URBANITE

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

Deliverable D3.2

Data Harvesting Module and Connectors Implementation v1

Editor(s):	TEC, ENG, FhG
Responsible Partner:	Fraunhofer FOKUS
Status-Version:	Final – v1.0
Date:	30.09.2021
Distribution level (CO, PU):	PU

Project Number:	GA 870338
Project Title:	URBANITE

Title of Deliverable:	Data Harvesting Module and Connectors Implementation v1
Due Date of Delivery to the EC:	30.09.2021

Workpackage responsible for the Deliverable:	WP3 – Data Management Platform
Editor(s):	TEC, ENG, Fraunhofer FOKUS
Contributor(s):	TEC, ENG
Reviewer(s):	Alma Digit
Approved by:	All Partners
Recommended/mandatory readers:	WP4, WP5

Abstract:	This deliverable will have two versions and will present the software implementation of the data harvesting module accompanied with the design specification and documentation. This deliverable is the result of Task 3.1.
Keyword List:	Harvester, Data Management, Piveau, Pipe, Software
Licensing information:	This document 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
v0.1	16/07/2021	Draft ToC	FhG
v0.2	27/08/2021	First Draft	FhG
v0.3	08/09/2021	Second Draft	FhG, TEC
v0.4	29/09/2021	Suggestions by reviewers	FhG
v1.0	30/09/2021	Layouting	FhG

Table of Contents

List of Figures

List of Tables

Terms and abbreviations

API	Application Programming Interface
EC	European Commission
CC	Creative Commons
CSV	Comma Separated Values
DCAT	Data Catalogue Vocabulary
DCAT-AP	DCAT Application Profile
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation Linked Data
MIF/MID	MapInfo Interchange Format
NGSI	Next Generation Service Interface
NGSI-LD	Next Generation Service Interface Linked Data
REST	Representational State Transfer
RDW	Specific Open Data Portal of Amsterdam
SOAP	Simple Object Access Protocol
SPDP	Standard for Publishing Dynamic Parking Data
URL	Uniform Resource Locator
XML	eXtensible Markup Language
XSD	XML Schema Definition

Executive Summary

This deliverable contains an overview over the software components that are related to the tasks of data harvesting. This refers to the process of downloading data for further processing, albeit without making substantial changes to the data itself. While minor adjustments or filtering is part of this step, thorough data preparation, transformation and curation are covered in deliverable D3.5. Due to the heterogeneous nature of the data present in the URBANITE context the connector modules typically require specific tailoring to the respective methods of access. As such, the components that have been developed for performing this task are described in this deliverable.

As shown in deliverable D5.4 the Data Management Platform follows a microservice architecture. Of course, all components involved in the steps of fetching to storing of data and metadata must integrate into this architecture. In order to achieve this goal, the Piveau Pipe Concept is employed, a design approach aimed at high flexibility and loose coupling when orchestrating software services. The Piveau Pipe Concept is covered in detail in this deliverable, but also applies to the aforementioned components described in D3.5. One key service in this architecture is a dedicated scheduling component that is responsible for ensuring that data is fetched in regular intervals. For each existing module described in this deliverable an overview along with a description is given. Where applicable, details on configuration and usage are provided.

1 Introduction

The term Data Management Platform stands for a variety of distinct software components that work together to deliver the key functionalities that are data harvesting, data preparation/transformation/curation/anonymization, and data aggregation and storage. The three deliverables D3.2, D3.5, and D3.7 focus on these core features respectively. Due to the interaction between these modules the aforementioned deliverables should be understood as a collection of documents related to the same overarching concept that is the Data Management Platform.

1.1 About this deliverable

Within the Data Management Platform this deliverable focuses on the data harvesting and the software components involved in this task, i.e. connectors, importers, and the Scheduler. It presents the challenges involved in harvesting, the proposed solution, and their implementation. Also, it features a section that describes the Piveau Pipe concept, an architecture and software design that is used for implementing all harvesting related components. Developers can get started by reading the relevant sections on how to write Piveau pipe compliant modules.

1.2 Document structure

Section 2.1 covers the functionalities provided by the harvesting components as well as how they fit into the general URBANITE architecture. This is followed by a description of the Piveau Pipe concept, which is the overarching design into which the individual harvesting related components are integrated. These are listed in section 2.2.2, along with technical specifications and explanations on how to develop connectors configure the scheduler. Next, section 3 contains instructions on how to build, configure, and run the application(s). The document wraps up with a conclusion and references.

2 Implementation

2.1 Functional description

The harvesting modules and connectors need to provide a number of functionalities. First and foremost, they need to implement ways to import (i.e. download) data and metadata from endpoints on the web. These endpoints can come in all shapes and forms, for example simple public REST APIs, restricted SQL dumps, simple file downloads, or geodata streams. All these different kinds of data and metadata then need to be checked, cleaned, and harmonized for further processing, which is covered in D3.5 [1]. This is achieved by data preparation and subsequent transformation steps, as well as curation. Once the data and metadata are brought into a common format (i.e. FIWARE Smart Data Model [2]) they need to be stored in dedicated databases (covered in D3.7 [3]).

Additionally, the (meta-)data needs to be downloaded in regular intervals to account for changes thereof. Managing these intervals is the responsibility of the Scheduler. Unlike the other components described in this deliverable it does not download data itself, but triggers the other data importers, which in turn download the data.

In summary, this deliverable therefore covers harvesting and scheduling. For completeness' sake, the exporting component, which is responsible for pushing arbitrary data to the applicable API endpoints of the data storage, is also featured in this deliverable. Once harvested, data could be stored directly through the exporter into the database repositories if no preparation or data transformation is necessary, or it can be pushed to the next step of the pipeline for data quality checks and transformations. Note that neither the Scheduler nor exporting component are shown as dedicated modules in Figure 1.

The functional requirements for harvesting and scheduling were listed in deliverable D5.1 and a detailed design was provided in deliverable D5.4 [4]. Table 1 shows a short summary of the development status. All the requirements applicable to the data models and datasets that are covered in the first prototype have been fulfilled or partially fulfilled. More data models and pagination will be supported for the second version.

Table 1 Status of Harvester requirements from D5.1

Requirements in D5.1	Current Status
Data Harvesting from heterogeneous data sources	Partially fulfilled: more data sources need to be harvested for v2
Pagination	Not fulfilled: the data sources managed in v1 did not require pagination. This will be addressed for v2
Data Harvesting extensibility	Fulfilled: the pipeline design is flexible and extensible
Data Harvesting supported protocols	Fulfilled: although for the moment all harvesters use HTTP(S)
Scheduled data fetching	Fulfilled: Cron triggers can be set up for pipelines

2.1.1 Fitting into overall URBANITE Architecture

In general, the harvesting modules and connectors are part of the backend services of the URBANITE architecture. They are managed by the scheduling component mentioned in the previous section. Since all related components follow a microservice approach they fit well with the docker-based architecture designed in WP5. As such, they also scale well, which is considered a key requirement when frequently downloading potentially large amounts of data and preparing/transforming them. The components that are described in this deliverable are highlighted in green in the architecture diagram (Figure 1) from deliverable D5.4.

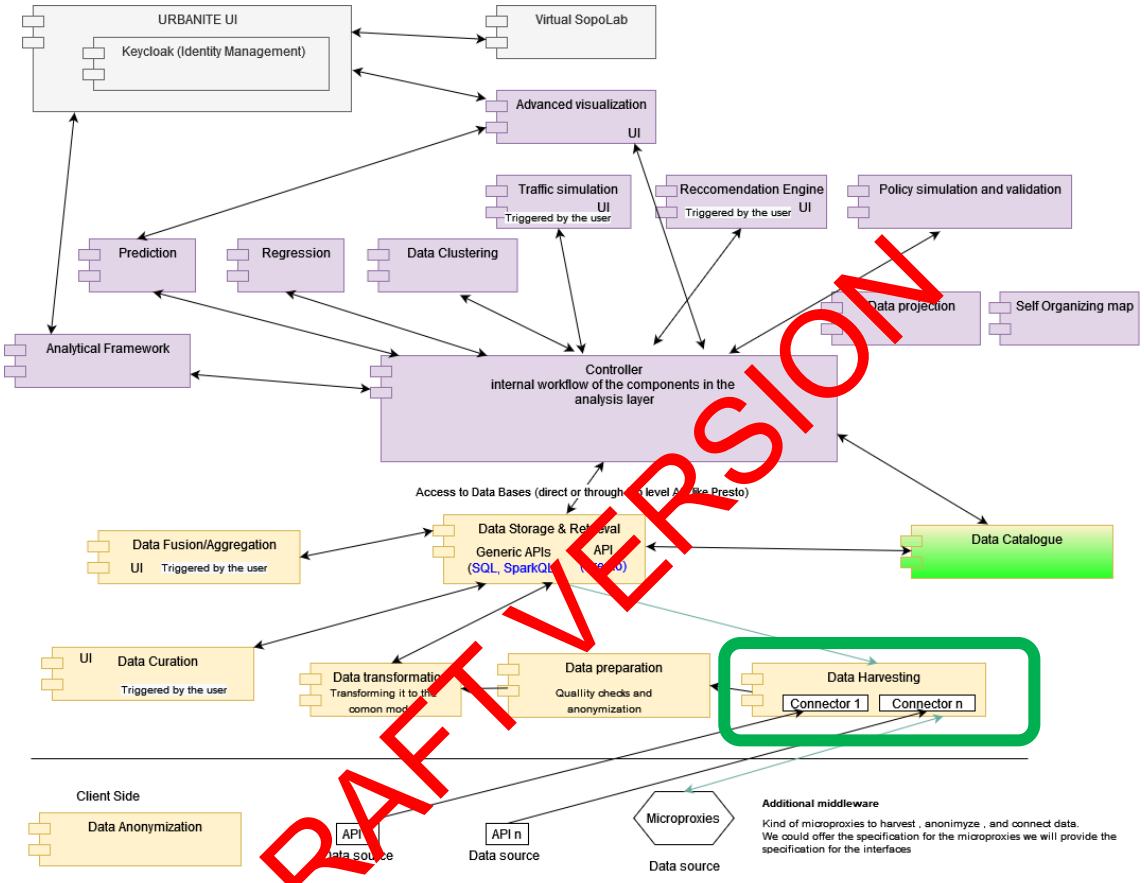


Figure 1: URBANITE Architecture

2.2 Technical description

This section describes the technical details of the implemented software. Data management is the process of fetching, anonymizing, preparing, transforming, storing, organizing and maintaining the data created and collected by an organization. Harvesting refers to the subset of steps from the import of data to the export into a data store. In URBANITE, this harvesting process has been implemented using a pipeline, i.e. a chain of processing components arranged so that the output of each component is the input of the next. The pipeline has been developed using the open source solution named Piveau Pipe Concept, which is explained in detail in section 2.2.1. This is followed by an overview of the components that have been developed thus far in section 2.2.2. Examples of how to write a compatible connector and how to schedule pipes are included.

2.2.1 Piveau Pipe Concept

The components involved in the steps from data fetching until storage are orchestrated by the Piveau Pipe concept [5] outlined in this section. A high-level overview of how components interact in this processing chain is shown in Figure 2.

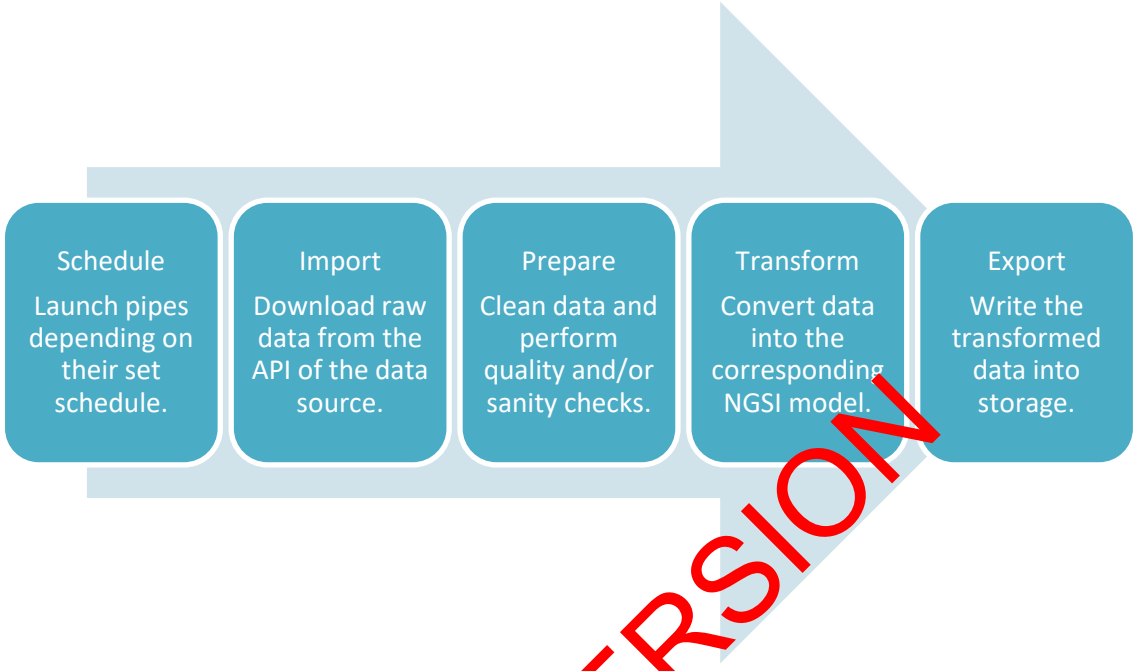
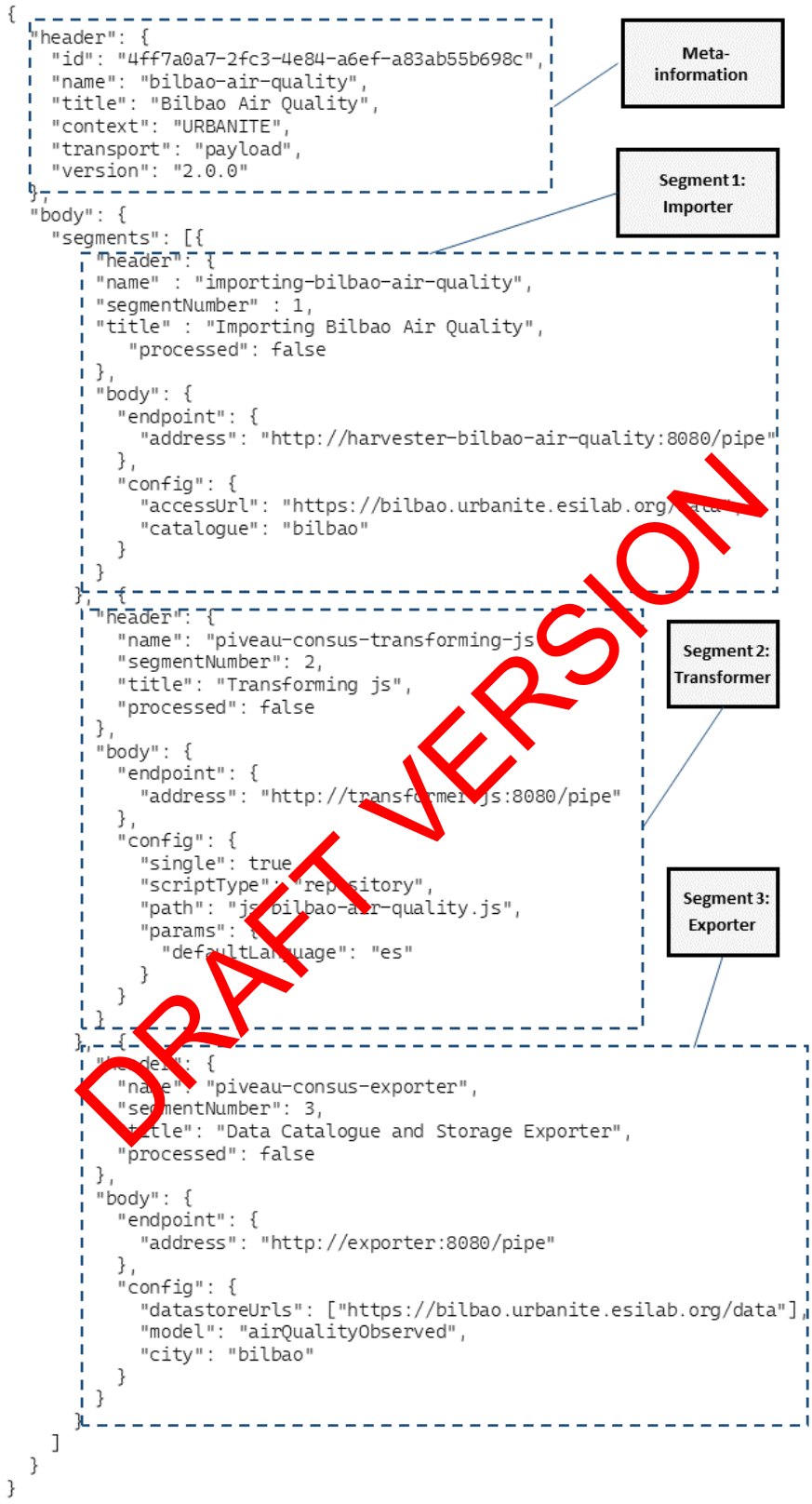


Figure 2: URBANITE data harvesting implemented using the Piveau Pipeline concept

On an architectural level, the Piveau Pipe allows the collection of data from heterogeneous data sources and the orchestration of a multitude of subsequent services. In order for a component to cohere to the Piveau Pipe concept, it needs to be developed as a web service that exposes a common RESTful interface, which is explained in detail in section 2.2.2.1. This means that the services can be connected in a generic fashion to implement specific data processing chains. No central instance is responsible for orchestrating the services. This is achieved by so-called pipe descriptors, a JSON file that contains a definition of components (endpoints, chronological order, specific configurations) that make up one processing sequence. Each processing chain is defined in one of these files (see Figure 3).

The Scheduler is the component responsible of managing and launching all the pipelines. To do so, the Scheduler either reads these files from disk or polls a Git repository to become aware of which pipes are available. These can then be assigned to a periodic trigger for recurring execution. When such a trigger fires a copy of the contents of the according pipe descriptor is sent to the first component in line, i.e. the one identified in the segment with segment number 1. During processing, the pipe descriptor is augmented. Data that needs to be passed along the processing chain is written into a payload field of the next component in line. For smaller amounts of data this can happen directly, for larger amounts of data a pointer to an external datastore can be used. Figure 3 shows an example of a pipe descriptor for downloading Bilbao air quality data, transforming it, and writing the transformed data to an instance of the data storage.




```
{
  "header": {
    "id": "4ff7a0a7-2fc3-4e84-a6ef-a83ab55b698c",
    "name": "bilbao-air-quality",
    "title": "Bilbao Air Quality",
    "context": "URBANITE",
    "transport": "payload",
    "version": "2.0.0"
  },
  "body": {
    "segments": [{
      "header": {
        "name" : "importing-bilbao-air-quality",
        "segmentNumber" : 1,
        "title" : "Importing Bilbao Air Quality",
          "processed": false
      },
      "body": {
        "endpoint": {
          "address": "http://harvester-bilbao-air-quality:8080/pipe"
        },
        "config": {
          "accessUrl": "https://bilbao.urbanite.esilab.org/
          "catalogue": "bilbao"
        }
      }
    }, {
      "header": {
        "name": "piveau-consus-transforming-js
        "segmentNumber": 2,
        "title": "Transforming js",
        "processed": false
      },
      "body": {
        "endpoint": {
          "address": "http://transformer-js:8080/pipe"
        },
        "config": {
          "single": true
          "scriptType": "repository",
          "path": "js/bilbao-air-quality.js",
          "params": {
            "defaultLanguage": "es"
          }
        }
      }
    }, {
      "header": {
        "name": "piveau-consus-exporter",
        "segmentNumber": 3,
        "title": "Data Catalogue and Storage Exporter",
        "processed": false
      },
      "body": {
        "endpoint": {
          "address": "http://exporter:8080/pipe"
        },
        "config": {
          "datastoreUrls": ["https://bilbao.urbanite.esilab.org/data"],
          "model": "airQualityObserved",
          "city": "bilbao"
        }
      }
    }
  ]
  }
}
```

Figure 3: Example of a Piveau Pipe Descriptor

As can be seen, each segment contains a header with metadata and a body with component-specific configurations, for example relevant URLs. As stated earlier, this resembles the pipe's descriptor. Whenever the Scheduler triggers a pipe, this descriptor is sent to the first component in line, in this case the `importing-bilbao-air-quality` module. Each dataset is immediately sent to the next component, in this case the `transforming-js` module. Once the datasets have been transformed to the desired output format, in this case the FIWARE `airQualityObserved` SmartData model, the result is sent to the Exporter, an adapter that is capable of uploading data and metadata to the data storage. The process of how the conversion of data between import and export is accomplished is explained in detail in deliverable D3.5 [1].

It is important to note that all this happens on a per-dataset basis, that is, the Importer does not wait for all datasets to download and send the payload in bulk, but each dataset is handled individually. This ensures flexibility, as no component needs to keep state or track how much data has been received. Also, each component can be scaled individually depending on the respective workloads.

The way this works is that payloads are injected into the descriptor as it is passed along the pipeline. The descriptor is therefore not a static, immutable object, but changes over time. This also makes debugging easier, as the evolution of the payload can be tracked. The Scheduler does not provide a payload, so the descriptor is sent as is. The Importer, however, creates a copy of the descriptor for each dataset it downloads and injects said dataset into this copy. Of course, each component is also capable of extracting their respective payloads. The Transformer works in the same way; it takes the payload sent by the Importer, processes it, and injects the result as a payload for the Exporter.

2.2.2 Components overview

At time of writing the following harvesting related components shown in Table 2 have been developed. All of these components are suitable for continuous fetching of data in regular intervals. In contrast, historic data or public holidays/calendar data by nature aren't prone to frequent changes and/or updates. For simplicities sake, this kind of data has been loaded into the data storage using Java code that is not included in the Piveau pipeline. However, using the file importer this could have been accomplished in coherence with the Piveau pipeline.

Table 2: Component Overview

Type	Name	Description
Importer/ Connector	OpenWeatherMap	Downloads weather data from the OpenWeatherMap provider. Requires an account with a valid API key.
	OpenStreetMap	Downloads data from OpenStreetMap. The query must be configured into the respective pipe descriptor.
	File	Generic importer for downloading files from URLs. The file is Base64 encoded prior to forwarding. For larger files, a mechanism utilizing the filesystem for storage and only passing pointer to this data will be implemented. The importer ships with a Python script that can be used to spin up a simple webserver that serves a file from local storage over HTTP.
	Bilbao Air Quality	Downloads weather data from a regional provider.
	Bilbao Traffic Flow	Download traffic data from a regional provider.
	Helsinki Traffic Flow	Download traffic data from a regional provider.
Misc.	Scheduler	Keeps track of existing pipe descriptors and manages triggers. The former are polled from a Git repository, the latter can be created/updated/deleted via a REST API. The service exposes a shell accessible by HTTP or Telnet that allows basic interaction like viewing existing pipe descriptors and launching them manually.
	Data Storage Exporter	Pushes incoming data and metadata to the data storage. Allows the specification of multiple storages, which is required for data that is relevant to multiple environments.
	Historic data wrappers	Manage the files with historical data (air quality, traffic) provided by the cities.
Library	Piveau Pipe Model	Container for storing information encoded in a pipe descriptor. Offers a selection of related methods, like (de-) serializing and setting certain fields.
	Piveau Pipe Connector	Handles communication between pipe components. Should be used when implementing pipe compliant services.
	Piveau Pipe Launcher	Is used by the Scheduler for initiating execution of existing pipes.

2.2.2.1 Writing an importer/connector

In this section, we provide a detailed explanation on how to create an importer/connector to easily integrate it into the pipeline. The connector or importer in segment 1 will download the data, make a first validation of it, create the necessary metadata and redirect to the next process in the pipeline. Harvesting related components are based on Vert.X[1] framework. Therefore, the connector/importer should be an instance of the Verticle class.

```
public class ImportingBilbaoAirQualityVerticle extends AbstractVerticle
```

Figure 4. Importer for air quality data in Bilbao

When this Verticle is launched for the first time, the handler for processing pipe messages is registered with the eventbus. The MainVerticle, which is responsible for spawning the Pipe API

[1] https://vertx.io/

and content-negotiation, send incoming requests along the eventbus. This is then consumed by the aforementioned handler. This is shown in Figure 5.

```
@Override
public void start(Promise<Void> startPromise) {
    vertx.eventBus().consumer("bilbao-importer", message -> {
        PipeContext pipeContext = message.body();
        pipeContext.log().info("Import started");
        accessUrl = pipeContext.getConfig().getString("accessUrl");

        ...
    });

    ...
}
```

Figure 5: Registering a Pipe handler with the Vert.X Eventbus

In the example in Figure 5, `accessUrl` is a parameter included in the pipe descriptor, as shown in Figure 6.

```
"config": {
  "accessUrl": "https://urbanite.esilab.org:8443/data",
  "catalogue": "helsinki"
}
```

Figure 6: Pipe descriptor with accessUrl

Thanks to the pipe descriptor, the same importer can be used to harvest different data sources without modifying the component's code. To download the data, a web client can be used if the data source is available through HTTP, e.g. in the case of a web service API or a URL.

```
webClient.getAbs("https://bilbao.urbanite.esilab.org/data")
        .addQueryParam("R01HNoPortal", "true")
        .addQueryParam("tipoICa", "2")
        .addQueryParam("idContaminante", "8")
        .putHeader("Accept", "application/json")
        .expect(ResponsePredicate.SC_OK)
        .send()
```

Figure 7: Web client to download air quality data from Bilbao's air quality service

Once the data is downloaded, not all of it is forwarded to the next process in the pipeline. For example, the Basque Country's air quality service (see Figure 8) returns data about all the meteorological stations in the entire province. However, in URBANITE, we are only interested in the information coming from the stations in the municipality of Bilbao. The rest of the data is discarded. In addition, the metadata is created for the downloaded data and forwarded in the `metadata` field. The helper classes for constructing DCAT-AP metadata are included in the Piveau libraries.

```
DCATAPGraph dcatapGraph = new DCATAPGraph();
Dataset dataset = dcatapGraph.createDataset("sample_dataset")
    .setTitle("Bilbao Air Quality")
    .setDescription("Air Quality information for Bilbao")
    .addKeyword("Bilbao")
    .addKeyword("Air Quality")
    .setIssued(Instant.now())
    .setModified(Instant.now())
    .setAccessRights("public")
    .setTheme("http://publications.europa.eu/resource/authority/data-theme/REGI")
    .setTheme("http://publications.europa.eu/resource/authority/data-theme/TRAN")
    .setPublisher("URBANITE", "https://urbanite-project.eu/");

dataset.createDistribution("sample_distribution")
    .setAccessURL(accessUrl + "/getTDataRange/airQualityObserved/bilbao)
    .setFormat("http://publications.europa.eu/resource/authority/file-type/JSON")
    .setLicense("http://publications.europa.eu/resource/authority/licence/CC_BY")
    .setDescription("Air Quality information for Bilbao")
    .setTitle("Bilbao Air Quality)
    .build();
```

Figure 8: Creation of the metadata for the downloaded dataset and distribution

Finally, both the data and metadata are forwarded to the next process in the pipeline (see Figure 9). For all interaction between the Piveau Pipe services the `piveau-pipe-connector`[2] library should be used. It provides an abstraction from the inner workings and communication protocols implemented. It also parses the incoming pipe descriptors and extracts the applicable segment info for a given service.

```
// pass dataset to next pipe module
private void forwardDataset(JsonObject dataset, PipeContext pipeContext, String identifier) {
    ObjectNode dataInfo = new ObjectMapper().createObjectNode()
            .put("identifier", identifier)
            .put("catalogue", pipeContext.getConfig().getString("catalogue"));

    pipeContext.log().info("Importer result:\n{}", dataset.encodePrettily());
    pipeContext.setResult(dataset.encodePrettily(), "application/json", dataInfo).forward();
}
```

Figure 9: Forwarding both the data and the metadata to the next process in the pipeline.

[2] https://github.com/piveau-data/piveau-pipe-connector

2.2.2.2 Scheduling the data fetching

Depending on the data source, the update frequency changes. For example, traffic flow data is updated every 5 minutes whereas air quality data is updated every hour. For this reason, each pipeline needs to be triggered with a different frequency. As explained before, the Scheduler is the component responsible of managing these triggers. To configure the Scheduler, triggers can be set using the provided REST API. It supports one-time ("immediate") and Cron[3] triggers. For example, to trigger the harvesting pipeline for the air quality data every hour we would send the request shown in Figure 10 via PUT method to `triggers/bilbao-air-quality`. Note the `pipeId` in the payload (`bilbao-air-quality`).

2.2.3 Technical specifications

```
[
  {
    "id": "BilbaoAirQuality",
    "status": "enabled",
    "cron": "0 0 0/1 ? * * *",
    "next": "2021-07-23T10:45:00Z"
  }
]
```

Figure 10: Triggering the harvesting pipeline every hour

All harvesting related components are written in Java and are based on the Vert.X[4] framework developed by the Eclipse Foundation. Vert.X proposes and supports an asynchronous programming paradigm which aims to improve performance and responsiveness by ensuring that a thread is never blocked by long-running tasks. The basis of this is the Netty[5] project.

The pipe functionality (parsing and manipulating the pipe descriptor) is provided by the Piveau Pipe Model library. The common endpoint each component exposes is implemented by the Piveau Pipe Connector library.

All pipe components except the Scheduler are stateless. As such, only the Scheduler requires a database and for this purpose relies on the embedded version of the Open-Source relational database H2[6], accessible via JDBC. The component uses the database to store pipeline triggers.

With all services being JVM based the software stack runs on any machine that is supported by the JVM. Depending on the number of instances running and the kind of data that is processed a sufficient amount of memory should be available.

[3] http://www.nncron.ru/help/EN/working/cron-format.htm
[4] https://vertx.io/
[5] https://netty.io/
[6] https://h2database.com

3 Delivery and usage

3.1 Package information

All components are Java applications that are built using Maven[7]. As such they cohere to the default standardized folder structure for source files (`/src/main/java/io/piveau/{componentName}`) and resource files (`/src/main/resources`). The latter contains files like the OpenAPI specification (`webroot/openapi.yaml`) when applicable and logging configuration (`logback.xml`).

3.2 Installation instructions

In order to integrate well into the URBANITE platform all components are available as Docker images. However, before building the Docker images the corresponding JAR file needs to be created. A JAR file is an executable that run runs on the JVM. The harvesting components rely on a build tool called Maven for dependency management and generation of the JARs. As such, the deployment of a service can be achieved using the three commands below. Note that curly brackets indicate that applicable values need to be substituted.

```
$> mvn clean package

$> docker build -t urbanite/{component-name} .

$> docker run -p {PORT}:8080 urbanite/{component-name}
```

Depending on the respective component a certain configuration may need to be applied, for example an API key. This can be achieved using environment variables, which can be passed to Docker containers like so:

```
$> docker run -e {ENV_VAR}={value} urbanite/{component-name}
```

3.3 User Manual

In general, each component provides a human-readable form of its OpenAPI specification at `{hostname:port}/index.html`. The corresponding file is stored at `src/main/resources/webroot/openapi.yaml`. However, this does not apply to those components that spawn their endpoints based on a library. This is specifically the case for all pipe components that rely on the Piveau Pipe Connector library. These expose the common endpoint at `{hostname:port}/pipe`, which accepts compliant pipe descriptors via the HTTP POST method.

Aside from this most of the components require very little configuration and work out of the box. The specific environment variables that need to be set for each service are listed in the respective `README.md` file in the root directory.

[7] https://maven.apache.org/

3.3.1 Scheduler

A special case however is the Scheduler, which requires a little more setup and also exposes more endpoints than the other pipe components. As described previously the Scheduler serves two main purposes: keeping track of existing pipe descriptors and managing triggers for these pipes. In order to fulfil the former task, the pipe descriptors can either be copied to the `src/main/resources/pipes` directory before compilation. Alternatively, the descriptors can be managed using a GitLab repository. For this a so-called cluster-config akin to the snippet shown in Figure 11 must be set.

```
PIVEAU_CLUSTER_CONFIG:

{
   "pipeRepositories": {
     "system": {
       "uri":
"https://gitlab.com/urbanite/harvesting-
pipes.git",
       "username": "gitlab-user",
       "token": "gitlab-token”,
       "branch": "master"
     }
  }
}
```

Figure 11: PIVEAU_CLUSTER_CONFIG variable

The Scheduler frequently polls this repository, thereby detecting changes to the pipe descriptors at runtime. In order to monitor pipe descriptors registered with the Scheduler, view their contents or launch them manually the component exposes a shell, accessible either via HTTP or Telnet. To enable this a shell config like the one in Figure 12 must be set.

```
PIVEAU_SHELL_CONFIG:

{
   "http": {
     "host": "0.0.0.0",
     "port": 8085
   }, "telnet": {
       "host": "0.0.0.0",
       "port": 5000
   }
}
```

Figure 12: PIVEAU_SHELL_CONFIG variable

This exposes HTTP access at `{hostname}:8085/shell.html` and Telnet access on port 5000. The available shell commands are shown in Table 3.

Table 3: Scheduler Shell Commands

Command	Description
pipes	List available pipes.
show {pipeId}	View contents specific pipe descriptor.
trigger {pipeId}	List triggers of specific pipe.
launch {pipeId}	Start specific pipe immediately.

Management of triggers is made possible via an exposed RESTful API. The available paths and corresponding methods are listed in Table 4.

Table 4: Scheduler API

Path	Method	Description
/triggers	GET	Get a list of pipe IDs and scheduled triggers.
	PUT	Bulk update of all triggers.
/triggers/{pipeId}	GET	Returns all triggers for the pipe with the specified pipeId.
	PUT	Create or update triggers for pipe with pipeId.
	DELETE	Delete previously created triggers.

3.4 Licensing information

The license terms for the software are under discussion among the consortium. AGPLv2 and AGPLv3[8] are being considered.

3.5 Download

All source code resides in the GitLab maintained by Tecnalia[9]. There, pilot specific components (i.e. data source adapters) are grouped in dedicated subgroups. Generic harvesters and components resign in the root.

[8] https://www.gnu.org/licenses/agpl-3.0.en.html
[9] https://git.code.tecnalia.com/urbanite/private/wp3-data-management/harvester

4 Conclusions

Overall, this document describes the technical details of the components involved in the harvesting process. This includes the custom adapters for data sources, both generic and pilot specific as well as common components like the Scheduler. It is shown how these modules integrate into the general URBANITE data management platform architecture and the Piveau Pipe concept. The latter describes a mechanism of loose component coupling by standardising exposed APIs, thereby fostering the reuse of existing services. For developers the deliverable contains instructions on how to develop Piveau pipe compliant services.

Additionally, noteworthy components like the Scheduler are described in detail with respect to implementation and configuration. A more general rundown of the other components is also provided. In conclusion this deliverable allows the reader to get an understanding of the technical solution(s) employed for the continuous harvesting of data sources.

5 References

[1] FhG, TEC and ENG, "Data curation module implementation-v1," 2021.

[2] TEC, C. Messina, ENG, BIL, MLC and FhG, "URBANITE Mobility Data Sources Analysis," European Commission, 2020.

[3] FhG, TEC and ENG, "Data aggregation and storage module implementation-v1," 2021.

[4] TEC, FhG, ENG and JSI, "URBANITE architecture," 2021.

[5] F. Kirstein, K. Stefanidis, B. Dittwald, S. Dutkowski, S. Urbanek and M. Hauswirth, "Piveau: A Large-Scale Open Data Management Platform Based on Semantic Web Technologies," 2020.

[6] FhG, TEC and ENG, "Data harvesting module and connectors implementation-v1," 2021.

[7] FhG, TEC and ENG, "URBANITE data structure and semantic model specification," 2020.

[8] TEC, FhG, ENG and JSI, "Detailed requirements specification," 2020.