr>
<tr>
<td>End</td>
<td>The moment in time an entity ceased to exist</td>
<td>DATE</td>
</tr>
<tr>
<td>Description</td>
<td>The description of the entity</td>
<td>[] STRING</td>
</tr>
<tr>
<td>Part of</td>
<td>Defines the connection from the part to the whole, e.g. locations to their administrative division</td>
<td>&lt;ENTITY&gt;</td>
</tr>
<tr>
<td>Class</td>
<td>The class of the entity</td>
<td>CONCEPT</td>
</tr>
</tbody>
</table>

Table 10. The Location eType

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latitude</td>
<td>the standard coordinate for the latitude (in WGS84 deecimal format)</td>
<td>FLOAT</td>
</tr>
<tr>
<td>Longitude</td>
<td>distance of something above a reference point (such as sea level)</td>
<td>FLOAT</td>
</tr>
<tr>
<td>Elevation</td>
<td>the angular distance between a point on any meridian and the prime meridian at Greenwich</td>
<td>FLOAT</td>
</tr>
</tbody>
</table>

Table 11. The Event eType

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participant</td>
<td>someone who takes part in an activity</td>
<td>&lt;PERSON&gt;</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>DataType</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Email</td>
<td>Email of the person</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Birth date</td>
<td>the date on which a person was born</td>
<td>DATE</td>
</tr>
<tr>
<td>Place of birth</td>
<td>The place where the person was born</td>
<td>&lt;LOCATION&gt;</td>
</tr>
<tr>
<td>Date of death</td>
<td>The date on which the person died</td>
<td>DATE</td>
</tr>
<tr>
<td>Place of death</td>
<td>The place where the person died</td>
<td>&lt;LOCATION&gt;</td>
</tr>
</tbody>
</table>

Table 13. The Structure eType

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>the place where the structure is situated</td>
<td>&lt;LOCATION&gt;</td>
</tr>
<tr>
<td>Opening hours</td>
<td>The times when the structure is open for people to use it and services are offered</td>
<td>STRING</td>
</tr>
<tr>
<td>Owner</td>
<td>The owner of the structure</td>
<td>&lt;PERSON&gt;</td>
</tr>
</tbody>
</table>
This annex presents the minimum requirements to run the Qrowd-DB components.

**Big Data Storage**

Before explaining the requirements, let us provide some terminology:

- **Data (D):** disk used to store data for the experiments. The **usable disk space** should be considered to be ¼ of the total hard drive space on the physical machine. ½ of the space is used by the backup configuration (best option will be RAID 10) and the additional ½ is used by Cassandra to compact the data (See Cassandra Compaction Strategies for more information). This means that if we want to be able to store 1TB of data we need a physical disk of 4TB.

- **Power resources (PR):** RAM and CPU used by the Cassandra process(es) to consume the data, where consuming means **writing** to disk while collecting or **reading** from disk to do analytics and reply to queries.

- **Cassandra node (CN):** a Docker container composed by 1 unit of D and 1 unit of PR, where D and PR are logically decoupled. The minimum requirements for one CN instance are:
  - **PR:**
    - at least 16GB of RAM (32GB best option)
    - 4 CPU cores (8 best option)
  - **D:**
    - One node can easily handle 1TB of data, we cannot overcome this limit. Considering the requirements on D as above, each CN needs up to 4TB of data.

- **Physical machine (PM):** a physical server that can host multiple CNs

- **Experiment (EXP):** one experiment needs multiple CNs, minimum 2, maximum size depends on the size of the data to be collected. This CN must be spread across multiple PM in order to guarantee fault tolerance and load balancing (ideally at the beginning we can also have one single PM and add the others later). An experiment can have 2 states: **running** and **stopped**. A **running** experiment means that both D and PR for every CN are allocated because it is collecting or serving the data. A **stopped** experiment, on the other hand, means that the D are allocated while the PR is not. In the stopped state, data cannot be written or read and is only saved for the future.

*Figure 5. 2 PM, with 4 CN each, running...* shows how the different elements work together in a configuration with 2 PM, with 8 total CN and 2 experiments EXP1 and EXP2 running in parallel.
Dynamic data integration, storage and access

When the experiment is in running state, both D and PR must be allocated. Once the experiment is finished, the PR can be de-allocated and those resources can be reused for another CN. Every time the data need to be accessed, PR can be allocated again. D cannot be deallocated otherwise the data is lost forever. One important consideration to be done is that D is generally cheaper than PR and this motivates the choices we made. This concept can be seen in Figure 6. 2 PM, with 6 total CN. 2... With respect to the previous view with 2 running experiments, EXP1 and EXP2, with respectively 5 and 3 CNs, EXP1 has been put in stop state. This means that its data is still saved on disk (D) but its PR are deallocated, meaning that the data cannot we accessed till new resources are allocated back to it. Instead, the 3 of the 5 now available PRs are allocated to a new experiment, EXP3 in this case only on PM2. This means that there are two additional PR on PM1 that are free for an additional new experiment.

Figure 5. 2 PM, with 4 CN each, running 2 EXPs.

Figure 6. 2 PM, with 6 total CN, 2 on PM1 and 4 on PM2 and 2 EXPs running, EXP2 and EXP3.
Following this design pattern we can run multiple experiments in parallel. This means that we can collect data for EXP3, while doing machine learning on EXP2 and while publicly exposing data from EXP1 for research purposes (or for hackatons).

**Linked Data Storage**

Table 14. Minimum and recommended system settings presents the minimum and recommended system setting in the case of the Linked Data Storage.

<table>
<thead>
<tr>
<th>Minimum</th>
<th>Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Xeon family 10+ cores 2.4–3.2 GHz 32MB Cache</td>
<td>CPU: 2x E5 Xeon family 10+ cores 2.4–3.2 GHz 32MB Cache</td>
</tr>
<tr>
<td>RAM: 48GB DDR4 RDIMM-2133 ECC</td>
<td>RAM: 64GB DDR4 RDIMM-2400 ECC</td>
</tr>
<tr>
<td>Disk: RAID 0 SSD Write Intense 500GB</td>
<td>Disk: RAID 0 SSD Write Intense 500GB</td>
</tr>
<tr>
<td>Network: 1Gbit adapter</td>
<td>Network: 1Gbit adapter</td>
</tr>
<tr>
<td>Integrated RAID controller</td>
<td>Dedicated Raid Controller</td>
</tr>
</tbody>
</table>

In case of a backup network could be useful to add a second network card. If there is a physical Windows server a higher speed network interface could be useful in order to connect the two machines.