URBANITE

Supporting the decision-making in urban transformation with the use of disruptive technologies

Deliverable D5.4

URBANITE Detailed architecture-v1

Editor(s):	Juncal Alonso/Maria José López
Responsible Partner:	Tecnalia
Status-Version:	Final
Date:	31.03.2021
Distribution level (CO, PU):	PU

Project Number:	GA 870338
Project Title:	URBANITE

Title of Deliverable:	URBANITE Detailed architecture-v1
Due Date of Delivery to the EC:	31/03/2021

Workpackage responsible for the Deliverable:	WP5 - URBANITE ecosystem integration and DevOps
Editor(s):	Tecnalia
Contributor(s):	Juncal Alonso, Maria José López, Sonia Bilbao, Gonzalo Lázaro, Iñaki Olabarrieta (Tecnalia), Fritz Meiners (FhG), Maj Smerkol (JSI), Giuseppe Ciulla (ENG)
Reviewer(s):	Denis Costa (WAAG)
Approved by:	All Partners
Recommended/mandatory readers:	WP2, WP3, WP4, WP5, WP6

Abstract:	This document contains the detailed design of URBANITE: its components, modules, and interfaces. Two releases of the document are planned. In the second one the comments received from the user cases implementation. This deliverable is the result of Task 5.2.
Keyword List:	Architecture, Development environment, integration, testing, requirements.
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

Document Description

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
V1	14/01/2021	First draft version with TOC	Tecnalia
V1.2	01/02/2021	Sections reorganized	Tecnalia
V1.3	28/02/2021	Section 3.1.6	Tecnalia
V1.4	03/03/2021	Contribution to section 3.1	Fraunhofer
V1.5	04/03/2021	Contribution to section 3.1.5	Tecnalia
V1.6	04/03/2021	Transformation section	Fraunhofer
V1.7	04/03/2021	Reorganization of the previous versions	Tecnalia
V1.8	05/03/2021	Section 3.2.6 and 3.2.7	Tecnalia
V1.8_jsi	08/03/2021	Section 3.2.2	JSI
V1.5_ENG	08/03/2021	Data Catalogue; Controller; Virtual SoPoLab; URBANITE UI; Identity/Authorization Management	ENG
V1.9	8/03/2021	Document Reorganized	Tecnalia
V1.9_jsi	12/03/2021	Contributions from JSI.	JSI
V1.10	15/03/2021	Version ready for internal review	Tecnalia
V1.11	26/03/2021	Internal review made	WAAG
V1.12	26/03/2021	First version of the document	Tecnalia
V2.0	29/03/2021	Final version	Tecnalia

Table of Contents

List of Figures

List of Tables

Terms and abbreviations

API	Application programming Interfaces
CKAN	Comprehensive Knowledge Archive Network
DAG	Direct Acyclic Graph
DCAT	Data CATalogue
DCAT-AP	DCAT Application Profile
EC	European Commission
GPS	Global Positioning System
JSON	JavaScript Object Notation
NGSI	*Next Generation* Service Interfaces
RDF	Resource Description Framework
REST	REpresentational State Transfer
SAML	Security Assertion Markup Language
UI	User Interface
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations

Executive Summary

The present document contains the detailed description of the URBANITE integrated architecture at month 12, starting with a theoretical vision of the URBANITE system that will cover all the functional and non-functional initial requirements set by the technical work-packages considering the social perspective and the input of the use cases detailed in deliverable D5.1 [1]. The definition of the interactions among components is shown through the specification of the interfaces, considering the dataflows envisioned for meeting the needs of the different stakeholders.

The components that form the architecture are developed within the technical work packages WP2, WP3, WP4 and WP5, supporting the functionalities provided by the Social Policy Lab, the Data Management Platform and the Algorithms and simulation techniques, including the URBANITE UI.

The sections of this document are organized to follow the outcomes of the different work packages and present an overview of the URBANITE integrated conceptual architecture. Starting with a general overview of the URBANITE architecture, covering the most important layers of the ecosystem as the “Data acquisition, aggregation and storage” for the management of the data within URBANITE, the “Data analysis” where the algorithms related to the analysis of the big data will be performed and the“Decision support” including the mechanisms for taking decisions based on the analysis made.

The general URBANITE UI involves all the layers mentioned above, allowing a fluid use of the platform, and managing the identities and authorizations, making this ecosystem a safe and secure context for the different stakeholders.

An important effort of integration will be required to ensure smooth interoperability of the different components covering the entire data processing chain, implementing the orchestration of all the components and services as an URBANITE controller module.

The architecture presented in this document is a lively design, subject to change following the research activities of the project and the analysis of the use case evolution. Moreover, other tools will be studied in case they fit into the architecture providing the functionality required and making easier the development and integration of the URBANITE ecosystem.

There will be another version of this document in M24 reflecting the status of the URBANITE Ecosystem at this point in the project.

1 Introduction

1.1 About this deliverable

This deliverable is the first release of the detailed design of the URBANITE architecture, being the second one the final schema of it. The description contains the components and interfaces that are part of the URBANITE ecosystem.

The document reflects the work done in the T5.2 and T5.4 tasks jointly with WP2, WP3 and WP4, and describes the main components of URBANITE architecture and the different Key Results, including the interactions among them. The URBANITE User Interface is an important part of the architecture since it is the contact point between the users and the ecosystem, as well as the final visualization of all the analysis that the platform allows in order to support the decisions to be made by those users. This UI will be developed using responsive web technologies that will bring a good user experience.

1.2 Document structure

The structure of the document is organized to offer a coherent reading and a clear vision of all the components that are part of the general architecture. There are some main sections grouping the components following the type of functions they will provide. Inside these sections, the individual components will be presented and described with a similar list of subsections devoted to explaining the main functionality, a structural overview regarding the internal and external interactions with other components or its own subcomponent, a dynamic overview of the general process or workflow, and design implications based on the tool choice

The mentioned structure is as follow:

- Section 2: An overview of the general schema of the integrated architecture, as well as the identification of the main actors and their roles.
- Section 3: The main section grouping the components following the type of functions they will provide. Inside these sections, the individual components will be presented and described with a similar list of subsections devoted to explaining the main functionality, a structural overview regarding the internal and external interactions with other components or its own subcomponent, a dynamic overview of the general process or workflow, and design implications based on the tool choice.
    - 3.1: A description of the components related to the acquisition, aggregation and storage of the data, corresponding to the process in which the data are harvested and finally stored in a formalized way. These components are Data Harvesting, Preparation and Transformation, Data Anonymization, Data Curation, Data Fusion/Aggregation, Data Storage & Retrieval and Data Catalogue.
    - 3.2: The Data analysis components regarding to the algorithms to perform the analysis and processes that conform the “Engine” of the Ecosystem. The Data projection, data clustering, self-organizing map, correlation discovery, prediction, and the analytical framework are the components that provide the intelligence of the platform in order to offer the analysis and processes needed by the management of the data. The controller component is a utility for orchestrating the components of the URBANITE Ecosystem.
    - 3.3: About the components for decision support that present the analysis results and guides the end-user towards the policy decisions. These components are Traffic simulation, Policy Simulation and Validation, Recommendation Engine, and Advanced visualization.

    - 3.4: SoPoLab. Description of this Digital virtual Space, for sharing experiences of different policy domains.
    - 3.5: The integrated URBANITE UI, for describing the component that provides access to the URBANITE technical tools offered by the rest of the components.
- Section 4: Conclusions arose from this version of the architecture and future work.
- Section 5: References made in the content of the document.

2 Overview of the URBANITE integrated conceptual architecture

2.1 URBANITE ecosystem generic architecture

The current vision of the URBANITE architecture is depicted in Figure 1.

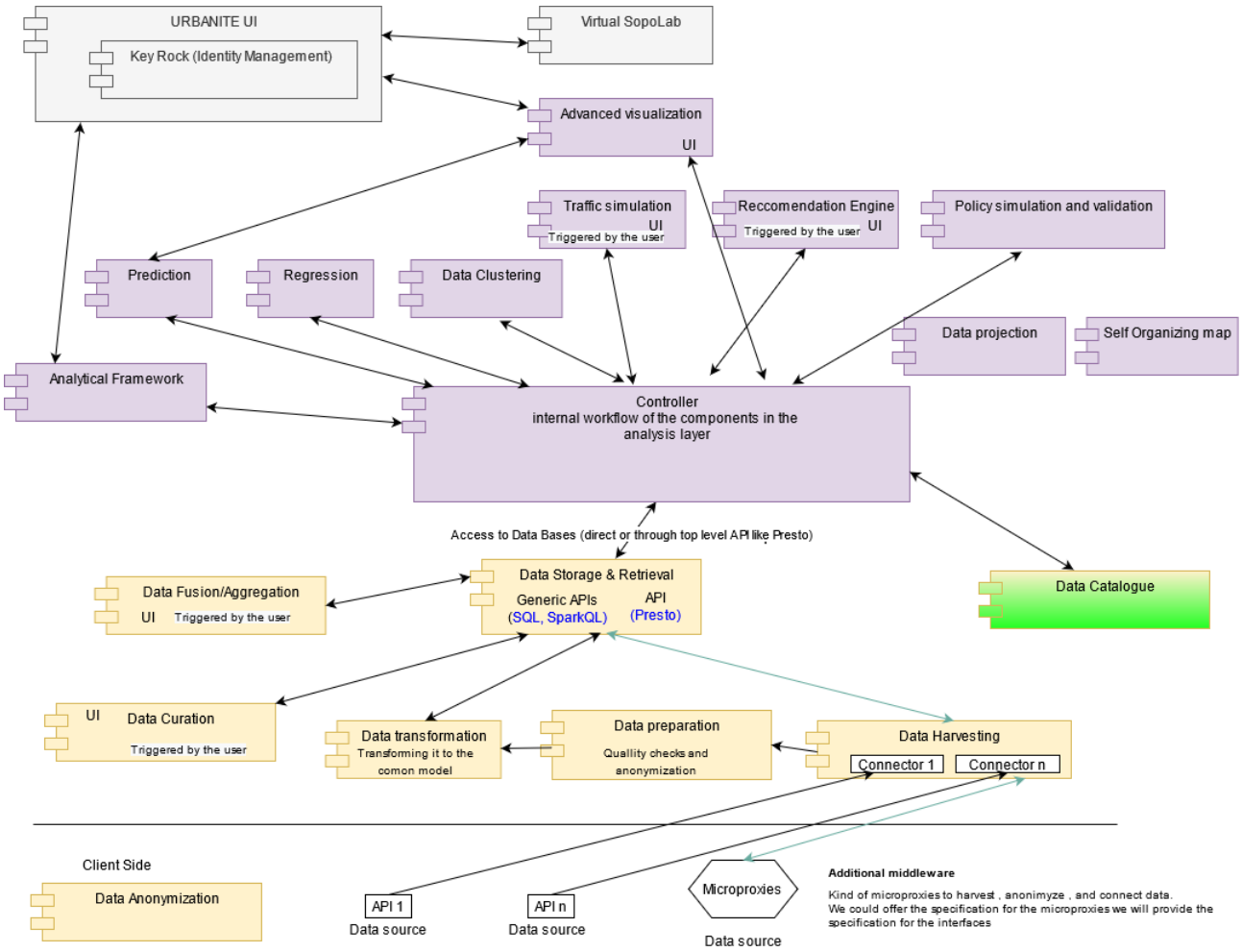

Figure 1. First version of URBANITE architecture

This schema sketches the structural view of the URBANITE architecture in the present status. More detailed views of the components that are part of it will be presented in the following sections.

The structure presented is the result of the analysis made at this point from the technical work packages, and it is liable to evolving regarding the outcomes of them, alongside the requirements that the use cases establish.

2.2 Intended users and their roles

2.2.1 URBANITE actors

The D2.2 deliverable [1] depicts the main actors of the new urban mobility scenario as the citizens, service providers, public servants and policy makers.

The work made in order to define the main users of the URBANITE platform has been developed with the participation of the pilot cities (Amsterdam, Bilbao, Helsinki and Messina), with the objective of identifying the most relevant agents involved in the use cases of each city. The common aspect of those actors has been the interest in the use of disruptive technologies in the area of mobility and urban transformation.

The stakeholders there are classified into several groups as follow:

- Public servants: City's administrators and local authorities
- Policy makers: Civil servants as municipalities.
- Service providers: Business companies, urban mobility platforms, Civil society/platform and users/neighbour association

Regarding the use of the platform from the perspective of a group that can modify the configurations and the flow of the different tasks to be performed with respects to the management of the data, the role of Admin shall be assigned to the technical areas of the cities.

The Admin role will be able to modify the configurations and the flow of the different tasks to be performed concerning data management and modify the configurations and the flow of the different tasks to be performed. This role shall be assigned to the technical areas of the cities.

The municipalities, seen as the servants to offer the best services to the citizens, play the role of administrators and those who harvest, curate and fuse data for visualizing and simulating different behaviours, so they can use those outcomes as a support for making policies and decisions in order to improve the citizens' life.

The data providers are the companies managing the services in the areas related to urban mobility, traffic analysis, health care, car and bike-sharing groups, etc.

More information about the social implication and participation as direct contributions of public servants, citizens and other stakeholders in a co-creation process, can be found in that document.

3 URBANITE ecosystem architecture detailed design

3.1 URBANITE components for data acquisition, aggregation and storage

3.1.1 Data Harvesting, Preparation and Transformation

3.1.1.1 Main Functionality

This section explains the concept of harvesting via a so-called pipe. In a broader sense, the entire process of fetching, preparing, transforming, and exporting data may be referred to as harvesting, i.e. providing a way to make heterogeneous data available in defined format and means of access. This process is depicted in Figure 2. Data anonymization and curation are, as shown in the architecture diagram, not part of this pipeline, but separate steps.

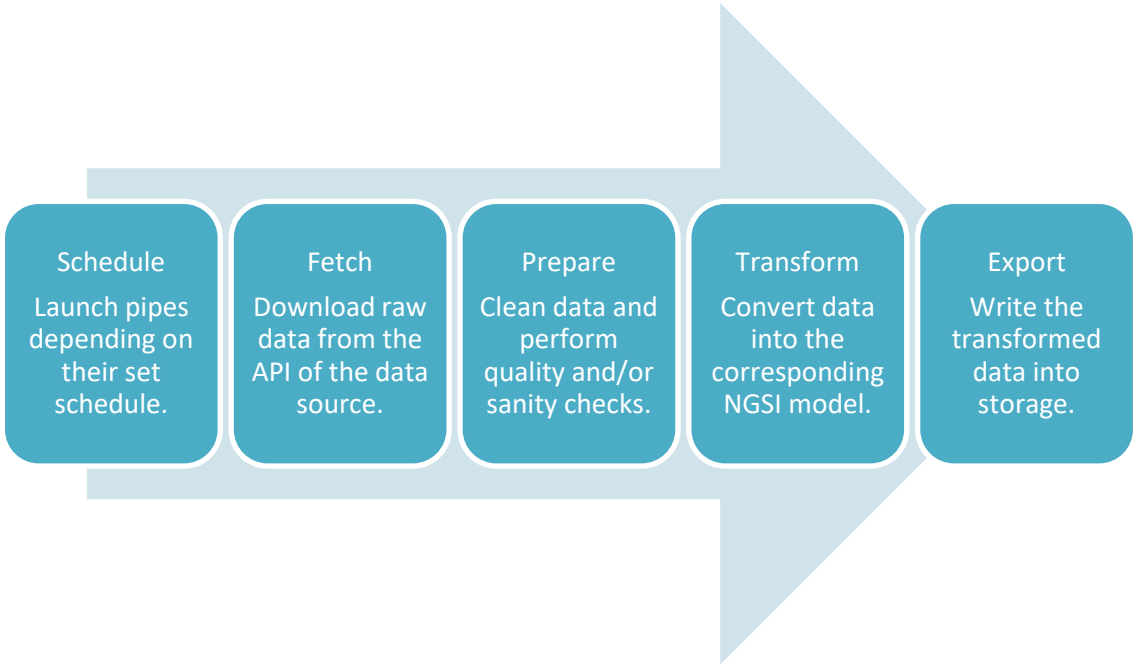

Figure 2: Harvesting process

As can be seen, the architecture follows the form of a pipeline. This means that data is passed through the pipeline, and each component is agnostic of the other steps. This leads to loose coupling and improves flexibility. For example, in a harvesting process for a data source that already serves NGSI-LD (the common format for all data stored in URBANITE), the transformation step could simply be omitted, with the preparation component writing directly into the exporting component without having to rewrite APIs.

In detail, the harvesting process would typically consist of the following steps:

1. The scheduler triggers a pipeline
2. The harvester retrieves the data from the source's API and forwards it into the preparation component.
3. After cleaning and validating, the preparation component forwards the data to the transformation component.
4. The data is transformed to the applicable NGSI data model and forwarded to the exporter.

5. Finally, the exporter writes the harmonized data into the data storage component.

The **data harvesting component** is responsible for fetching data from a given API. It does not alter the data. It can be considered the entry point of the data into the pipeline. As such, a dedicated component is required for each type of data source. The harvesting component may implement pagination mechanisms for handling data in chunks. However, this does not impact the pipeline – each chunk is handled individually and does not depend on other chunks.

The **data preparation component** is responsible for performing initial cleaning and sanitation of the data provided by the harvesting component. This ensures a fixed level of data quality and integrity, which is required by the transformation component to operate flawlessly.

Data transformation is a key step in the harvesting pipeline. It cannot be expected that the municipalities provide their data in one of the common data models developed by FIWARE used in the URBANITE context. As such, the transformation of the heterogeneous data sources into common models is vital for frictionless processing of the data henceforth. For a flexible approach, the actual transformation instructions are loaded via scripts, either Javascript for JSON based payloads or XSLT for XML based payloads. More engines can be added as pipeline modules at a later point in time.

3.1.1.2 Structural overview

For a component to be compatible with the pipe concept they need to cohere to a certain specification, which describes the orchestration of all components that make up a pipeline, as well as storing their individual configuration. The specification is passed along the various components in their declared order. Each component writes its output into the corresponding section of the specification and forwards the data structure to the next service in line.

An example of such a pipe specification is shown in Figure 3.

```
Header:
  name: URBANITE harvester
  version: '2.0.0'
  transport: payload
body:
  segments:
    - header:
        name: fetch-data
        segmentNumber: 1
      body:
        config:
          address: https://dataProvider.eu/api
          inputFormat: application/rdf+xml
    - header:
        name: transform-data
        segmentNumber: 2
      body:
        config:
          scriptType: repository
          repository:
            uri: https://gitlab.com/pipes.git
            script: my-transformation-script.js
            username: sampleUser
            token: sampleToken
    - header:
        name: export-data
        segmentNumber: 3
      body: {}
```

Figure 3: Example of a pipe specification

For this to work, all components must implement a common endpoint, which is shown below. Due to the limited scope of each component only one endpoint, aside from possible metrics/monitoring endpoints, needs to be exposed:

POST	/pipe
Accepts a pipe specification in JSON format.	
Payload	
JSON/YAML Object	Pipe specification
Responses	
202	Accepted
400	Bad request

3.1.1.3 Dynamic Overview (Scheduling)

The initial trigger for each harvesting run is issued by the scheduling component. It keeps track of all pipeline specifications and their schedules. Whenever a timer triggers, the corresponding pipe specification is sent to the first service specified.

The scheduling components, not part of the pipeline itself, feature different APIs than the pipe modules.

GET	/triggers
Get a list of pipe ids and scheduled triggers.	
Payload	
JSON Object	Map of pipe IDs and triggers
Responses	
200	Ok

PUT	/triggers
Bulk update of all triggers.	
Payload	
JSON Object	• IntervalTrigger • CronTrigger • SpecificTrigger
Responses	
200	Ok

GET	/triggers/{pipeId}
Returns all triggers for a pipe with pipeId.	
Payload	
JSON Object	• IntervalTrigger • CronTrigger • SpecificTrigger
Responses	
200	Ok
404	Pipe not found.

PUT	/triggers/{pipeId}
Create or update triggers for pipe with pipeId.	
Payload	
JSON Object	• ImmediateTrigger • IntervalTrigger • CronTrigger • SpecificTrigger
Responses	
200	Triggers created successfully.
201	Triggers updated successfully.
404	Pipe not found.

DELETE	/triggers/{pipeId}
Delete previously created triggers.	
Responses	
200	Triggers deleted successfully.
404	Pipe not found.

GET	/triggers/{pipeId}/{triggerId}/{status}
Set status of a trigger. To be changed to POST in future versions.	
Responses	
200	Trigger status successfully set.

404	Pipe or trigger not found.
409	Status already set or unknown.

3.1.1.4 Design implications based on the tool choice

None

3.1.2 Data Anonymization

3.1.2.1 Main functionality

The anonymization component is a restful microservice capable of transforming large datasets in conformity with data protection requirements for further data analysis. In order to achieve a certain degree of anonymization the user can mark specific attributes that are likely to reveal information about a person or a smaller group. Those identifiers are then transformed in a way that ensures a sufficient level of anonymization. Currently supported anonymization methods are suppression and generalization, which either delete attribute entries in a row or generalise them according to a fixed hierarchy, such as street -> zip code -> city.

3.1.2.2 Structural overview

All endpoints exposed by the provided APIs serve the configuration management. Since this component is a work in progress, the APIs are subject to change.

POST **/**	
Generate JSON Configuration to anonymize personal data.	
Payload	
JSON Object	Anonymization configuration.
Responses	
200	Ok
400	Bad request

GET **/configuration**	
Get configuration as JSON from a database.	
Payload	
JSON Object	Anonymization configuration.
Responses	
200	Ok
400	Bad request

POST	/configuration
Store configuration as JSON in Database.	
Payload	
JSON Object	Anonymization configuration.
Responses	
200	Ok
400	Bad request

PUT	/configuration
Update configuration as JSON in Database.	
Payload	
JSON Object	Anonymization configuration.
Responses	
200	Ok
400	Bad request

DELETE	/configuration
Delete configuration.	
Responses	
200	Configuration deleted successfully.

3.1.2.3 Dynamic overview

In accordance with the architectural diagram, the anonymization is not part of the harvesting pipeline. Instead, it will run as a separate service, if possible, on the client side. This means that the anonymization ideally takes place before any data is exposed via API and subsequently harvested.

3.1.2.4 Design implications based on the tool choice

None

3.1.3 Data Curation

3.1.3.1 Main functionality

Often the value of data can be raised by curation, i.e. enrichment and annotation. As such, the curation component plays a vital role in getting the most out of the data provided by the municipalities and deriving the most precise recommendations.

3.1.3.2 Structural overview

Since this component is still in development, the APIs are not final yet. The granularity with which the curation can be fine-tuned is to be determined.

3.1.3.3 Dynamic overview

Like anonymization, data curation is also not part of the harvesting pipeline. Instead, the curation process will be triggered by the user.

3.1.3.4 Design implications based on the tool choice

None

3.1.4 Data Fusion/Aggregation

3.1.4.1 Main functionality

Data aggregation is the process of gathering data and presenting it in a summarized format. It can be used to hide personal information, or to provide information in a synthetic form.

Data fusion is the process of integrating multiple data sources to produce more consistent, accurate, helpful information and sophisticated models than that provided by any individual data source. This means that the result of the data fusion process, once the N different datasets are integrated should be worth more than the sum of each single dataset's result.

3.1.4.2 Structural overview

Since this component is still in development, the APIs are not final yet.

3.1.4.3 Dynamic overview

The processes of data fusion and data aggregation will use the data that is already stored in the data storage & retrieval repositories. In the case of aggregation, they will be probably batch processes that are executed via a scheduler.

3.1.4.4 Design implications based on the tool choice

None

3.1.5 Data Storage & retrieval

3.1.5.1 Main functionality

The Data Storage & Retrieval component provides the means to store and retrieve datasets metadata and related data. Hence, this component will have repositories to store both DCAT-AP compliant metadata and transformed data, as depicted in Figure 4.

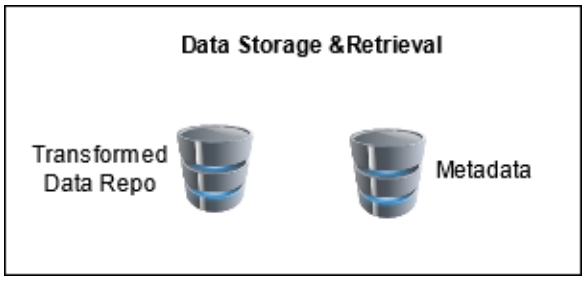

Figure 4. Data Storage & Retrieval repositories

Figure 5 represents the main concepts of DCAT which are catalogue, dataset and distribution. A ***catalogue*** represents a datasets collection. A ***dataset*** represents a data collection published as part of a catalogue. A ***distribution*** represents a specific way to access a specific dataset (such as a file to download or an API). Figure 6 provides a simplified representation of the DCAT-AP Model. For example, the following dataset taken from Euskadi Open Data Portal, https://opendata.euskadi.eus/catalogo/-/estadistica/ofertas-de-empleo-registradas-en-lanbide-durante-el-2021/ is represented following DCAT-AP metadata model in RDF format at this link:

https://opendata.euskadi.eus/contenidos/estadistica/ofertas_empleo_2021/es_def/r01DCATDataset.rdf

Dataset metadata include, for example, the title, description, publisher, a temporal period, etc.

The concept dcat:Distribution provides metadata about the distribution, e.g. the property dcat:accessURL provides the information about how to access a specific dataset. Other important metadata related to the distribution is, for instance, the license, a description, the format of the data (e.g. CSV, JSON), etc.

Figure 5. Main concepts of DCAT

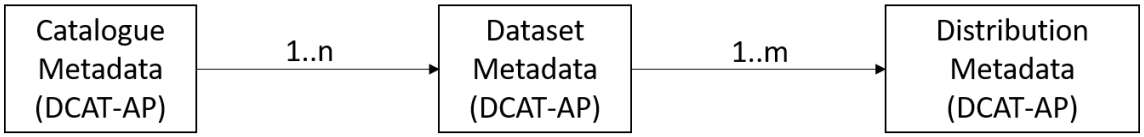

Figure 6. Simplified DCAT-AP Model

3.1.5.2 Structural overview

As mentioned before, this component provides APIs for the storage and retrieval of transformed data and datasets metadata.

In the case of the storage of transformed data, the API will offer one method for all data models. The data model will be specified as a parameter. Before storing the data, this component validates the fields according to the NGSI related data model and, if valid, then inserts the data into a dedicated MongoDB collection <DataModelName>_GeographicZone, e.g. TrafficFlowObserved_Helsinki.

POST	**insertTData**
Inserts the transformed data into the database, in the collection of the model <Model> and the zone specified by the "zone" parameter	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: "Helsinki")
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses	
200	Ok
400	Bad request

* mandatory

GET	**getTData**
Retrieves transformed data from the repository according to the given data model	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: "Helsinki")
filters (JSON)	Different filters (NGSI data model fields) to apply.
count (positive integer)	Number of documents to retrieve
Responses (JSON with the following fields)	
model (String)	Name of the Data Model (i.e.: "TrafficFlowObserved"). The same as input.
Data (JSON Array)	Requested data, in NGSI Data Model

GET	**getTDataRange**
Retrieves transformed data from the repository according to the given data model and for a given period in time.	
Parameters	

model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
start* (DateTime in ISO8601 UTC format)	Date and time from which records are obtained. Mandatory if “end” is not present.
end* (DateTime in ISO8601 UTC format)	Date and time until which records are obtained. Mandatory if “start” is not present.
Responses (JSON with the following fields)	
model (String)	Name of the Data Model (i.e: “TrafficFlowObserved”). The same as input.
Data (JSON Array)	Requested data, in NGSI Data Model

GET **getSupportedDataModels**	
Retrieves the supported data models for transformed data	
Parameters	
None	
Responses	
Data (JSON Array)	List of data models

In the case of storage and retrieval of datasets metadata, the API offers the following methods.

PUT **dataset**	
Inserts new dataset metadata or updates it if it already exists for that dct:identifier.	
Parameters	
id* (String)	Dataset identifier (dct:identifier)
metadata* (JSON-LD)	Updated metadata in JSON-LD format
Responses	
200	Ok
400	Bad request

DELETE **dataset**
Deletes a data set metadata
Parameters

id* (String)	Dataset identifier (dct:identifier)
Responses	
200	Ok
400	Bad request

GET	**getDataset**
Retrieves DCAT-AP dataset metadata.	
Parameters	
id* (String)	Dataset identifier (dct:identifier)
Responses	
dataset	metadata in JSON-LD format

GET	**getCatalogueDatasets**
Retrieves all DCAT-AP datasets in the catalogue.	
Parameters	
None	
Responses	
dataset	metadata in JSON-LD format

GET	**searchDatasets**
Searches among the metadata of the existing dataset	
Parameters	
search params*	Search parameters
Responses	
Dataset list	List of datasets whose metadata fulfils the search parameters criteria

3.1.5.3 Dynamic overview

The main components involved in the process of storage of datasets metadata and related data are the Data Harvester, the Data Transformation, the Data Catalogue and the Data Storage & Retrieval components, as shown in Figure 7.

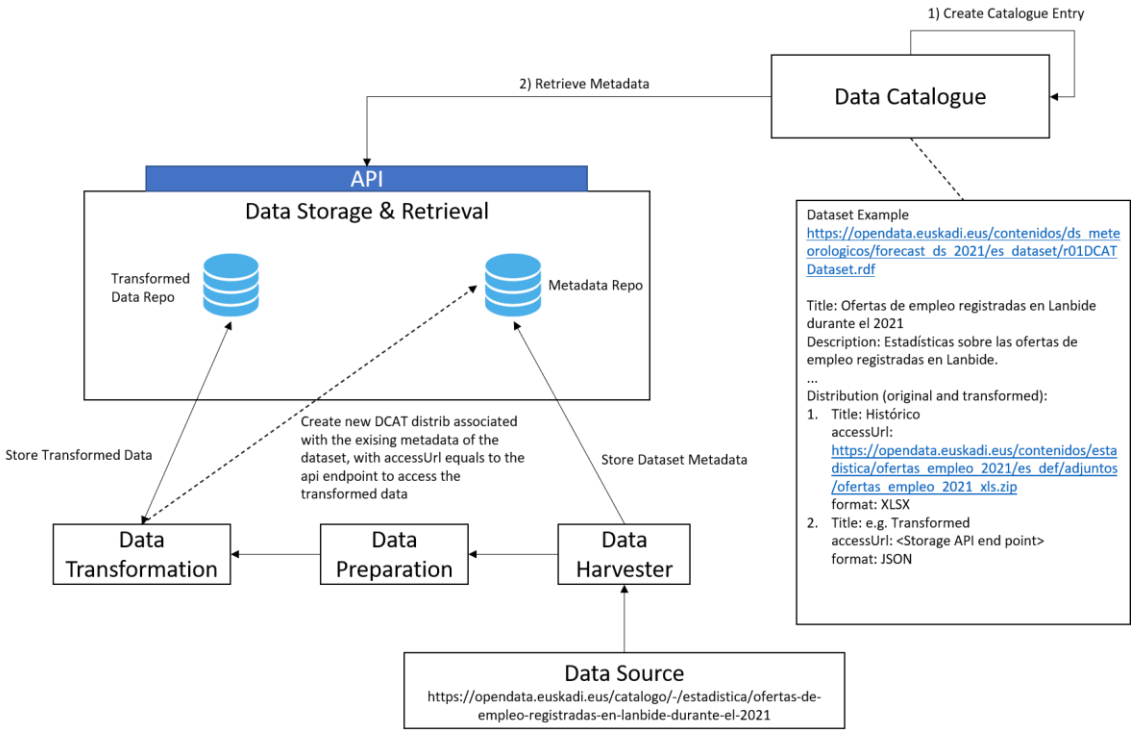

Figure 7. Process of storage of datasets metadata and related data

3.1.5.4 Design implications based on the tool choice

None.

3.1.6 Data Catalogue

3.1.6.1 Main functionality

The Data Catalogue will offer the functionalities to discover and access the datasets collected and managed by the components of URBANITE Ecosystem for data acquisition, aggregation and storage, such as the *Data Harvesting* and *Data Storage & Retrieval* components.

Figure 8 depicts the main actors of the Data Catalogue and the main functionalities it provides.

Apart from the possibility to search over the datasets directly collected by the URBANITE Ecosystem, the Data Catalogue will offer the possibility to search useful data across external "federated catalogues" (such as Open Data Portal) to increase the chance to find useful data.

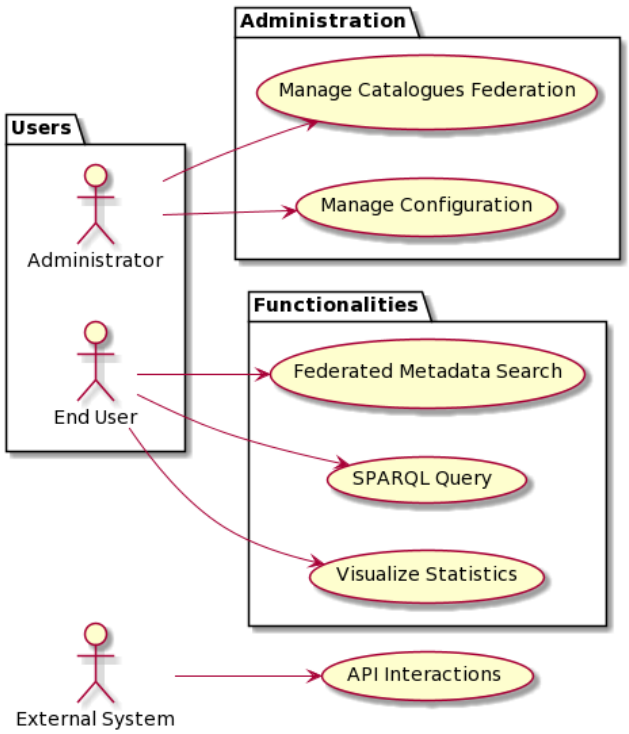

Figure 8: Data Catalogue – Data Catalogue Use Case diagram

The Administrator is in charge of managing the federation of the catalogues, where a catalogue represents a data source, for instance, the Data Storage & Retrieval component. He/she can add new catalogues, delete or edit the existing ones. Moreover, the administrator can manage the platform configurations.

The End user is then able to perform a federated metadata search among the harmonized DCAT-AP datasets provided by the federated catalogues. Moreover, the end user can perform SPARQL queries over the federated RDFs provided by the federated catalogues, or he/she can access to statistics about the federated catalogues.

The Data Catalogue will expose APIs to access its functionalities; thus an external system will be able to interact with the platform using such APIs.

The candidate tool to realize the Data Catalogue is Idra[1]. Idra is a web application able to federate existing Open Data Management Systems (ODMS) based on different technologies providing a unique access point to search and discover open datasets coming from heterogeneous sources. Idra uniforms the representation of collected open datasets, thanks to the adoption of international standards (DCAT-AP) and provides a set of RESTful APIs to be used by third-party applications.

3.1.6.2 Structural overview

In the context of the URBANITE's architecture, the Data Catalogue component provides access to the metadata of the different data sources and their datasets. The component harmonizes the metadata following DCAT-AP[2] standard and provides and unique point of access to search among the available metadata coming from different and heterogeneous data sources. The

[1] https://idra.readthedocs.io/en/latest/

[2] https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/dcat-application-profile-data-portals-europe/release/11

following Figure 9 depicts the interaction among the Data Catalogue and the other URBANITE's components.

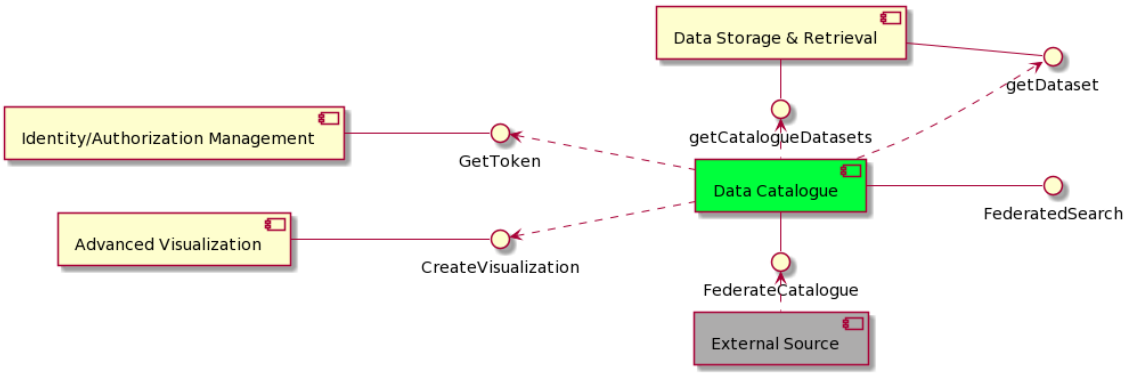

Figure 9: Data Catalogue - Component diagram

The Data Catalogue interacts with 1) Identity/Authorization Management component to allow administrators to access their specific functionalities retrieving the access token that will be further provided to the APIs, 2) Advanced Visualization to build visualization taking advantage of the DCAT-AP distributions it manages and 3) Data Storage & Retrieval to retrieve DCAT-AP datasets and distribution metadata. Finally, the Data Catalogue component is able to federate external sources such as Open Data portals or other sources providing DCAT-AP metadata.

Among the APIs provided by the component, the following are used by the administrator to manage a catalogue and by the user to perform a federated search:

Table 1: Data Catalogue Administration - Retrieve Federated Catalogues

GET	**/administration/catalogues**	
Retrieve the list of the federated catalogues. The administrator accesses additional catalogue's information		
Headers		
Authorization	Bearer <token>	the administrator token
Responses		
200	The catalogues list	
400	Bad request	
401	Unauthorized	
500	Internal Server Error	

Table 2: Data Catalogue Administration - Create Catalogue

POST	**/administration/catalogues**[3]

[3] https://idraopendata.docs.apiary.io/#reference/administration-api/catalogues-resources/post

Create a new catalogue into the Data Catalogue component.		
Headers		
Authorization	Bearer <token>	the administrator token
Body		
Catalogue	An object representing the catalogue to be added	
Responses		
200	The catalogue was successfully created	
400	Bad request	
401	Unauthorized	
500	Internal Server Error	

Table 3: Data Catalogue Administration - Update Catalogue

PUT **/administration/catalogues/{id}**		
Update the catalogue identified by the id.		
Headers		
Authorization	Bearer <token>	the administrator token
Parameters		
Id	Id of the catalogues, path parameter	
Body		
Catalogue	An object representing the catalogue to be updated	
Responses		
200	The catalogue was successfully updated	
400	Bad request	
401	Unauthorized	
500	Internal Server Error	

Table 4: Data Catalogue Administration - Retrieve Catalogue

GET **/administration/catalogues/{id}**
Retrieve the catalogue identified by the id. The administrator accesses additional catalogue's information

Headers		
Authorization	Bearer <token>	the administrator token
Parameters		
Id	Id of the catalogues, path parameter	
Responses		
200	The catalogue object	
400	Bad request	
401	Unauthorized	
500	Internal Server Error	

Table 5: Data Catalogue Administration - Delete Catalogue

DELETE **/administration/catalogues/{id}**		
Delete the catalogue identified by the id.		
Headers		
Authorization	Bearer <token>	the administrator token
Parameters		
Id	Id of the catalogues, path parameter	
Responses		
200	The catalogue was successfully deleted	
400	Bad request	
401	Unauthorized	
500	Internal Server Error	

Table 6: Data Catalogue User - Federated Search

POST **/search**[4]	
Searches among the metadata of the dataset of the federated catalogues	
Parameters	
filters	An array of the filters to be used. A filter is defined with a field and a value (e.g. field: 'title' and value: 'test')

[4] https://idraopendata.docs.apiary.io/#reference/end-user-api/metadata-search/post

rows	The number of results to be returned per request.
start	The offset in result list from which to start.
nodes	The list of the identifier of the catalogues.
Responses	
Search Response	An object with the following fields: • count: the number of results found • results: the list of datasets • facets: the list of facets

3.1.6.3 Dynamic overview

This section describes the interactions between the Data Catalogue, the Identity/Authorization Management, the Advanced Visualization and a generic External Source providing DCAT-AP metadata. The interaction between the Data Catalogue and the Data Storage & retrieval component are described into section 3.1.5.3.

Figure 10 depicts the high-level interactions of an administrator with the components during the federation of an external source[5]. The administrator must login first, providing valid credentials that are forwarded to the Identity/Authorization Management component that validates the credentials and returns an access token. The token is used to interact with the Data Catalogue administrator APIs. The administrator will then add the external source to the Data Catalogue federation providing some information about the source itself, depending on the nature of the External Source (for instance, the API endpoint in the case of an Open Data portal). The Data Catalogue will retrieve from the External Source the DCAT-AP metadata that will be made available to the user to search datasets.

[5] The Data Catalogue will offer a user interface that will be included in the URBANITE UI; to simplify the representation of the interactions, the URBANITE UI is not represented in the diagrams, but each actions of the administrator or of the user is executed against the URBANITE UI and then forwarded to the Data Catalogue.

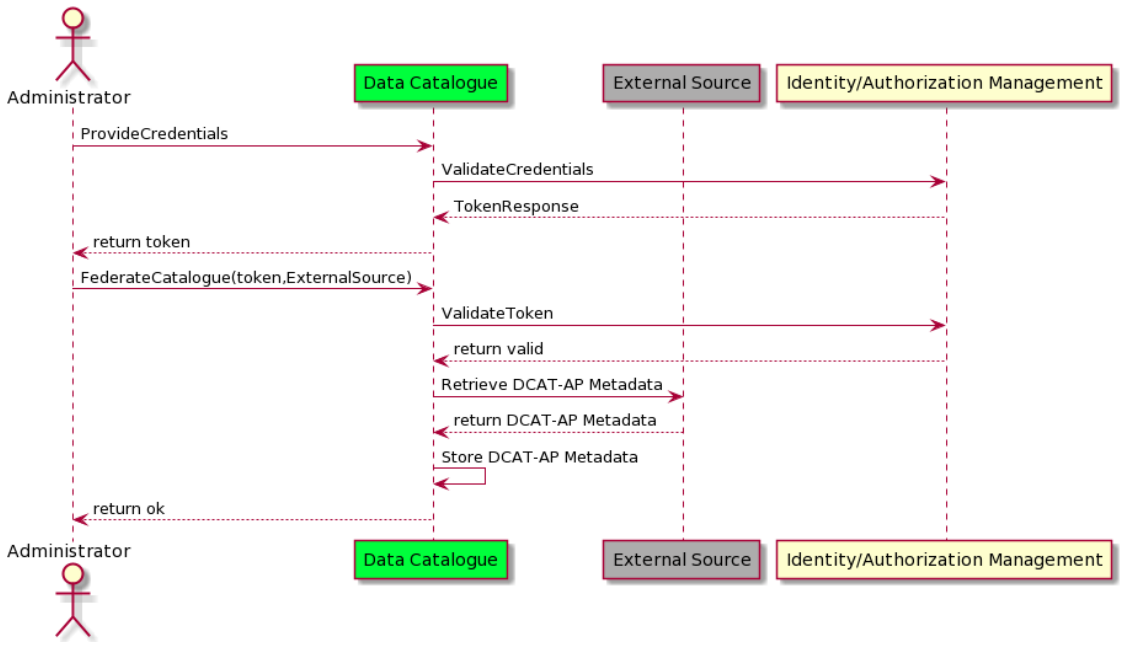

Figure 10: Data Catalogue - Administrator Sequence diagram

Figure 11 depicts the interactions between a user and the Data Catalogue to build a visualization taking advantage of the Advanced Visualization component functionalities. The user performs a search over the DCAT-AP datasets managed by the Data Catalogue component and selects a dataset among the ones identified by the Data Catalogue. The user then selects a distribution that is used to build the visualization through the Advanced Visualization component.

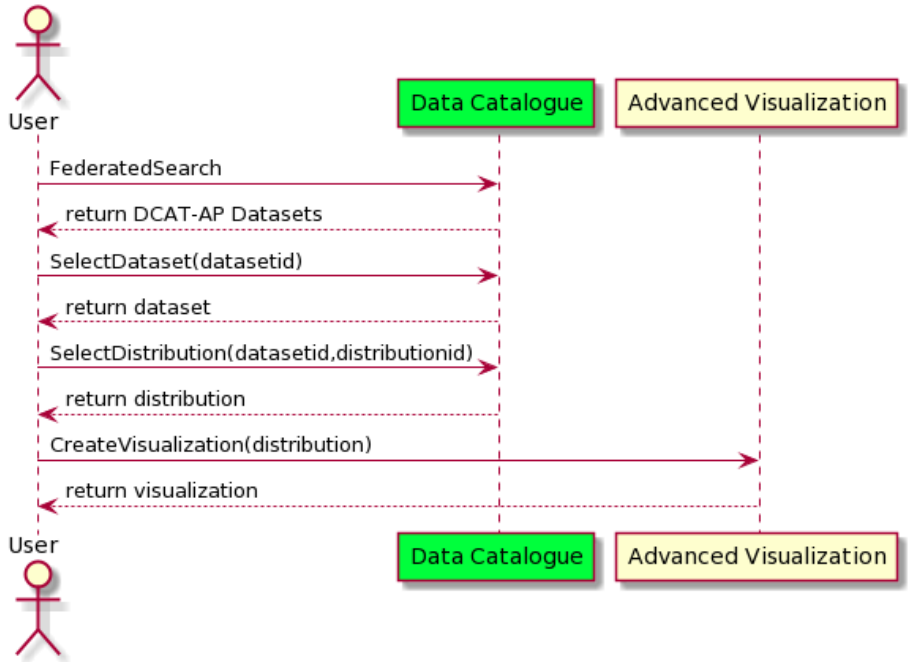

Figure 11: Data Catalogue - User Sequence diagram

3.1.6.4 Design implications based on the tool choice

As reported in section 3.1.6.1 the candidate tool for the realization of the Data Catalogue is Idra. Idra is composed by two components, a backend module and a web portal that offers its user

interface (for instance, to allows the users to search among the available datasets); the backend module provides a set of DCAT_AP REST APIs[6] to interact with the platform along with a set of CKAN[7] compliant APIs.

The technological stack of the backend module is made by Java 8, MySQL v5.7, RDF4J Server v2.2.1 and Apache SOLR-Lucene v6.6.0. The front-end module is based on AngularJs. The application server used to deploy the platform is Apache Tomcat v8.5. A detailed list of libraries and frameworks used within Idra is available in its documentation[8].

3.2 URBANITE components for data analysis

3.2.1 Controller

3.2.1.1 Main functionality

The Controller is responsible for the orchestration of the components of URBANITE Ecosystem devoted to the analysis of data. In this sense, the main functionality of the controller is the management and execution of the workflows that orchestrate the steps to be performed through these components.

To this aim, the Controller interacts with the other components and services of the URBANITE Ecosystem. These are linked together through the definition of workflows. For this specific purpose, Apache Airflow[9] has been chosen as the candidate tool for the realization of the Controller.

Airflow is written in Python language and allows defining workflows following the principle of "configuration as code"[10]. Indeed, to manage workflow orchestration in Airflow a file (named DAG[11]) should be written in Python to describe a workflow and its tasks. In the URBANITE Ecosystem, the defined tasks will interact with the components of the URBANITE Ecosystem itself (in particular with the ones that perform the analysis of the data, but potentially with any other).

3.2.1.2 Structural overview

The Controller (as the orchestrator of the URBANITE Ecosystem) interacts with diverse components, as shown in Figure 12.

For instance, the Controller could interact with the Data Storage & Retrieval component to retrieve the data location and orchestrate all components needed to analyze a specific dataset (such as a machine learning model to perform predictions) providing them information on how to access the data they require.

[7] https://ckan.org/

[8] https://idra.readthedocs.io/en/latest/admin/installation/

[9] https://airflow.apache.org/

[10] Configuration as code approach allow to define the configuration of a servers, code, or other resources as a text or script file (configuration file) managed in a repository.

[11] DAG stands for Direct Acyclic Graph. A DAG if a workflow definition in python code. More information can be found at https://airflow.apache.org/docs/apache-airflow/stable/concepts.html?highlight=dag#dags

The Controller can execute different workflow, on the basis of specific needs. Figure 12 depicts a non-exhaustive structural diagram of the relations between the Controller and the other components of the URBANITE Ecosystem.

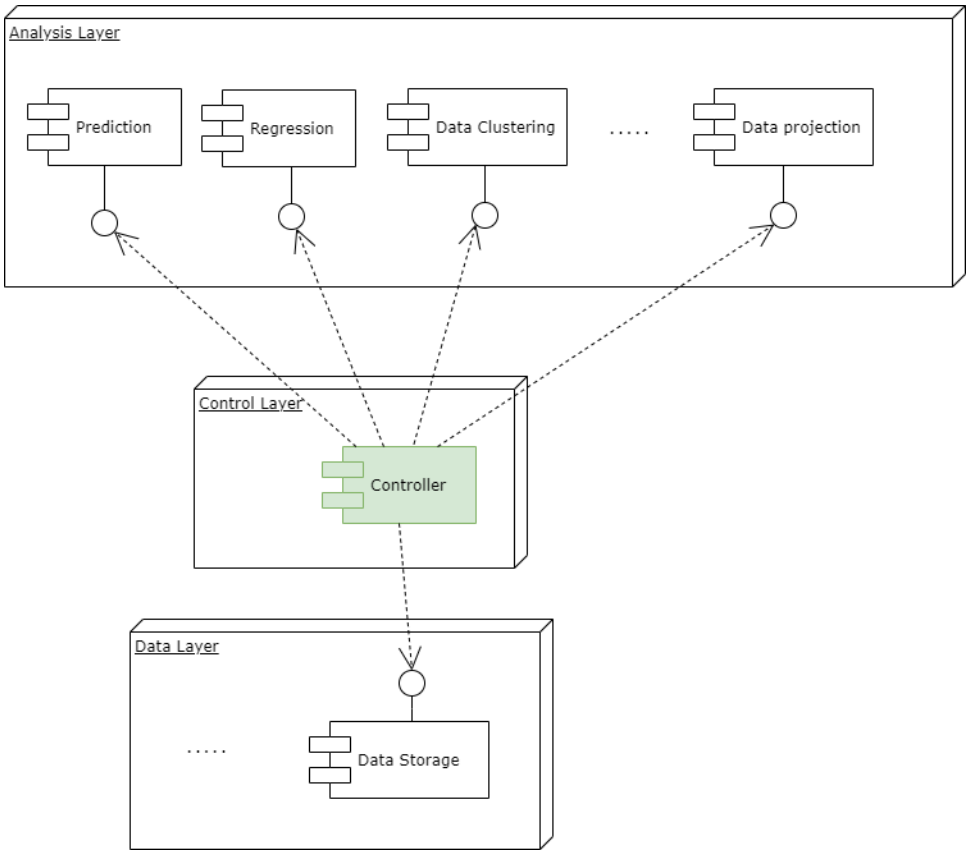

Figure 12: Controller - Relations between the Controller and the other components of the URBANITE Ecosystem

3.2.1.3 Dynamic overview

This section describes how the Controller can interact with the other components of the URBANITE Ecosystem. Figure 13 depicts a sequence diagram about a hypothetical scenario in which a generic "actor" (such as another URBANITE component or an end-user) invokes the execution of a workflow. In this hypothetical scenario, in addition to the Data Storage & Retrieval component, two generic components for data analysis are involved (*Analysis Component A* and *Analysis Component B*).

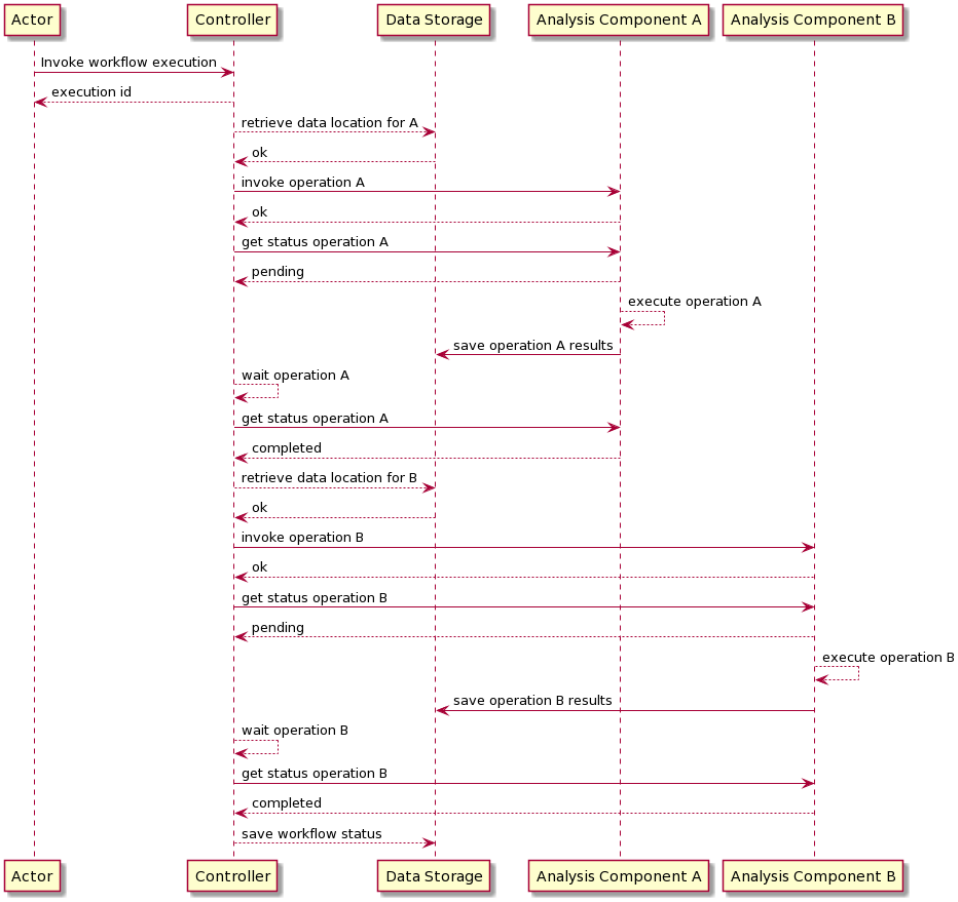

Figure 13: Controller - Sequence diagram

The flow starts with the "actor" that invokes the Controller to execute a specific workflow.

The Controller orchestrates three different components, respectively: The Data Storage & Retrieval (to retrieve data location and to store metadata about workflow execution), and two different components of the analysis layer in the hypothetical scenario are invoked one after the other.

More specifically, the workflow depicted is divided into two main steps. In the first one, the controller retrieves and pass data location to the *Analysis Component A* and then wait for the completion of the analysis by this last one. Once the analysis is completed, the *Analysis Component A* saves the results into the *Data Storage & Retrieval*. In the second step, the controller retrieves and pass data location (I.e., results obtained from *Analysis Component A*) to the *Analysis Component B*; also, in this case, the Controller wait for the completion of the analysis from *Analysis Component B*. As for the previous step, *Analysis Component B* saves the obtained results into the *Data Storage & Retrieval*. At the end of the entire workflow execution, even the execution status is saved in the Data Storage & Retrieval too.

3.2.1.4 Design implications based on the tool choice

As reported in section 3.2.1.1 the candidate tool for the realization of the controller component is Apache Airflow.

Version 2.X of Airflow is based on Python 3.X language and Flask[12] framework. Airflow exposes a set of RESTful APIs to interact with its functionalities. Some of the most useful APIs are reported in the following tables. For a more detailed list, you can refer to official documentation[13]that provides a detailed Airflow description.

GET **/dags/{dag_Id}/dagRuns**	
List DAG runs. This endpoint allows specifying the dag_id to retrieve DAG runs.	
Parameters	
limit : The numbers of items to return. (default value 100) offset : The number of items to skip before starting to collect the result set. dag_id *(mandatory): The DAG id.	
Responses	
200	List of DAG runs
401	Request not authenticated due to missing, invalid authentication info.

POST **/dags/{dag_id}/dagRuns**	
Trigger a new DAG run.	
Parameters	
dag_id *(mandatory): The DAG id.	
Payload	
JSON Object	dag_run_id: A string with the Run ID. (If not provided a value will be generated based on execution_date.) execution_date: A string with the execution date. This is the time when the DAG run. state: A string with the DAG state. Enum: success, running, failed
Responses	
200	Success.
400	Client specified an invalid argument.
401	Request not authenticated due to missing, invalid authentication info.
403	Client does not have sufficient permission.

[12] https://flask.palletsprojects.com/en/1.1.x/

[13] https://airflow.apache.org/docs/

404	A specified resource is not found.
409	The resource that a client tried to create already exists.

DELETE /dags/{dag_id}/dagRuns/{dag_run_id}	
Delete a DAG run.	
Parameters	
dag_id* (String)	The DAG Id.
dag_run_id* (String)	The DAG run Id.
Responses	
204	Success.
400	Client specified an invalid argument.
401	Request not authenticated due to missing, invalid authentication info.
403	Client does not have sufficient permission.

POST /dags/~/dagRuns/list	
This endpoint is a POST to allow filtering across a large number of DAG IDs, whereas a GET it would run into maximum HTTP request URL length limit.	
Payload	
JSON Object	dag_runs: array of these elements • dag_id: A string with the DAG ID. • dag_run_id: A string with the Run ID. • execution_date: A string with the execution date. This is the time when the DAG run. • state: A string with the DAG state.
Responses	
200	Success.
400	Client specified an invalid argument.
401	Request not authenticated due to missing, invalid authentication info.
403	Client does not have sufficient permission.

3.2.2 Data Projection

3.2.2.1 Main functionality

The data projection module provides functionality that allows the projection of data into lower dimensions and other transformations.

We are considering specific functionality:

- PCA (principal component analysis) is used to decompose a multivariate dataset into lower dimensionality while maximizing the amount of variance.
- Factor analysis is similar to PCA; however, the resulting components are not necessarily orthogonal.
- ICA (independent component analysis) allows the separation of a multivariate signal into additive, maximally independent components.

All these methods can be visualized to provide insight of some of the dataset's attributes, e.g. recognition of inherent clusters. The following methods have the advantage of being interactive when appropriately visualized:

- Projection pursuit is a method that generates a projection of datasets into lower dimensionality in a way that maximizes the projection's interestingness – the measure of interestingness is typically deviation from the normal distribution.

Targeted projection pursuit is a similar method with the advantage of interactivity. The user can manipulate the projection directly by using interactive visualizations.

3.2.2.2 Structural overview

PUT **/dataProjection/pca**	
Create and fit a new PCA model on the data.	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: "Helsinki")
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses	
200	Ok
400	Bad request

GET **/dataProjection/pca**
Returns the data, transformed using the PCA model.
Parameters

Responses	
200	ok
400	Bad request

PUT /dataProjection/factorAnalysis	
Create and fit a factor analysis model on the data.	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses	
200	Ok
400	Bad request

GET /dataProjection/factorAnalysis	
Returns the data, transformed using the factor analysis model.	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses JSON data object	
200	Ok
400	Bad request

3.2.3 Data Clustering

3.2.3.1 Main functionality

The main functionality of this module is to enable the clustering of the available data. More specifically, we are considering the following functionalities:

- K-Means clustering is a standard way of clustering data based on Euclidean distance. As this method is computationally effective, it is appropriate to use large datasets, as long as the clusters are somewhat balanced. This method cannot be used on serial data such as time series or series of locations.
- The affinity propagation method can be used on more complicated data types, including serial data and performs well even when the clusters are heavily unbalanced. Due to higher computational complexity, it is not appropriate for large data sets. The main use of this method will be used to cluster simulated trips and construct of traffic data models.
- Spectral clustering is a clustering method that is especially appropriate for clustering graph data and distance matrices. This method will be used to cluster simulated trips and generate traffic pattern models.
- Hierarchical clustering is a clustering method that is highly appropriate for clustering demographical data. The results of this method are highly intuitive hierarchical trees that can be easily explained and understood.

3.2.3.2 Structural overview

POST **/dataCluster/k_means**	
Create and fit a K-Means model on the data.	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses JSON – uuid of the created and trained model.	
200	Ok
400	Bad request
PUT **/dataCluster/k_means**	
Use an existing model to cluster the data.	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
Pre-trained model uuid (integer)	Uuid of the selected pre-trained clustering model.
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses	

200	Ok
400	Bad request
GET **/dataCluster/k_means**	
Returns the data, transformed using the K-Means model.	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses JSON projected data, according to the NGSI specification.	
200	Ok
400	Bad request

Similar API endpoints will be defined for other clustering methods discussed.

3.2.4 Self Organizing Map

3.2.4.1 Main functionality

The main functionality of Self-Organizing Maps is to enable an interactive overview of different data that cover the same population. SOMs are commonly used to generate low-dimensional (typically 2-D) views of high-dimensional data.

The Self-Organizing Map will be used for exploratory data visualizations implemented by the Advanced visualization module.

3.2.4.2 Structural overview

POST **/som**	
Create and fit a new self-organizing map model.	
Parameters	
model* (String)	Name of the data model, e.g. TrafficFlowObserved
Attribute* (string)	(Optional) name of the primary attribute for initial clustering.
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses JSON – uuid of the created and trained model, values of the map according to training attribute.	

200	Ok
400	Bad request
PUT **/som**	
Create and fit a new self-organizing map model.	
Parameters	
Attribute* (String)	Selected attribute to visualize
Uuid* (String)	Uuid of the selected pre-trained self-organizing map model
Responses JSON – values of the map according to the selected attribute.	
200	Ok
400	Bad request

3.2.4.3 Dynamic overview

The self-organizing map must first be created and trained using selected data. After training, the SOM will be visualized using the advanced visualization module. The attribute shown on the visualization of the SOM can be selected, and values will be updated accordingly.

3.2.4.4 Design implications based on the tool choice

None.

3.2.5 Correlation discovery

3.2.5.1 Main functionality

Correlation discovery is a process that highlights correlated attributes across data sets. The module will highlight interesting relations and help the users identify previously unknown data relations.

Main functionality includes:

- Loading the data.
- Pair-wise search for highly correlated attributes.
- Retrieval of partial results.
- Stopping the search on user input

3.2.5.2 Structural overview

POST **/correlation/data**	
Create a new correlation discovery object dased on selected data.	
Parameters	

model* (String)	Name of the data model, e.g. TrafficFlowObserved
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
data* (JSON Array)	Transformed data according to the NGSI specification.
Responses JSON – uuid of the created model.	
200	Ok
400	Bad request

GET **/correlation/status**	
Get the status of the correlation search and partial results when available.	
Parameters	
uuid* (String)	Identifier of the selected correlation discovery object.
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
Responses JSON – status of the correlation discovery (TBD), partial results (pairs of attributes and their correlations).	
200	Ok
400	Bad request

GET **/correlation/data**	
Get the complete correlation discovery results.	
Parameters	
uuid* (String)	Identifier of the selected correlation discovery object.
zone* (String)	Name of the geographic zone (e.g.: “Helsinki”)
Responses JSON – results (pairs of attributes and their correlations).	
200	Ok
400	Bad request

3.2.5.3 Dynamic overview

After the data is loaded, correlations of attributes are calculated pairwise in the order provided by a heuristic function. Because of this we can expect that attribute pairs with higher correlations will be discovered sooner, making the partial results useful even before the algorithm finishes.

The final results can be retrieved after the algorithm is stopped by the user or has calculated all the pairwise correlations.

3.2.5.4 Design implications based on the tool choice

None.

3.2.6 Prediction

3.2.6.1 Main functionality

To perform heuristic prediction for the vehicle flow at a location within the city by the processing of historic values measured by a fixed sensor and other information.

3.2.6.2 Structural overview

This component provides APIs for performing the different actions related to the predictive process. Specifically:

- List information about the sensors available for each city. This information includes the location, size of the data set, frequency and others.
- Train a new predictive model for a particular sensor.
- Query an existing model to obtain a prediction.
- List the models that are already available and the characteristics of them.

GET **/traffic/pred**	
To obtain an individual prediction	
Parameters	
features* (array)	Array of values to perform a query
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
id* (string)	Sensor identification (i.e. 248 for Bilbao, 62_1 for Helsinki)
Inference_type	0: Bayesian inference 1: Random Forest
Responses	

200	Ok { "code": 200, "lower": 48, "method": "GET", "num_vals": 47, "pred": 66.1368900352121, "upper": 87, "version": 0.1 }
400	Bad request

POST **/traffic/pred**	
To train specific model	
Parameters	
No parameters	
Request Body (JSON with the following fields)	
Ini_date	%YYYY-%M-%D
End_date	%YYYY-%M-%D
Num_features	1: [slot of the day] 2: [slot of the day, weekday] 3: [slot,weekday,month] 4: [slot,weekday,month,holiday] 5: [slot,weekday,month,holiday,schoolDay] 6: [slot,weekday,month,holiday,schoolDay, temperature] 7:[slot,weekday,month,holiday,schoolDay,temperature, precipitation] 8:[slot,weekday,month,holiday,schoolDay,temperature, precipitation,cruiseArrival] 9: other TBD
Inference_type	0: Bayesian inference 1: Random Forest
city	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
id	Sensor identification.
Responses	
200	ok

GET **traffic/pred_date**	
To obtain a prediction for a full day	
Parameters	
features* (array)	Array of values to perform a query (should be in accordance with the date given)
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
id* (string)	Sensor identification (i.e. 248 for Bilbao, 62_1 for Helsinki)
Inference_type	0: Bayesian inference 1: Random Forest
Responses	
200	Ok { "method": "GET", "result": [{ "lower": 8, "num_vals": 48, "pred": 15.474470722789533, "time": "2021-06-12 00:00:S", "upper": 29 }, { "lower": 7, "num_vals": 48, "pred": 13.696208371999424, "time": "2021-06-12 00:05:S", "upper": 24 }, …, { "lower": 8, "num_vals": 48, "pred": 17.252187718095623, "time": "2021-06-12 23:55:S", "upper": 26 }]}
400	Bad request

GET **/traffic/models**	
To obtain a list of the models already available within the platform	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
id (string)	Sensor identification (i.e. 248 for Bilbao, 62_1 for Helsinki)
Responses	

200	Ok JSONArray
400	Bad request

GET **/traffic/sensor**	
To obtain information about an individual sensor	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
id* (string)	Sensor identification (i.e. 248 for Bilbao, 62_1 for Helsinki)
Responses	
200	Ok { "code": 200, "delta_t": 5, "end_date": "2020-01-01 00:00:00", "ini_date": "2019-01-31 00:00:00", "lat": 60.176326, "lon": 24.958575, "method": "GET", "size": 96481, "version": 0.1}
400	Bad request

GET **/traffic/sensors**	
To obtain all the ids of the sensors for a given city	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
Responses	
200	Ok

	{ "code": 200, "ids": ["62_1", "62_2"], "method": "GET", "version": 0.1}
400	Bad request

GET **/traffic/sensors_ext**	
To obtain all the information of the sensors for a given city	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
Responses	
200	Ok { "code": 200, "ids": [{ "delta_t": 5, "end_date": "2020-01-01 00:00:00", "id": "621", "ini_date": "2019-01-31 00:00:00", "lat": 60.176326, "lon": 24.958575, "size": 96481 }, { "delta_t": 5, "end_date": "2020-01-01 00:00:00", "id": "622", "ini_date": "2019-01-31 00:00:00", "lat": 60.176326, "lon": 24.958575, "size": 96481 }], "method": "GET", "version": 0.1}
400	Bad request

3.2.6.3 Dynamic overview

The steps to obtain predictions for the flux of vehicles at a specific point, in general, involves the following steps:

- Choosing the location at which the prediction is going to be performed. In order to find the available locations, the user can check the positions of the sensors for the different cities. The services /traffic/sensors and /traffic/sensor provide information not only about the location but also about the data within the platform.
- Once the id of the sensor and the city has been chosen and in order to perform a prediction, a predicting model needs to be constructed and trained within the platform. The service /traffic/models/ list the different already available models.
- If there is no appropriate model within the URBANITE platform, a new model needs to be constructed. The training of a new model is performed the POST /traffic/pred model. This service only sends to command to start training the model, this is a costly process that can take a long period of time, which implies that from the command is issued until the new model is available can take a while.

For training a new model, the number of features to be considered in the model need to be specified; the following cases are considered within this module:

1: [slot of the day]
2: [slot, weekday]
3: [slot, weekday, month]
4: [slot, weekday, month, holiday]
5: [slot, weekday, month, holiday, schoolDay]
6: [slot, weekday, month, holiday, schoolDay, temperature]
7: [slot, weekday, month, holiday, schoolDay, temperature, precipitation]
8: [slot, weekday, month, holiday, schoolDay, temperature, precipitation, cruiseArrival]
… other possibilities to be defined.

- Once the model is available, the prediction can be performed, providing the parameters, the feature vector that corresponds to the moment when the prediction is requested. Two different GET services can be used: /traffic/pred, which provides a prediction at a particular instant of time and /traffic/pred_date, which provides a 24 hour expansion of predictions. In the former case, the number of predictions depends on the frequency of the data for the sensor located.

3.2.6.4 Design implications based on the tool choice

None.

3.2.7 Analytical Framework

3.2.7.1 Bicycle Analysis

3.2.7.1.1 Main functionality

This module involves several services with various functionalities related to the mobility of bicycles. More specifically, the module transforms GPS information obtain from the bicycles into more useful and actionable information. It also provides auxiliary services helpful in these transformations.

The following functionalities are considered:

- Computation of origin-destination matrixes (OD matrixes)
- Computation of the locations more popular for the bicycle trajectories stored within the platform.
- Computation of the trajectories more popular among the bicycle trajectories stored within the platform.

The following auxiliary functionalities, not necessarily for bike trajectories but applicable to data in general, are considered:

- Computation of Voronoi areas. Compute the tessellation of the city map composed of polygons that correspond to the points placed closest to a given set of points.

- Map-Matching of trajectories. Adjust a set of GPS points to the navigational network and perform reconstruction of the trajectories connecting the corrected locations.

3.2.7.1.2 Structural overview

GET **/bikeAnalysis/od_matrix**	
Parameters	
city* (integer)	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina
Inference_type	1: Bayesian 2: Random Forest
features	Array type of minimum size equal to 2, corresponding to the integers which define the zone_id for the origin and the destination.
Responses	
200	Ok
400	Bad request

POST **/bikeAnalysis/od_matrix**	
Create and train a new model for the computation of OD matrix	
Parameters	
No parameters	
Request Body (JSON with the following fields)	
Ini_date	%YYYY-%M-%D
End_date	%YYYY-%M-%D
Inference_type	1: Bayesian inference 2: Random Forest
city	0: Bilbao 1: Amsterdam 2: Helsinki

	3: Messina
num_features	0: no extra features considered, only Origin and Destination (OD). 1: OD + [slot] 2: OD + [slot, weekday] 3: OD + [slot, weekday, month] 4: OD + [slot, weekday, month, holiday] 5: OD + [slot, weekday, month, holiday, school_day] 6: others TBD
delta_t	Discretization of time. Needed to compute the slot within a day. Typical values 15 mins, 60 mins. The computation of the OD-Matrix integrates all the trips realized within the given time interval.
polys	Geojson of type FeatureCollection that consists polygons defining the origins and the destinations for the output matrix. Each polygon should include an integer property "zone_id" used internally to identify each polygon.
Responses	
200	ok

GET **/bikeAnalysis/trajectory_analysis**	
Parameters	
city* (integer)	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina
analysis_type	0: computation of popular points 1: computation of popular trajectories 2: TBD
features	Array that defines the features considered for the analysis.
Responses	
200	Ok For analysis_type equal 0 a collection of points with an index of popularity. The collection of points includes all the points in the navigational network. For analysis_type equal 1, a collection of trajectories.

400	Bad request

POST /bikeAnalysis/ trajectory_analysis	
Create and train a new model for the computation of O/D matrix	
Parameters	
No parameters	
Request Body (JSON with the following fields)	
Ini_date	%YYYY-%M-%D
End_date	%YYYY-%M-%D
analysis_type	0: computation of popular points 1: computation of popular trayectories 2: TBD
city	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina
num_features	0: no extra features considered,. 1: [slot] 2: [slot, weekday] 3: [slot, weekday, month] 4: [slot, weekday, month, holiday] 5: [slot, weekday, month, holiday, school_day] 6: others TBD
delta_t	Discretization of time. Needed in order to compute the slot within a day. Typical values 15 mins, 60 mins.
Responses	
200	Ok

PUT /bikeAnalysis/voronoi_areas

Produces the Voronoi areas for a given set of points.
Parameters
No parameters
Request Body GeoJSON of type FeatureCollection. The collection consists of: N features of type Points, each with an integer property “name” used internally to give a zone_id to the resulting polygons. A single feature of type LineString which defines the city limits and bounds the resulting polygons.

Responses
 	
JSON
 	
{
 	
"code": 200,
 	
"method": "PUT",
 	
"polys": {GEOJSON of type “FeatureCollection” that consists of polygons defining the origins and the destinations for the output matrix. Each polygon should include an integer property “zone_id” used internally to identify each polygon.},
 	
“version”: 0.1
}

200	ok

PUT **/bikeAnalysis/mm**
Clean a trajectory performing Map Matching to the Navigational network.
Parameters
No parameters
Request Body (JSONArray) With at least two elements of the following type:

t	Timestamp in milliseconds at which the GPS capture has been obtained
la	Float that corresponds to the latitude of GPS capture using WGS 84 Projection.
lo	Float that corresponds to the longitude of GPS capture using WGS 84 Projection.

Responses (JSONArray) same type of output as the input but corrected.	
200	ok

3.2.7.1.3 Dynamic overview

For the Origin Destination functionality, a set of polygons need to be passed to the service. In order to obtain a proper tessellation, the Voronoi service can be called to produce a proper geojson of polygons to be consumed by the OD matrix service.

For the trajectory_analysis, the trajectories need to be matched to the navigational network; this can be obtained by means of the map-matching (MM) service.

3.2.7.1.4 Design implications based on the tool choice

None

3.3 URBANITE components for decision support

3.3.1 Traffic simulation

3.3.1.1 Main functionality

Is to provide the simulations of traffic under specified conditions. Those include the proposed mobility policy, different weather conditions, changes to the traffic infrastructure etc. The traffic simulation will support:

- Importing the traffic network of the city as provided by the geographic maps from OpenStreetMap.
- Importing district shapes in order to correctly interpret the demographical data.
- Generating the population model according to the demographical data of the city.
- Generating the traffic demand model according to the population model and available traffic data.
- Multi-modal traffic simulation including: cars, bicycles, public transport, heavy traffic and pedestrians.
- Access to the simulation results for further analysis and visualizations.

3.3.1.2 Structural overview

POST **/trafficSimulation/scenario**	
Creates a new and empty simulation scenario.	
Parameters	
Simulation sceraio uuid	integer
Description	string
Name	string
City (integer)	0: Bilbao

	1: Amsterdam 2: Helsinki 3: Messina
Request Body – empty	
Responses	
200	ok
400	

PUT **/trafficSimulation/network**	
Imports the network file and include it in the simulation scenario.	
Parameters	
Level of detail (integer)	0: all streets and roads 1: collector streets, arterial streets and main roads 2: arterial streets and main roads 3: main roads only
City (integer)	0: Bilbao 1: Amsterdam 2: Helsinki **3: Messina**
Request Body XML Network file (as defined: "http://www.matsim.org/files/dtd/network_v2.dtd")	
Responses	
200	ok
400	

PUT **/trafficSimulation/district_shapes**	
Imports the shape files of the city districts.	
Parameters	
City (integer)	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina

Request Body Shapefile of districts shapes (ESRI Shapefile)	
Responses	
200	ok
400	

PUT **/trafficSimulation/demo_data**	
Imports the demographical data for population model generation.	
Parameters	
City (integer)	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina
Request Body Demographical data per district (TBD)	
Responses	
200	ok
400	

PUT **/trafficSimulation/population_model**	
Generate the population model based on demographical data and parameters.	
Parameters	
City (integer)	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina
TBD	
Request Body – empty	
Responses	
200	ok
400	

PUT **/trafficSimulation/traffic_demand_model**	
Generate the traffic demand model based on the population model and parameters.	
Parameters	
City (integer)	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina
TBD	
Request Body – empty	
Responses	
200	ok
400	

PUT **/trafficSimulation/vehicles**	
Import the vehicles definition files.	
Parameters	
City (integer)	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina
Request Body Vehicle definition files (http://www.matsim.org/files/dtd/vehicleDefinitions_v1.0.xsd)	
Responses	
200	ok
400	

GET **/trafficSimulation/results**	
Retrieve the simulation results for analysis and visualizations.	
Parameters	
Simulation scenario uuid	integer

City (integer)	0: Bilbao 1: Amsterdam 2: Helsinki 3: Messina
Request Body - empty	
Responses	
200	ok
400	

3.3.1.3 Dynamic overview

The traffic simulation needs to be prepared before running it. In order to provide the required data, the following data needs to be provided to the traffic simulation service:

- Traffic network is generated using the city maps. Based on the target scenario different subsets of the network edges and vertices, for example, the whole city, only main roads, a specific part of the city, or only the main roads in a specific part of the city.
- Population demand is generated based on the known demographical data. In order to enable the simulation of hypothetical scenarios or certain mobility policy proposals, the population model can be parametrized.
- Different types of vehicles are represented with appropriate attributes. These need to be set up appropriately to the task, e.g. when analyzing air quality, the emissions of the vehicles are needed.
- The traffic demand model is generated and optimized based on agents' local knowledge.
- The traffic demand model is refined using the available traffic data.
- The simulation is run, and the results gathered.

3.3.1.4 Design implications based on the tool choice

Due to the selection of the traffic simulation tool MATSim, the process of preparing the simulation scenarios and running the simulation has been appropriately changed to support the requirements of the URBANITE project.

- The process of preparing the simulations is guided by a wizard-like UI construct in order to simplify the preparation and enable the non-technical users to use the traffic simulation module.
- Importing traffic network from OpenStreetMaps is streamlined and a third-party network editor is provided.

The population model is generated in a multi-step process that enables the parametrization of the population model, gives the users control over the traffic demand generation process and is compatible with the tool's technical requirements

3.3.2 Policy simulation and validation

3.3.2.1 Main functionality

The main functionality of the policy simulation and validation module is to enable the encoding of the proposed policies, design the validation scenarios and validate the policy using the traffic simulation and KPI calculation, thus enabling the policy validation.

3.3.2.2 Structural overview

TBD

3.3.2.3 Dynamic overview

TBD

3.3.2.4 Design implications based on the tool choice

TBD

3.3.3 Recommendation engine

3.3.3.1 Main functionality

The main functionality of the recommendation engine is to support the process of the mobility policy design; more specifically, it will offer recommendations for creating the evaluation scenario and evaluation framework for the proposed policies:

- Support the generation of evaluation scenarios for specific proposed policies using a scenario creation wizard UI tool.
- Support the selection of appropriate KPIs and enabling their calculation.
- Support the definition of the decision model, including choosing the attributes.
- Enable the comparison of multiple policies with the same evaluation framework.

The steps considered for the scenario creation wizard are the following:

- Select the map region that will be simulated.
- Select the level of detail for the region.
- (Optionally) use a network editing tool to introduce changes to the network.
- Select the relevant KPIs from the list of available KPIs.
- (Optionally) select the relevant simulation attributes and define the calculation of custom KPIs.
- Define the decision model by selecting the relevant attributes, the attribute and KPI hierarchy and the decision rules for the comparison of different evaluation scenarios.
- (Optionally) select the optimizations of proposals when applicable, including metrics to minimize/maximize.
- Select the proposed variations of the evaluation's scenarios based on available data, such as:
  - Weekday traffic demand variation vs weekend traffic demand.
  - Sunny, rainy and snowy variations of the traffic demand.
  - Number of randomized variations to consider.

  Using multiple variations of evaluation simulations will ensure the robustness of the KPI calculations.

Using the proposed process, the module provides:

- Multi-criteria decision analysis for comparing mobility policy proposals.

- Robust calculations of KPIs.
- Provide recommendations for creating the evaluation scenario.
- Provide an evaluation framework for the policy proposals.

Provide a mechanism for comparison of policy proposals using a common framework.

3.3.3.2 Structural overview

PUT **/recommendation/decision_model**	
Create or replace the wizard-generated decision model.	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
Scenario uuid	integer
Attributes array	JSON array of attributes
Scales array	JSON array of scales corresponding to attributes
Hierarchical decision model	JSON tree structure of the attributes
Decision rules	JSON dictionary of rules for attribute agglomeration
Responses	
200	Ok
400	Bad request

PUT **/recommendation/kpi**	
Create or replace a KPI.	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
Scenario uuid	integer

Attributes array	JSON array of attributes required to calculate the KPI.
Function	The function that calculated the KPI.
Optimize	Boolean – should the simulation optimize for this KPI or not
Optimization target (optional)	0: minimize KPI 1: maximize KPI
Responses	
200	Ok
400	Bad request

PUT **/recommendation/scenario_variation**	
Create or replace variations of a scenario.	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
Scenario uuid	integer
Attributes array	JSON array of attributes to vary
Values	Values of the attribute to be simulated.
Responses	
200	Ok
400	Bad request

POST **/recommendation/scenario_evaluate**	
Start the evaluation of the generated scenario.	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki

	3: corresponds to Messina
Scenario uuid	integer
Responses: JSON including results of decision analysis, KPIs calculated	
200	Ok
400	Bad request

GET **/recommendation/results**	
Start the evaluation of the generated scenario.	
Parameters	
city* (integer)	0: corresponds to Bilbao 1: corresponds to Amsterdam 2: corresponds to Helsinki 3: corresponds to Messina
Scenario uuid	integer
Responses	
200	Ok
400	Bad request

3.3.3.3 Dynamic overview

The proposals are described, and an evaluation scenario is generated for each of the proposals. For each evaluation, scenario variations are generated based on the user's input. Each variation is then simulated, and the results are collected.

The user selects the appropriate KPIs and is optionally guided to create custom KPIs based on choosing the relevant attributes and defining the function to calculate.

The user is guided through the creation of a decision model used for the comparison of proposals. The decision model, including the decision attributes, scales, rules, and hierarchical model, is stored.

Using the results of the simulations, selected and custom KPIs are calculated. These can be retrieved for visualization and further analysis.

Using the calculated KPIs the decision model performs multi-criteria decision analysis. The results of the decision analysis can be retrieved for visualisation or further analysis.

3.3.3.4 Design implications based on the tool choice

None.

3.3.4 Advanced visualization

3.3.4.1 Main functionality

The main functionality is to provide a different kind of visualizations, appropriate to the data displayed. The advanced visualizations module will consist of the front-end implementation of visualization methods and calls to appropriate API endpoints to get the data to represent.

Specific visualization types considered are detailed in the URBANITE deliverable D4.1.

The main functionality includes:

- Retrieve the data to visualize.
- Allow the selection of visualization type to use.
- (Optionally) recommend the appropriate visualization type based on the metadata.
- Interact with visualizations (when applicable).

3.3.4.2 Structural overview

This module is implemented on the front-end. It will consist of UI elements needed for showing the visualizations, local storage for visualization data and call-backs to appropriate API endpoints.

3.3.4.3 Dynamic overview

After the user selects data to visualize, the data will be retrieved from the appropriate service, and minimal data needed for visualizations will be stored locally.

Appropriate visualizations will be proposed based on the metadata available. The user will be able to choose among the proposed visualizations as well as other types of visualisation.

3.3.4.4 Design implications based on the tool choice

3.4 URBANITE virtual SoPoLab

3.4.1 Main functionality

The Virtual SoPoLab, also called URBANITE Forum[14], offers a virtual environment where citizens discuss and propose new ideas and/or challenges following a co-creation approach. This component takes advantage of the participatory democracy platform Decidim[15]. Within URBANITE, the Virtual SoPoLab leverages the Assembly[16] functionality provided by Decidim to manage and organize discussions and interactions among the users.

Figure 14 depicts the main actors and functionalities of the Virtual SoPoLab.

[14] https://forum.urbanite-project.eu/

[15] http://decidim.org/

[16] https://docs.decidim.org/en/features/participatory-spaces/#_assemblies

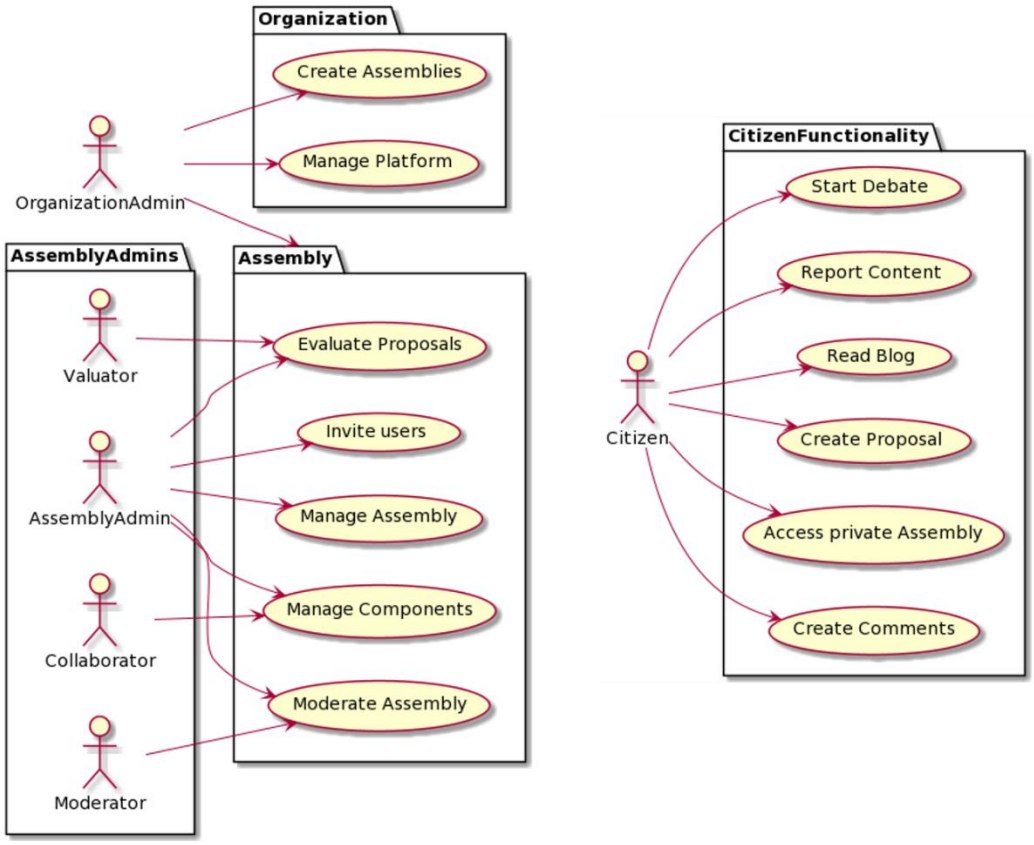

Figure 14: Virtual SoPoLab - Use Cases diagram

The Organization Administrator is in charge of managing the entire Virtual SoPoLab and he/she is the only one able to create new assemblies. Moreover, this user can perform every operation within the Virtual SoPoLab; indeed, he/she is able to also manage assemblies.

Considering the functionalities of the assemblies, the following roles can be defined: *Administrator*, *Collaborator*, *Valuator* and *Moderator*. The Administrator is able to manage the entire assembly functionalities; the Collaborator manages the component (such as blog, forum, etc.) to be made available within an assembly, assigns proposal to a Valuator or evaluate the proposal by himself/herself; the Valuator assesses the proposals; and the Moderator moderates the assembly content, if any report from citizens is provided.

The Citizens can start new debates or participate in the existing ones providing comments; they can create new proposals to be further evaluated by the administrators; they access to the blog posts provided by the administrators, and they can create reports on existing contents that will be further checked by the moderators.

Additional details about the Virtual SoPoLab (URBANITE Forum) can be found in the URBANITE Forum guide[17].

3.4.2 Structural overview

Even if the Virtual SoPoLab is not strictly part of the URBANITE Ecosystem (ad depicted in section 2.1), some potential interactions between the Virtual SoPoLab and the rest of the URBANITE Ecosystem are planned. For instance, a potential interaction would consist of giving to the user

[17] https://urbanite-forum-guide.readthedocs.io/en/latest/

the chance to integrate the visualization provided by the Advanced Visualization component (section 3.3.4) to the proposals and/or the debates managed in the Virtual SoPoLab to drive the discussion among the users. The interaction between the two components (the Virtual SoPoLab and the Advanced Visualization components) is depicted in Figure 15.

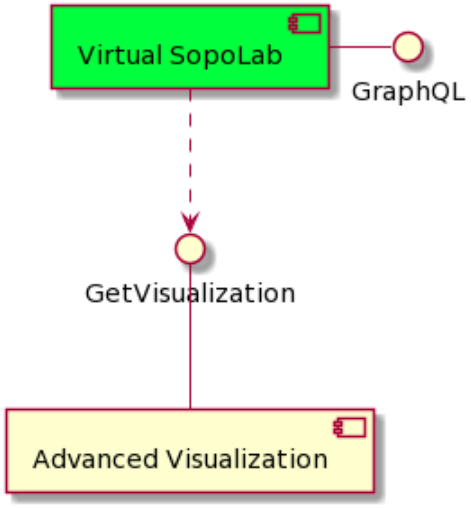

Figure 15: Virtual SoPoLab - Component diagram

The Virtual SoPoLab exposes a GraphQL[18] interface to query its database.

3.4.3 Dynamic overview

Figure 16 depicts the expected flow of the possible interactions between the Virtual SoPoLab and the Advances Visualization components to include in a proposal or debate.

[18] https://graphql.org/

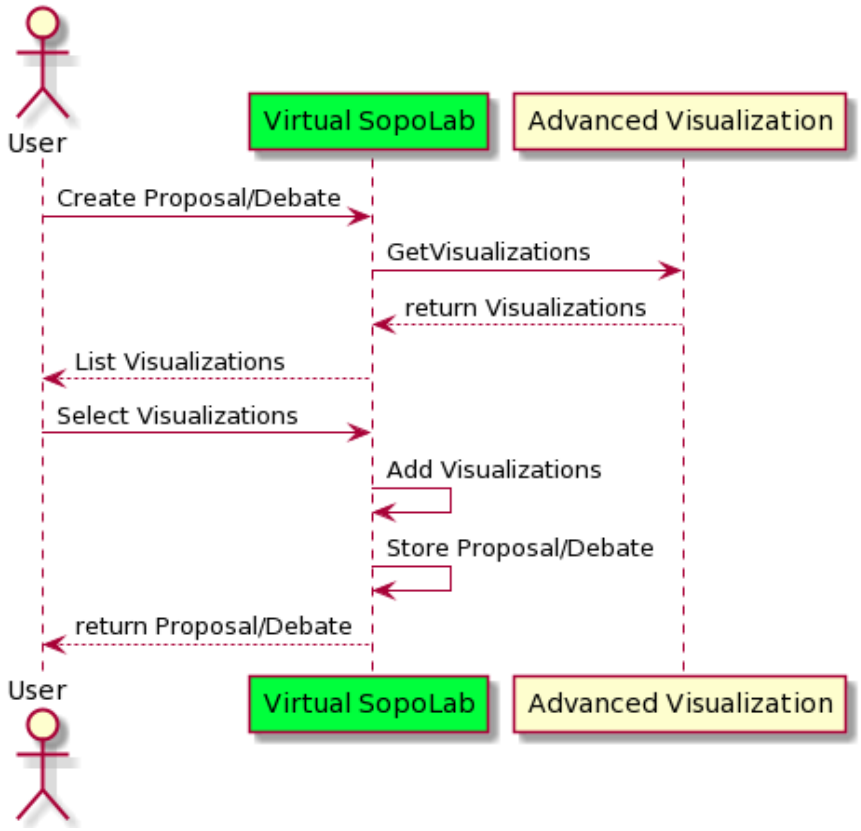

Figure 16: Virtual SopoLab - Sequence diagram

The first step consists of a user that wants to create a new proposal or debate in the Virtual SoPoLab. The Virtual SoPoLab retrieves the available visualizations from the Advanced Visualization component and lists them to the user. Then the user selects a visualization to be added to the new proposal or debate. Finally, the Virtual SoPoLab add the selected visualization, store the proposal or debate and returns the newly created element (the proposal or the debate) to the user.

3.4.4 Design implications based on the tool choice

Decidim (the tool selected for the realization of the Virtual SoPoLab) is a Ruby on Rails[19] application composed of several *gems*[20] that can be enabled and/or disabled to create a custom version of the platform. Moreover, additional gems can be created and added to Decidim to provide new functionalities within the Virtual SoPoLab. The database used within the Virtual SoPoLab is PostgreSQL[21] v10. The platform also supports a deployment based on docker.

3.5 Integrated URBANITE UI

3.5.1 Main functionality

The URBANITE UI component is intended to be the first point of access to the different components of the URBANITE platform. The several options for integrating the components into the URBANITE UI are described in the Deliverable D5.3 Integration Strategies [2]. For instance,

[19] https://rubyonrails.org/

[20] A *gem* is a software package which contains a packaged Ruby application or library.

[21] https://www.postgresql.org/

the URBANITE UI will offer the possibility to integrate cockpits or dashboards taking advantage of the capabilities provided by the Advanced Visualization component.

Figure 17 illustrates the main actors and use cases of the URBANITE UI.

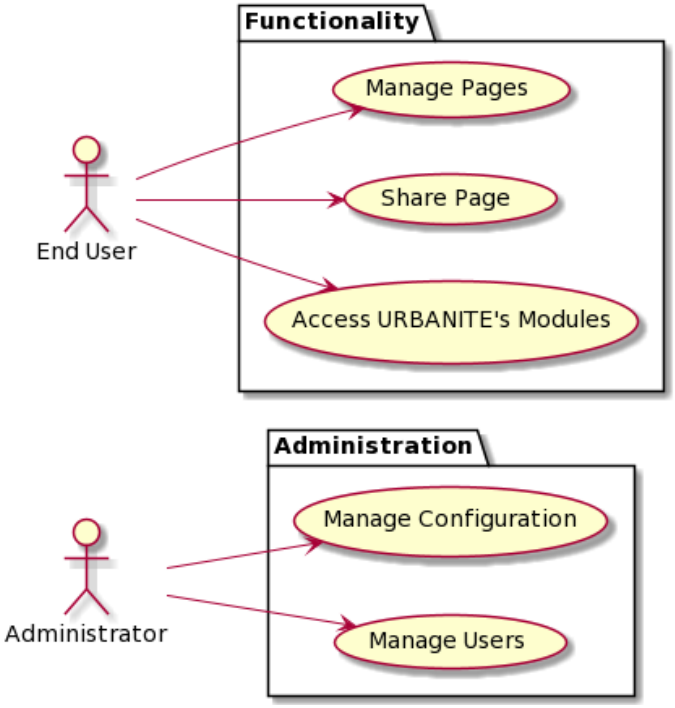

Figure 17: URBANITE UI - Use Case diagram

The Administrator is in charge of managing the configurations of the component and the users. End Users through the UI, will access the different integrated modules. Moreover, they will be able to create, modify or delete pages taking advantage of the visualizations provided by the Advanced Visualization component. Finally, the user will be able to share these created pages among each other.

3.5.2 Structural overview

The URBANITE UI, being the entry point of the different modules of the platform, will mainly interact with most of them. Considering the use cases depicted in Figure 17, the URBANITE UI interacts with the Identity/Authorization Management component and the Advanced Visualization components. The Figure 18 depicts the interactions at the component level.

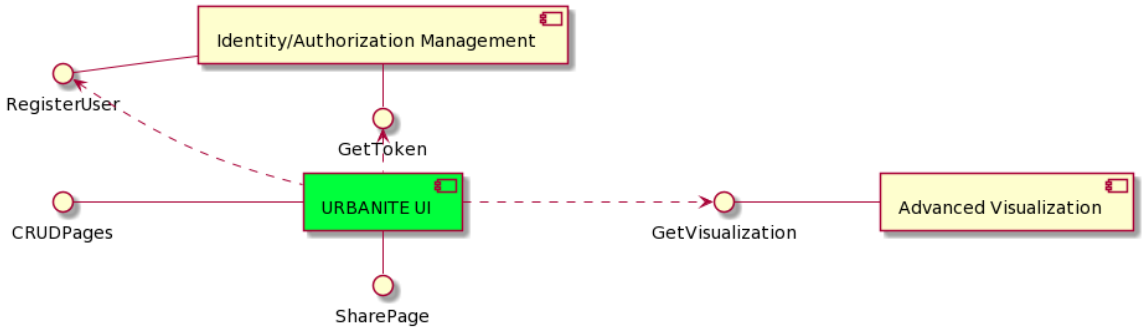

Figure 18: URBANITE UI - Component diagram

The UI interacts with the Identity/Authorization Management component to retrieve the authentication token for an existing user or to allow a new user to register. Once the user logs in into the UI, he/she will interact with the Advanced Visualization component to retrieve the visualization that will be used to create and share pages.

The URBANITE UI will expose the functionalities to create, read, update and delete the pages. Moreover, the Share Page interface will be used to share a page with other user and/or to a group of users.

3.5.3 Dynamic overview

The following Figure 19 depicts the interactions among the components to create a new page for the user taking advantage of the visualizations provided by the Advanced Visualization component.

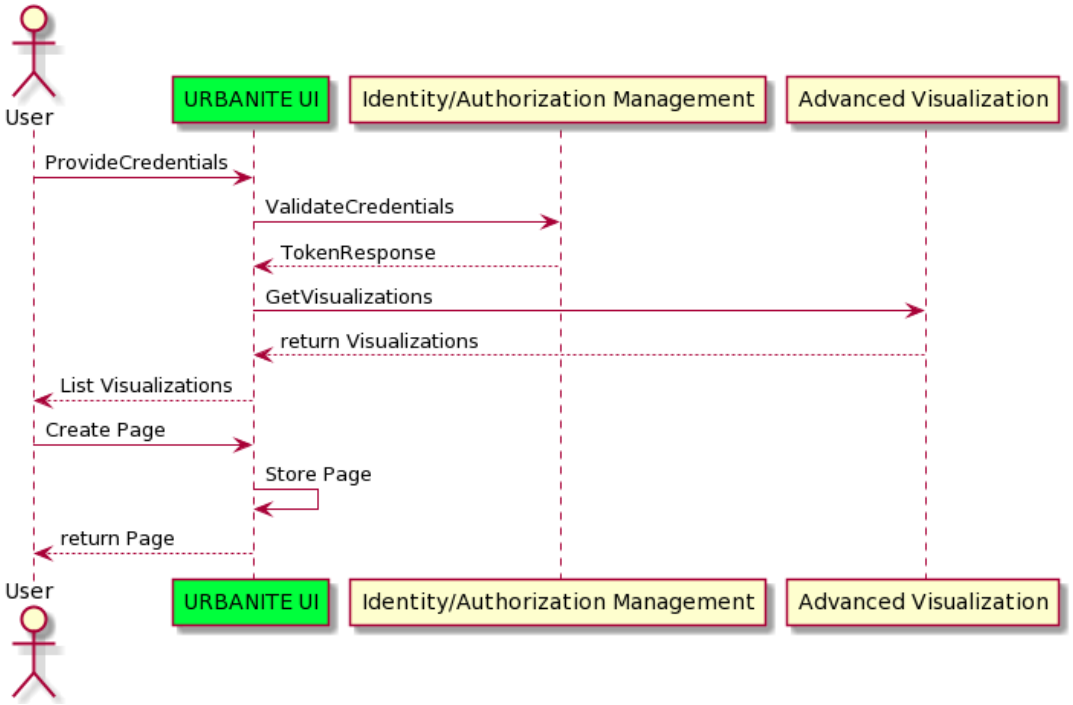

Figure 19: URBANITE UI - Sequence diagram

To access the URBANITE UI the user should provide credentials that are validated over the Identity/Authorization Management component. Providing valid credentials, the Identity/Authorization Management component will return an authentication token to the user that will allow to access the URBANITE UI. From the UI, the user builds his/her custom page. The UI retrieves the available visualizations from the specific Advanced Visualization modules. The visualizations will be used through the UI to compose a custom page that will finally be stored.

3.5.4 Design implications based on the tool choice

The URBANITE UI is a front-end application built using ngx-admin[22] framework. This framework is based on Angular[23] and a Nebular[24], an Angular UI Library that provides Eva Design System[25] features.

To allow users to create and manage their custom pages, a backend module is going to be provided. This module is a SpringBoot[26] application, written in Java, that exposes a set of REST API to manage the CRUD operations for the pages and to share such pages among the users. The underlying database is MongoDB[27].

3.6 Identity/Authorization Management

3.6.1 Main functionality

The Identity/Authorization Management component is in charge of securing the access to the other URBANITE's component, resources and services, whether security is needed. It will manage the authentication and authorization of the users to access the different functionalities of URBANITE Platform. The candidate tool for the realization of this component is Keycloak[28], an open-source Identity and Access management tool, that adopt standard protocols such as OpenID Connect[29], Oauth2[30] and SAML[31]. Moreover, among the several functionalities, Keycloak allows to configure single-sing on, to login using social accounts (e.g. GitHub, Google) and offers an administration console. Finally, the tool allows managing users and applications, giving the chance to register, modify or delete new user or application. Please, refer to the official documentation[32] for further details.

3.6.2 Structural overview

Each component in the URBANITE's architecture could be configured to interact with the Identity/Authorization Management component if authentication and authorization functionalities are needed. Sections 3.1.6.2 and 3.5.2 describes the interactions between the Identity/Authorization component and the Data Catalogue component (3.1.6.2) and the URBANITE UI (3.5.2). The backend functionalities provided by these components are configured as Resource Servers[33] and the Identity/Authorization component is in charge of authorizing each request following if needed, specific role-based rules. Integrations with other tools could be performed following the same solution described previously.

Keycloak provides a set of REST APIs[34] for administering the tool. Moreover, it provides the Oauth2 standard set of APIs.

[22] https://akveo.github.io/ngx-admin/
[23] https://angular.io/
[24] https://akveo.github.io/nebular/
[25] https://eva.design/
[26] https://spring.io/projects/spring-boot
[27] https://www.mongodb.com/
[28] https://www.keycloak.org/
[29] https://openid.net/connect/
[30] https://oauth.net/2/
[31] http://saml.xml.org/saml-specifications
[32] https://www.keycloak.org/documentation
[33] https://www.oauth.com/oauth2-servers/the-resource-server/
[34] https://www.keycloak.org/docs-api/12.0/rest-api/index.html

3.6.3 Dynamic overview

The tool supports the standard Oauth2 flows, such as the Authorization Code Flow, the Implicit Flow, the Resource Owner Password Credentials Grant (Direct Access Grant) and the Client Credentials Grant. High-level interactions are depicted in Figure 10 and Figure 19, where the Identity/Authorization Management component interacts with the Data Catalogue and the URBANITE UI.

3.6.4 Design implications based on the tool choice

Keycloak is available in different versions[35]. Among such versions, the tool provides guides to deploy it as a native application over OpenJDK[36] or using a docker-based deployment[37].

[35] https://www.keycloak.org/getting-started

[36] https://www.keycloak.org/getting-started/getting-started-zip

[37] https://www.keycloak.org/getting-started/getting-started-docker

4 Conclusions

This document provides a detailed description of the entire global architecture of the URBANITE ecosystem and provides a general representation.

The document also shows a deeper analysis, both structural and behavioural, of each component of the architecture identifying interactions and dependencies among them.

The architecture depicted here is the base for the first integrated version for URBANITE Ecosystem, due to the end of June. Regarding the requirements defined in D5.1 [3] for this month 15th and the state of the research achieved and applied on the component's implementation. That version will be a part of this proposed schema, with the components needed to support a basic scenario. That basic scenario will be defined in parallel with the elements of the first integrated version.

The experience of making that basic scenario run in this first platform could lead to an evolution of the components as well as of the schema of the planned architecture. These changes will be reflected in future versions of the architecture and explained in the deliverables resulted from the different work packages as well as in the documents related to those future versions of the URBANITE platform.

This schema is the result of technical decisions taken by the partners during the initial discussions regarding technical aspects of the solution. The deployment of this platform will follow the DevOps methods and mechanisms described in the D5.6 deliverable [4], made within this same work package.

This document will be updated in subsequent versions in M27 and M33, reflecting the advances in the decisions and implementation of the URBANITE Key Results and components.

5 References

[1] URBANITE Consortium, «D5.1 Detailed requirements specification-v1,» 2020.

[2] URBANITE Consortium, «D2.2 Mapping of Stakeholders,» 2021.

[3] URBANITE Consortium, «D5.3 - Integration Strategies,» 2020.

[4] URBANITE Consortium, «D5.6 URBANITE DevOps Infrastructure,» 2021.