

D6.5 – APIs and documentation for software extension

Deliverable No.	D6.5	Due Date	30/06/2020
Type	Other	Dissemination Level	Public
Version	1.0	Status	Final
Description	Technical specification of developed methods and services. Both API and other software-related documentation will be in a standardised format permitting particular deployments to access the software adequately.		
Work Package	WP6		

Authors

Name	Partner	e-mail
Benjamín Molina	P01 – UPV	benmomo@upvnet.upv.es
Ignacio Lacalle	P01 – UPV	iglaub@upv.es
Carlos E. Palau	P01 – UPV	cpalau@com.upv.es
José A. Clemente	P02 – PRO	jclemente@prodevelop.es
Ismael Torres	P02 – PRO	itorres@prodevelop.es
Flavio Fuart	P03 XLAB	flavio.fuart@xlab.si
Damjan Murn	P03 XLAB	damjan.murn@xlab.si
Dejan Štepec	P03 XLAB	dejan.stepec@xlab.si
Tomaž Martinčič	P03 XLAB	tomaz.martincic@xlab.si
Marc Despland	P06 ORANGE	marc.despland@orange.com

History

Date	Version	Change
10-April-2020	0.1	ToC and task assignment
22- April-2020	0.2	OT contribution
02-June-2020	0.21	Consolidated OT contribution
15-June-2020	0.5	Contributions for DAL, IH, D&N and Security. Internal review
24-June-2020	0.6	Final corrections after internal review
30-June-2020	1.0	Final release

Key Data

Keywords	API, Interface, development, software extension
Lead Editor	Benjamín Molina – P01 UPV
Internal Reviewer(s)	Gilda De Marco – P04 INSIEL Paolo Casoto – P04 INSIEL Luka Traven – P08 MEDRI

Abstract

PIXEL Enabling ICT Infrastructure framework is one of the key outcomes of PIXEL activities. The main goal is to compose a complete data-centric port solution, allowing data-level interoperability of different systems, including legacy industrial and port operations systems.

This framework provides sound technological foundations for efficient and cost-effective execution of models, simulations and predictions that are part of the PIXEL environmental impacts assessment model, to be used by ports of the future for efficient management and tackling environmental issues. The framework will also integrate supporting ICT tools for the calculation of the Port Environmental Index (PEI), as a key parameter to improve operations in the ports of the future.

The most important asset of this deliverable is the provision of APIs in the different main building blocks of the final PIXEL architecture as defined in deliverable D6.2. The main components are:

- The **PIXEL Data Acquisition Layer**, with a standard **NGSI interface** able to connect various types of sensors to a common Broker. This Broker is also able to support subscription modes acting as a means of exporting (notifying) data in real time.
- The **PIXEL Information Hub**, using the subscription mode to collect and store data from the Data Acquisition Layer and offering a dual interface for data search and retrieval. On the one hand, there is **REST API** to retrieve data that has been imported from **sensor data**. On the other hand, it also supports access to the **REST API** exposed by the core storage open source component: **Elasticsearch**. The dual support offers the bridge to port stakeholders to benefit for ELK (Elasticsearch, Logstash, Kibana) if they are currently using them in their internal structures, facilitating the integration of different platforms.
- The **PIXEL Operational Tools** are divided into two subcomponents. The first one related to the main manager, which provides a **REST (Swaggerized) API** to manage models, predictive algorithms, executions (instances) and scheduled executions. Furthermore, the API also allows managing KPIs and linking them to event processing based on ElastAlert. The second component is particularized for every model or predictive algorithm to be properly integrated in the PIXEL platform, so that they expose a common API that simplifies the management.
- The **PIXEL Dashboard** does not have a proper interface as it represents the user interface; therefore, it includes connectors to manage all other components (Information Hub, Operational Tools and Security). However, the **Notification system** based on ElastAlert, as for the Operational Tools, includes a **REST API** for generating rules, rule templates and notification mechanisms (alerts).
- The **PIXEL Security** framework includes a set of **standard APIs** and interfaces to guarantee a trusted and secured interaction among components and users (authentication, authorization and accounting). It is based on FIWARE components such as **Wilma**, **Keyrock** and **AuthForce**.

In addition to APIs that define how to interact with the different components of the architecture, this deliverable also provides **documentation** for those subcomponents that are prone to be **extended by software developers**, so that they can better the initial scope, their limitations but, at the same time, the potential enhancements and how to achieve them in an efficient way according to specific port's needs.

Finally, documentation is an **ongoing work** that should be updated throughout the lifetime of the project. Even if this deliverable is scheduled for M26 and not the end of the project, any documentation update will be reflected in the specific website of the project, placed at <https://pixel-ports.readthedocs.io/en/latest/>. This serves as **main entry point for the PIXEL documentation**, covering updates both on deliverables D6.4 and D6.5.

Statement of originality

This document contains material, which is the copyright of certain PIXEL consortium parties, and may not be reproduced or copied without permission. This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

The information contained in this document is the proprietary confidential information of the PIXEL consortium (including the Commission Services) and may not be disclosed except in accordance with the consortium agreement.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the project consortium as a whole nor a certain party of the consortium warrant that the information contained in this document is capable of use, nor that use of the information is free from risk, and accepts no liability for loss or damage suffered by any person using this information.

The information in this document is subject to change without notice.

The content of this report reflects only the authors' view. The Innovation and Networks Executive Agency (INEA) is not responsible for any use that may be made of the information it contains.

Table of contents

Table of contents	5
List of tables	7
List of figures	8
List of acronyms	10
1. About this document.....	12
1.1. Deliverable context	12
1.2. The rationale behind the structure.....	14
2. Introduction	15
3. PIXEL Data Acquisition.....	16
3.1. Overview.....	16
3.2. NGSi Agents	17
3.2.1. Labels for all agents.....	17
3.2.2. Labels for daemon agents	17
3.2.3. Labels for scheduled agents.....	18
3.2.4. Examples.....	18
3.3. DAL Orchestrator	19
3.3.1. Paths.....	19
3.3.2. Models	19
3.3.3. Quick start guide.....	20
3.4. Developer’s guide	22
3.4.1. Managing Data Format with Information Hub	22
3.4.2. Additional notes.....	23
4. PIXEL Information Hub.....	24
4.1. Overview.....	24
4.2. REST API	26
4.2.1. Paths.....	27
4.2.2. Models	33
4.3. Developer’s guide	34
4.3.1. Building from sources.....	35
4.3.2. Development environment.....	36
4.3.3. Potential extensions	38
5. PIXEL Operational Tools.....	44
5.1. Introduction.....	44
5.1.1. Main concepts and architecture	44
5.1.2. Models	45
5.1.3. Key Performance Indicators	46
5.1.4. Event processing.....	47

5.2.	Developer’s guide	48
5.2.1.	Identification of interfaces	48
5.2.2.	Management interface.....	49
5.2.3.	Execution interface	52
5.2.4.	Software Extensions	54
5.2.5.	Compilation from the sources.....	57
6.	PIXEL Dashboard and Notification	61
6.1.	Overview.....	61
6.2.	Developer’s guide	62
6.2.1.	Introduction.....	62
6.2.2.	Folder structure (Client solution).....	63
6.2.3.	Add new views.....	63
6.2.4.	Internationalization	64
6.2.5.	Notifications	66
6.2.6.	Access to APIs	67
6.2.7.	Add a new entity to the server solution	70
6.2.8.	Add new visualizations	74
7.	PIXEL Security	75
7.1.	Overview.....	75
7.2.	(REST) API.....	76
7.2.1.	Paths.....	76
7.2.2.	Models	76
7.3.	Developer’s guide	77
7.3.1.	Potential extensions	77
7.3.2.	Additional notes.....	77

List of tables

Table 1. Deliverable context.....	12
Table 2. Information Hub services and corresponding Main classes	37
Table 3. Logging format example for model execution (error).....	54
Table 4. Logging format example for model execution (success).....	54

List of figures

Figure 1. Interaction scheme DAL- Dashboard with security support.....	16
Figure 2. Swagger UI for the DAL Orchestrator.....	19
Figure 3. Model schemas for the DAL Orchestrator API.....	20
Figure 4. Information Hub - Architecture overview.....	24
Figure 5. List of submodules.....	35
Figure 6. Build process.....	36
Figure 7. Docker images after building the IH.....	36
Figure 8. Zookeeper monitoring.....	39
Figure 9. Information Hub Management (TideSensorObserved example).....	39
Figure 10. Orion entity (TideSensorObserved example).....	40
Figure 11. Orion entity conversion by the Collector (TideSensorObserved example).....	40
Figure 12. Nodes structure (Orion Data Collector).....	42
Figure 13. Dockerfile for Orion Data Collector.....	43
Figure 14. Operational Tools - Architecture overview.....	44
Figure 15. Operational Tools - Functional overview.....	45
Figure 16. Link between models and the Operational Tools.....	46
Figure 17. Key Performance Indicators.....	47
Figure 18. Operational Tools- Event Processing overview.....	47
Figure 19. Operational Tools- Identification of interfaces.....	48
Figure 20. Operational Tools – Link interfaces and internal components.....	49
Figure 21. Operational Tools – Management Swagger API.....	50
Figure 22. Operational Tools- Management APIs with code samples.....	51
Figure 23. Operational Tools- Management API ported to Apiary.....	52
Figure 24. Operational Tools- Execution interface overview.....	53
Figure 25: Dashboard diagram.....	61
Figure 26: Dashboard menu options.....	62
Figure 27: Dashboard features (client solution).....	62
Figure 28: Folder structure.....	63
Figure 29: Example of views inside its container folder.....	64
Figure 30: Example of how to fulfil a menu entry in index.js file.....	64
Figure 31: Files used to translate PIXEL (called attending to their ISO Language Codes).....	65
Figure 32: English translations for alerts functionality.....	65
Figure 33: Syntax, HTML Code.....	65
Figure 34: Syntax, JavaScript Code.....	66
Figure 35: Message syntax.....	66
Figure 36: Message appearance.....	66
Figure 37: Notification Syntax.....	66
Figure 38: Notification appearance.....	66
Figure 39: Content of the API folder.....	67
Figure 40: OTools endpoints.....	68
Figure 41: dataextractor API endpoints.....	68
Figure 42: Elasticsearch endpoints.....	69
Figure 43: Endpoints for resource entity.....	69
Figure 44: Request class for Operational Tools endpoints.....	70
Figure 45: Structure of server solution.....	71
Figure 46: Controller file.....	71
Figure 47: Service file.....	72
Figure 48: Example of model.....	72
Figure 49: Section for the import of the controller's entity.....	73
Figure 50: Section responsible for the routing of the controllers.....	73
Figure 51. . Operational Tools - Architecture overview.....	75

Figure 52. PIXEL security scheme.....	75
Figure 53. Related models for managing permissions	77

List of acronyms

Acronym	Explanation
AIS	Automatic Identification System
API	Application Programming Interface
ARPA	Agenzia regionale per la protezione ambientale
CEP	Complex Event Processing
CRUD	Create Read Update Delete (common functions in REST APIs)
DAL	Data Acquisition Layer
DoA	Description of Action
FAIR	Facility for Antiproton and Ion Research
GIS	Geographic information system
GPMB	Grand port maritime de Bordeaux
GUI	Graphic User Interface
i18n	Internationalization (support for multiple languages)
ICT	Information and communications technology
IdM	Identity Management
IH	Information Hub
IT	Information Technology
KPI	Key Performance Indicator
LDAP	Lightweight Directory Access Protocol
LTS	Long Term Storage
MVC	Model View Controller (software design pattern common for UIs)
NGSI	Name of the FIWARE API
NIFI	Apache NiFi is a software project designed to automate the flow of data between software systems
NMEA	National Marine Electronics Association
OS	Operating System
PAP	Policy Administration Point
PCS	Port Community System
PDP	Policy Decision Point
PEI	Port Environmental Index
PEP	Policy Enforcement Point
PIP	Policy Information Point
WP	Work Package
PMS	Port Management System

PMIS	Port Management Information System
REST	Representational state transfer
SILI	Sistema Informativo Logistico Integrato
SotA	State-of-the-Art
UI	User Interface
URL	Universal Resource Locator
Vue.js	The Progressive Javascript Framework (similar to React and Angular)
XACML	eXtensible Access Control Markup Language

1. About this document

This deliverable describes and provides the work related to APIs that has been done in the technical tasks of WP6, Enabling ICT (Information and communications technology) infrastructure framework. The deliverable consists of:

- A more descriptive part of the document, where developed software APIs are described for each of the main components of the architecture. Inputs, outputs and return values (including errors) are given for every endpoint to understand the whole functionality of each component. This should be the core part and the main interest for typical developers.
- A more analytical part of this document, though also descriptive, showing the different parts of the source code that could be potentially extended to meet specific port needs, so that external developers can adapt and enrich the (source) code.

This document is a final deliverable, no further iteration is envisioned; however, any update during the project will be reflected in PIXEL's official documentation site: <https://pixel-ports.readthedocs.io/en/latest/>.

1.1. Deliverable context

Table 1. Deliverable context

Keywords	Description
Objectives	<p><i>Objective 1: Enable the IoT-based connection of port resources, transport agents and city sensor networks</i></p> <p>This deliverable provides the documentation of the APIs of the software modules for IoT enablement and interconnection of different data providers to be integrated in the architecture. The API already supports data collection from generic sources. The document is complimentary with D6.4 to provide the source code and full documentation at WP6 level, whereas the integration and deployment itself will be finalised in WP7.</p> <p><i>Objective 2: Achieve an automatic aggregation, homogenization and semantic annotation of multi-source heterogeneous data from different internal and external actors</i></p> <p>D6.5 provides the different APIs of the different modules and submodules, supporting the exchange of data models and data formats that facilitate the aggregation and homogenization of the different modules of the architecture, mainly Information Hub and Data Acquisition Layer.</p> <p><i>Objective 3: Develop an operational management dashboard to enable a quicker, more accurate and in-depth knowledge of port operations</i></p> <p>The dashboard is provided in D6.4. However, this software makes use of the different APIs of the other modules (e.g. Operational Tools and Information Hub) described in D6.5</p>

	<p><i>Objective 4: Model and simulate port-operations processes for automated optimisation</i></p> <p>D6.4 provides the operational tools prototype resulting from T6.5. These tools give high-level technological support for the configuration and execution of models developed in WP4. The API specified in D6.5 allows to first describe models and execute or schedule them.</p> <p><i>Objective 5: Develop predictive algorithms</i></p> <p>D6.4 provides the operational tools prototype resulting from T6.5. These tools give high-level technological support for the configuration and execution of predictive algorithms developed in WP4. The API specified in D6.5 allows to first describe predictive algorithms and execute or schedule them. Note that from the Operational Tools perspective the API was specified in the same way for models and predictive algorithms to smooth the alignment.</p>
Exploitable results	D6.5 provides the API documentation of the following exploitable assets: PIXEL Data Acquisition; PIXEL Information Hub; PIXEL Operational Tools; PIXEL Integrated Dashboard and Notification; PIXEL Security and Privacy. It is complimentary to D6.4 for software developers who intend to understand the full potential of the components, or even modify or extend them with additional features.
Work plan	This deliverable is the result of work performed from M7 to M26 on tasks T6.2 – PIXEL Data Acquisition, T6.3 - PIXEL Information Hub, Task 6.4 - PIXEL Operational Tools, T6.5 - PIXEL Integrated Dashboard and Notification, Task 6.6 – PIXEL Security and Privacy.
Milestones	This deliverable, together with D6.4, serves as verification of the MS7 ICT solution developed (M26).
Deliverables	This deliverable follows the system architecture defined in D6.2 PIXEL Information system architecture and design v2, describing the different APIs of each of the modules encompassing the architecture. This deliverable is the only one related to API documentation. Any update during the project will be reflected in PIXEL’s official documentation site: https://pixel-ports.readthedocs.io/en/latest/
Risks	<p><i>WT5#6 Technical activities are not completed on time, are not aligned with the main objective, are not accurate or present a lack of consistency.</i></p> <p>This deliverable shows that technical activities related to T6.2 – T6.6 have been executed in a timely fashion in accordance to the architecture proposed in D6.2.</p> <p><i>WT5#14 Due to harshly divergences between formats of output/input data of ICT systems to integrate, the development can be delayed or paralyzed, and some extra effort will be needed to carry out the project.</i></p> <p>Particular attention is being devoted to the analysis and definition of data models in WP6. While the generic principles have been provided in D6.2, this deliverable provides a more detailed list of data entities identified in PIXEL. This mainly relates to FIWARE data models as well as models a predictive algorithms description formats.</p>

	<p><i>WT5#15 IoT components have security vulnerabilities.</i></p> <p>This assessment is part of T6.6. The software is described in D6.4 whereas its API is documented in D6.5. This allows to provide an assessment of IoT security vulnerabilities in PIXEL use-cases.</p>
--	---

1.2. The rationale behind the structure

This document describes the work performed in T6.2 – T6.6 of PIXEL related to APIs documentation. It should be considered as a ‘joint pack’ with D6.4 in order to have a full documentation of the software produced in PIXEL, both at user’s and developer’s level. Both deliverables describe each of the modules identified in D6.2, focusing D6.5 mainly on APIs documentation. This report consists of the following sections:

1. **About this document:** Deliverable context in relation to the PIXEL DoA, work packages, tasks and other deliverables.
2. **Introduction:** Relation with PIXEL objectives, use cases and requirements.
3. **PIXEL Data Acquisition:** Identification of subcomponents and interfaces, with special focus on NGSI Agents and DAL orchestrator to correctly manage them via Docker containers.
4. **PIXEL Information Hub:** Identification of components, REST API, compilation from sources, associated Docker instances and hints for software extensions.
5. **PIXEL Operational Tools:** Identification of interfaces, mainly at management level, to publish, execute and schedule algorithms. Compilation from sources is also provided.
6. **PIXEL Integrated Dashboard and Notification:** Identification of subcomponents and interfaces, with special focus on the different files needed to be modified for configuration and extension based on a MVC development pattern.
7. **PIXEL Security and Privacy:** Identification of interfaces and description of the different FIWARE enablers able to support different aspects for authentication and authorization. They are open source tools with a wide community support.

2. Introduction

Developments of the PIXEL ICT Infrastructure Framework are driven by real needs of the ports involved in the project. Those needs and requirements are the key results of WP3. In order to keep development in line with overall PIXEL objectives, each technical deliverable provides an introductory section where the relation with objectives, use cases and requirements is defined.

The relation with PIXEL objectives, use cases and requirements have already been described in deliverables D6.3 and D6.4, therefore this document will be mainly focussed on APIs and potential extensions of the different architecture modules.

3. PIXEL Data Acquisition

3.1. Overview

The main purpose of the Data Acquisition Layer (DAL) is to interface the external data sources to the PIXEL Information Hub and to convert the original and heterogeneous data formats to PIXEL Data Models.

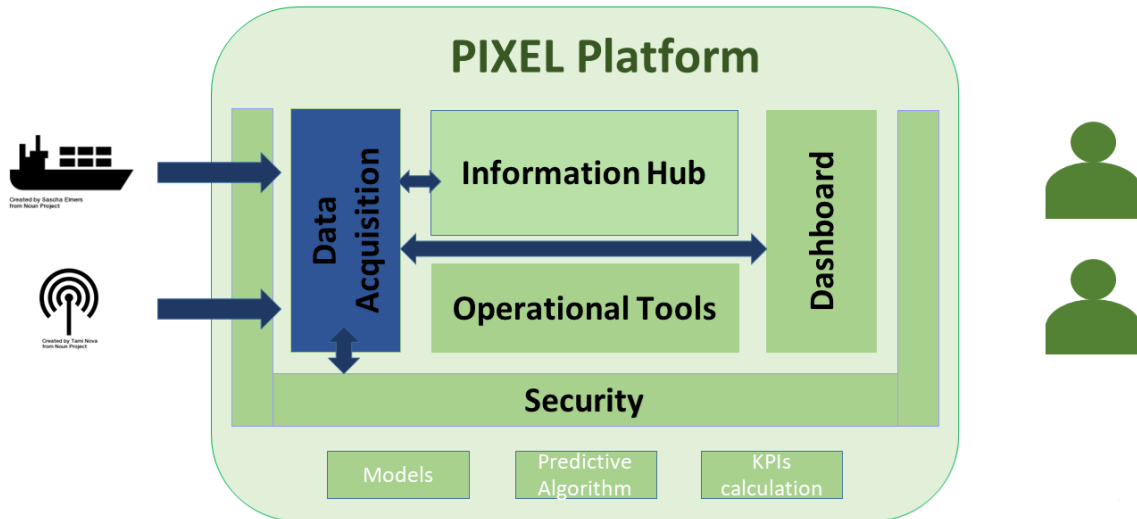


Fig. Data Acquisition Layer - Architecture overview

The Data Acquisition Layer exposes an API to the PIXEL Dashboard to allow the administrator to manage the different NGSI Agents, and it interacts with the PIXEL Security module to protect the NGSI Agents.

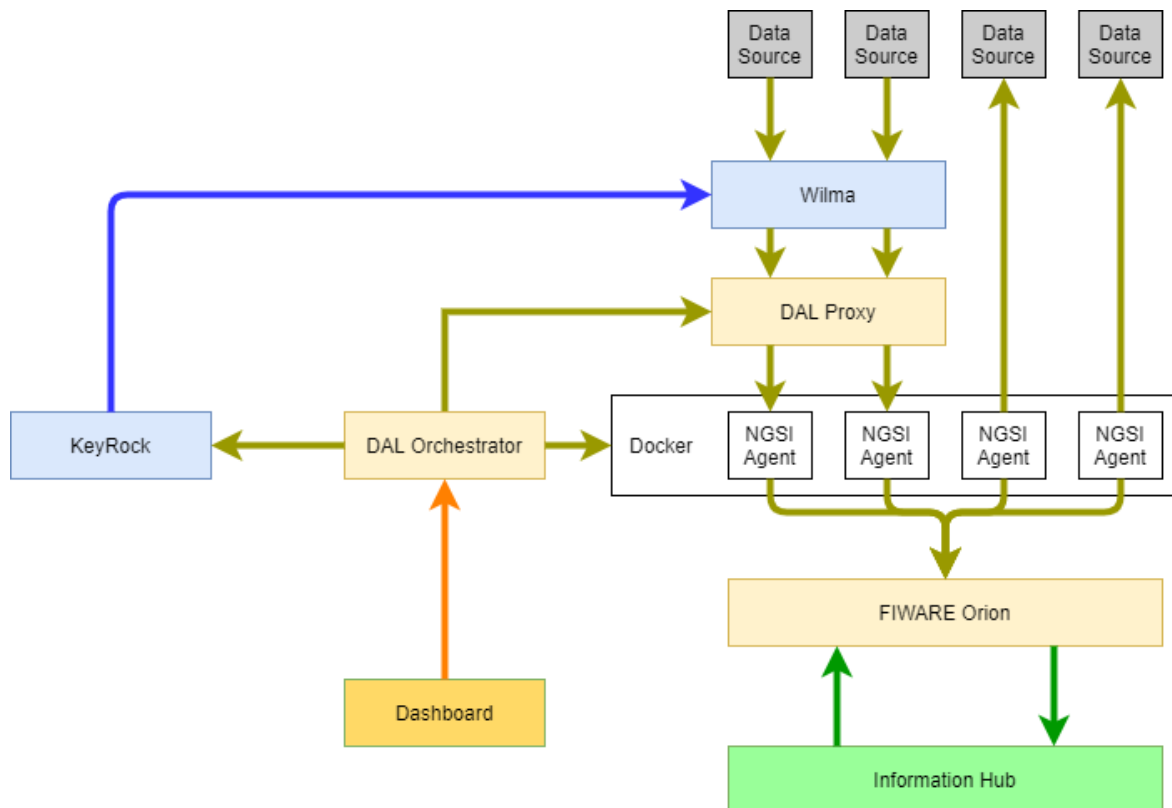


Figure 1. Interaction scheme DAL- Dashboard with security support

The main activity of the DAL relates to NGSI Agents pushing data to Orion. Then the Information Hub subscribes to new data using the subscription API of FIWARE Orion. It is then notified using a call-back when new data arrive.

The API of FIWARE Orion is well documented by FIWARE. The documentation on the NGSIv2 API is available [here](#), and documentation on how to use it is available [here](#).

To facilitate NGSI Agents management, the DAL provides an orchestrator that pilots the creation of the NGSI Agents using a Docker Interface. The DAL Orchestrator also communicates with the DAL Proxy to automatically expose new daemon NGSI Agents behind WILMA. The DAL Orchestrator exposes its API to the Dashboard to allow the creation of an administrative UI.

The DAL Orchestrator also communicates with the Keyrock enabler to manage the permissions on WILMA for each NGSI Agent that exposed an API through the DAL Proxy.

3.2. NGSI Agents

NGSI Agents are the small software pieces used to import data from external data sources into the PIXEL platform through the Data Acquisition Layer. There are 3 kinds of NGSI Agents:

- **Daemon:** running as a server to manage data continuously
- **Scheduled:** starts automatically at the given period
- **Manual:** running only when asked (on demand)

In order to run as an NGSI Agent (as Docker container) it needs some special configurations. Those configurations are done using Docker LABEL that could be overwritten when deploying an agent on the destination platform.

In order to be identify the Docker image of an agent it has to contains specifics labels.

3.2.1. Labels for all agents

- ✓ **ngsiagent="pixel":** this is the key label to be identified as a NGSI Agent
- ✓ **ngsiagent.type="daemon":** define the type of NGSI Agent daemon, scheduled or manual
- ✓ **ngsiagent.datasources="[\"urn:pixel:DataSource:dummies\"]":** this label provides the name of the datasource managed by this agent
- ✓ **ngsiagent.datamodels="[\"/Dummies/minimal-schema.json\"]":** this label provide the path to each JSON Schema generated by the agent

The DataModels Path is the relative path to the specs folder of the Data_Models repository.

For example for the data model TideSensorObserved the label should be set like this:
ngsiagent.datamodels="[\"/Pixel/TideSensorObserved/schema.json\"]"

3.2.2. Labels for daemon agents

- **ngsiagent.internal.port:** the port exposing the API, it also has to be specified with EXPOSE
- **ngsiagent.internal.path:** the base path of the API configured in the agent
- **ngsiagent.external.path:** the base path of the API configured in the proxy to expose the agent

3.2.3. Labels for scheduled agents

- **ngsiagent.scheduled:** the frequency to run the agent (CRON format)

```
* * * * *
| | | | |
| | | | | +-- Year          (range: 1900-3000)
| | | | +---- Day of the Week (range: 1-7, 1 standing for Monday)
| | | +----- Month of the Year (range: 1-12)
| | +----- Day of the Month (range: 1-31)
| +----- Hour (range: 0-23)
+----- Minute (range: 0-59)
```

3.2.4. Examples

- **Daemon**

```
FROM nginx
LABEL ngsiagent="pixel"
LABEL ngsiagent.type="daemon"
LABEL ngsiagent.internal.port="80"
LABEL ngsiagent.internal.path="/api"
LABEL ngsiagent.external.path="/empire"
LABEL ngsiagent.datasources=["urn:pixel:DataSource:dummies\"]]
LABEL ngsiagent.datamodels=["/Dummies/minimal-schema.json\"]]
EXPOSE 80
ENV PIXEL=test
ENV MYTEST=pixel
RUN mkdir /usr/share/nginx/html/api
RUN echo "Execute order 66" > /usr/share/nginx/html/api/order
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

- **Scheduled**

```
FROM ubuntu
LABEL ngsiagent="pixel"
LABEL ngsiagent.type="scheduled"
LABEL ngsiagent.scheduled="* * * * *"
LABEL ngsiagent.datasources=["urn:pixel:DataSource:dummies\"]]
LABEL ngsiagent.datamodels=["/Dummies/minimal-schema.json\"]]
ENV PIXEL=test
ENV MYTEST=pixel
ENV SCHEDULED_DELAY=0
COPY docker_entrypoint.sh /docker_entrypoint.sh
RUN chmod u+rx /docker_entrypoint.sh
ENTRYPOINT ["/docker_entrypoint.sh"]
```

- **Manual**

```
FROM ubuntu
LABEL ngsiagent="pixel"
LABEL ngsiagent.type="manual"
LABEL ngsiagent.datasources=["urn:pixel:DataSource:dummies\"]]
LABEL ngsiagent.datamodels=["/Dummies/minimal-schema.json\"]]
ENTRYPOINT ["/bin/bash"]
CMD ["date"]
```

3.3. DAL Orchestrator

The DAL Orchestrator has been developed to simplify the deployment and management of the NGSI Agents offering an API to be managed. The server exposes a Swagger and SwaggerUI to simplify the integration for the developer.

- <http://<server ip>:<exposed port>/api-docs>

3.3.1. Paths

The full API is described in a Swagger file. The list of functions is listed below.

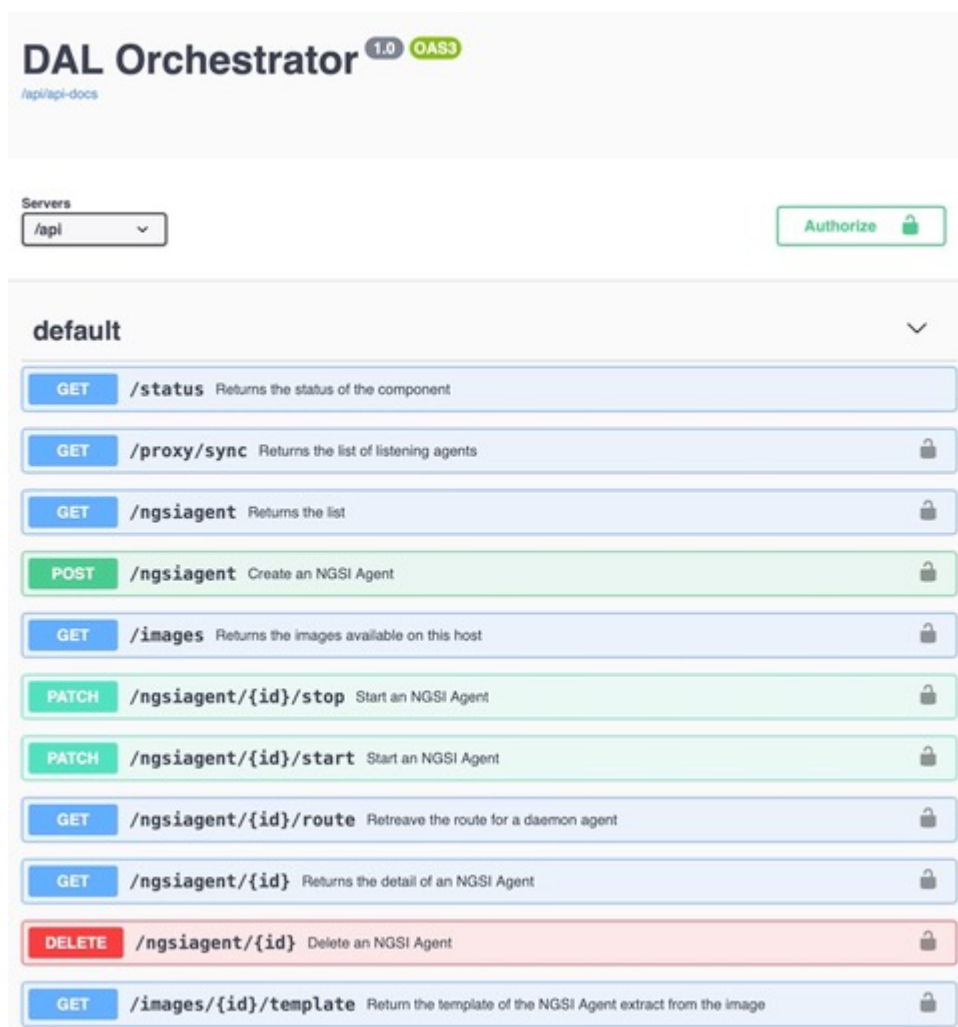


Figure 2. Swagger UI for the DAL Orchestrator

3.3.2. Models

The swagger file (UI) also describes the models used with this API

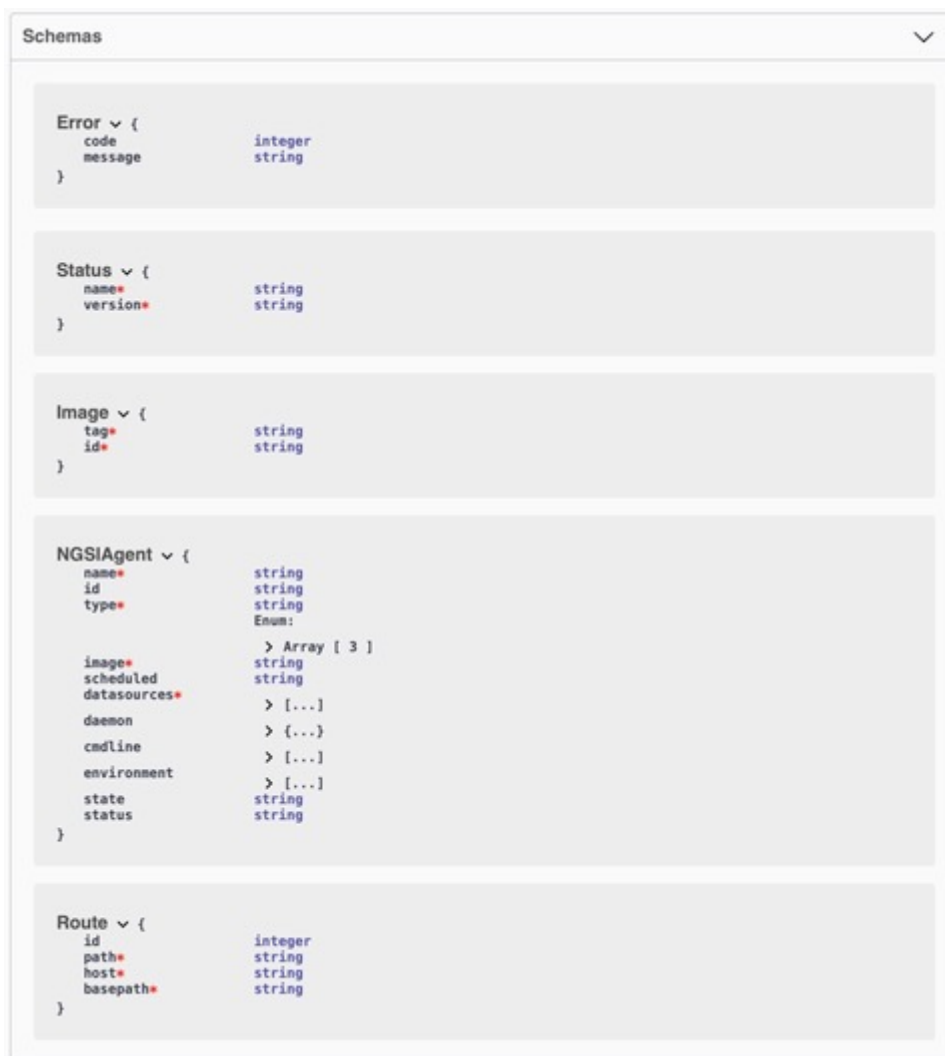


Figure 3. Model schemas for the DAL Orchestrator API

3.3.3. Quick start guide

- **NGSI Image management**

For security purposes, right now you have to run the command of **docker pull** to get (pull) the NGSI Agents images directly on the host. The next version will propose to manage that using the API.

You can request the list of available NGSI Agents images already available on the host with an API call:

```

curl -H "X-Auth-Token: default" http://172.17.0.1:8888/api/images
[
  {
    "id": "sha256:620877b976447800bc7ce8672d6b688369b429ad77afba0968f20088c8daf8fd",
    "tag": "pixelh2020/frbodtidesensor:1.0.0"
  }
]

```

- **Get a template**

When you have chosen the image of your NGSi Agents, you can generate a template to create it

```
curl -H "X-Auth-Token: default"
http://172.17.0.1:8888/api/images/sha256:620877b976447800bc7ce8672d6b688369b429ad77afba096
8f20088c8daf8fd/template
{
  "name": "/?[a-zA-Z0-9_-]+",
  "image": "pixelh2020/frbodtidesensor:1.0.0",
  "type": "scheduled",
  "scheduled": "22 * * * *",
  "datasources": [
    "urn:pixel:DataSource:frbod:TideSensorObserved"
  ],
  "datamodels": [
    "/Pixel/TideSensorObserved/schema.json"
  ],
  "environment": [
    {
      "key": "PATH",
      "value": "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    },
    {
      "key": "NODE_VERSION",
      "value": "13.6.0"
    },
    {
      "key": "YARN_VERSION",
      "value": "1.21.1"
    },
    {
      "key": "NODE_TLS_REJECT_UNAUTHORIZED",
      "value": "0"
    },
    {
      "key": "ORION_URL",
      "value": "changeit"
    },
    {
      "key": "NAMI_AUTH_URL",
      "value": "https://nami.bordeaux-port.fr/?q=accueil"
    },
    {
      "key": "NAMI_URL",
      "value": "https://nami.bordeaux-port.fr/hauteurs"
    },
    {
      "key": "NAMI_LOGIN",
      "value": "changeit"
    },
    {
      "key": "NAMI_PASSWORD",
      "value": "changeit"
    },
    {
      "key": "FIWARE_SERVICE="
    },
    {
      "key": "FIWARE_SERVICE_PATH="
    }
  ]
}
```

- **Create the NGSI Agent**

Change the name of the agent (it will be the name of the container) and adjust the parameters or let their default values. Be sure that your name match the given pattern.

```
curl -X POST -H "X-Auth-Token: default" http://172.17.0.1:8888/api/ngsiagent -d @- <<EOF
{
  "name": "/my-agent",
  "image": "pixelh2020/frbodtidesensor:1.0.0",
  "type": "scheduled",
  "scheduled": "22 * * * *",
  "datasources": [
    "urn:pixel:DataSource:frbod:TideSensorObserved"
  ],
  "datamodels": [
    "/Pixel/TideSensorObserved/schema.json"
  ],
  "environment": [
    {
      "key": "ORION_URL",
      "value": "http://172.17.0.1:1026"
    },
    {
      "key": "NAMI_LOGIN",
      "value": "mylogin"
    },
    {
      "key": "NAMI_PASSWORD",
      "value": "mypassword"
    }
  ]
}
EOF
```

3.4. Developer's guide

3.4.1. Managing Data Format with Information Hub

To allow the Information Hub to automatically import data from Orion, it was decided to publish Data Sources and Data Formats in the Orion database. That information is fulfilled by the DAL Orchestrator using the information contained in the Dockerfile of the NGSI Agents.

- **DataSource format is**

```
{
  "id": "urn of the data source",
  "type": "DataSource",
  "name": {
    "type": "Text",
    "value": "the source name if it is not an urn"
  }
}
```

- **DataModel format is**

```
{
  "id": "type name as declared in the orion entity",
  "type": "DataModel",
  "schema": {
    "type": "StructuredValue",
    "value": "an object containing the json schema"
  },
  "schemaUrl": {
    "type": "string",
    "value": "an url to the schema"
  },
  "schemaEncoded": {
    "type": "STRING_URL_ENCODED",
    "value": "a text version URL encoded of the schema if it contains forbidden
characters"
  }
}
```

Here **schemaUrl** is mandatory, **schema** should be provided for compatibility reasons with previous version, and **schemaEncoded** has to be present only if the schema contains forbidden chars.

- **SourceModelRelation format is**

```
{
  "id": "urn of the relation datasource/dataModel",
  "type": "SourceModelRelation",
  "source": {
    "type": "Text",
    "value": "urn/id of the DataSource"
  },
  "model": {
    "type": "Text",
    "value": "Data Model provide by the DataSource"
  }
}
```

3.4.2. Additional notes

To facilitate development a **docker-compose** is provided, with a shell script to create the environment test needed to develop and test the DAL Orchestrator.

This environment allows running the complete tests suite delivered with the software.

The start.sh script allows creating a nodejs environment to develop with DAL Orchestrator.

Refer to the README.md for detailed documentation of the software architecture.

4. PIXEL Information Hub

4.1. Overview

As described in D6.2, the Information Hub is a functional block in charge of centralising all the data retrieved from the DAL, homogenising and storing in a database capable to support big queries and scale horizontally. Unlike the DAL, the Information Hub is designed to be high performant and scalable, and the data is stored to support long-term queries. This is considered the central storage point of the IoT solution in PIXEL and is the block that replies the queries from other functional blocks (such as Operational Tools or Dashboards) and externals (API). The Information Hub's main components are a high-performance data broker and a NoSQL database, although it contains accessory components that support its correct functioning.

The following diagram depicts the architecture of Information Hub (source: D6.2):

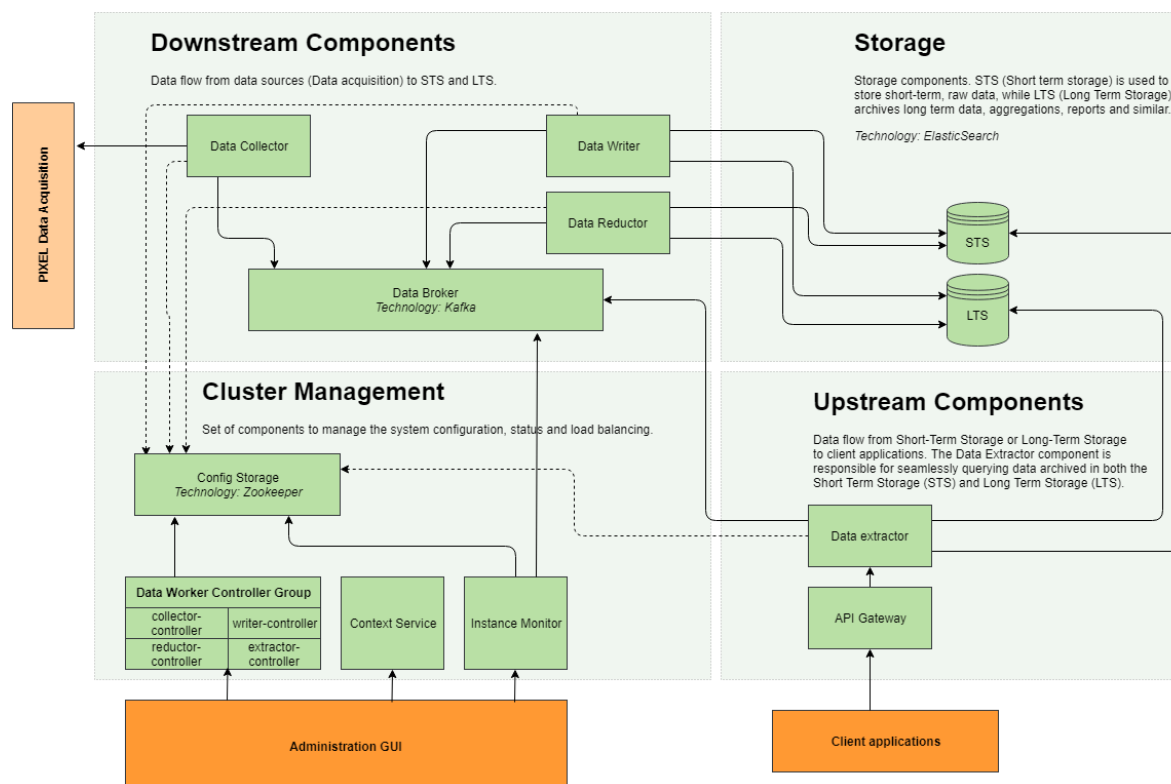


Figure 4. Information Hub - Architecture overview

The Information Hub consists of several parts conceptually divided into components that push data towards the database (downstream), components involved in stored data retrieval and further processing (upstream) and components responsible for data persistence and storage. In addition, the system provides supporting services for configuring, managing and monitoring the Information Hub and libraries contributing to greater extensibility and reusability. Below is a list of all services that form the Information Hub.

Components involved in the downstream flow:

- Data Collector
- Data Writer
- Data Broker
- Data Reductor

- Data Processor

Components involved in the upstream flow:

- Data Extractor
- Data Broker

Storage components:

- Short-Term Storage
- Long-Term Storage
- Configuration Service

Supporting components:

- Context Service
- Instance Monitor
- Data Collector Controller
- Data Writer Controller
- Data Processor Controller
- Data Reductor Controller
- Data Extractor Controller
- Management Console

The Controller components form the Data Worker Controller Group. Together with the Context Service and Instance Monitor they provide REST and SSE endpoints for configuring and monitoring the Information Hub, e.g. from another network, without imposing dependencies on core components. This is accomplished by an agreement that all interactions between core components and other parts of the system must be realized through the Configuration Service.

Multiple supporting components belonging to the Data Worker Controller Group are used for configuring the Information Hub at different stages by accessing and modifying data in the Configuration Service. Client applications communicate with controller components using an HTTP REST protocol and server sent notifications, both implemented on top of the Jersey framework. To reduce code duplication and provide common ways of accessing controller APIs, the Controller Library has been developed as a thin client library, containing REST API resource definitions and helper methods encapsulating Jersey framework specifics. The Context Service has been developed for a similar purpose but is not restricted to controlling only one specific core component. Therefore, it provides REST endpoints for configuring Sources and Source Types, regulates system maintenance and other common operations.

The Information Hub has a built in mechanism for monitoring hardware utilization and data flow in different stages of the Information Hub data pipeline. Similar to controllers, the Instance Monitor provides REST and SSE endpoints. In comparison to controllers it also depends on long term storage, for retrieving the aforementioned metrics and for persisting non-configuration related data. Utilization and data flow metrics are retrieved in form of status event records and analysed based on pre-programmed conditions to produce system notification.

4.2. REST API

As described in the previous chapter, following Information Hub services provide a REST API for configuring and monitoring the Information Hub:

- Context Service
- Data Monitor
- Data Collector Controller
- Data Writer Controller
- Data Processor Controller
- Data Reductor Controller
- Data Extractor Controller

Furthermore, Data Extractor provides a REST API for querying the data archived in both the short-term and long-term storages.

Complete specification of Information Hub APIs is available at the [PIXEL Documentation Hub](#) in the form of Swagger generated documentation.

4.2.1. Paths

Context Service

Context Service API ^{0.8.0}

[Base URL: CONTROLLER_HOST:8015/archivingSystem/context/v1]

Schemes

HTTP

Components

GET /components Get a list of custom component instances

GET /components/{componentId}/instances
/{instanceId} Get details about specified custom component instance

Configuration

GET /config Export the global system configuration data from ZooKeeper configuration service

PUT /config Import the global system configuration data

Data Fields

GET /sourceType/{sourceTypeId}/field Get a list of all data fields of specified source type

POST /sourceType/{sourceTypeId}/field Add a new data field to specified source type

GET /sourceType/{sourceTypeId}/field/{fieldId} Get details about the specified data field

PUT /sourceType/{sourceTypeId}/field/{fieldId} Update the specified data field

DELETE /sourceType/{sourceTypeId}/field/{fieldId} Delete the specified data field

Notifier

GET /config/event Subscribe to Context service configuration events

Settings

GET /settings/connection Get system connection settings from the ZooKeeper configuration service

PUT /settings/connection Update system connection settings in the ZooKeeper configuration service

GET /settings/context Get system context settings from the ZooKeeper configuration service

PUT /settings/context Update system context settings in the ZooKeeper configuration service

Sources ▼

- GET /sourceType/{sourceTypeId}/source/{sourceId} Get details about the specified source
- PUT /sourceType/{sourceTypeId}/source/{sourceId} Update the specified source
- DELETE /sourceType/{sourceTypeId}/source/{sourceId} Delete the specified source
- GET /sourceType/{sourceTypeId}/source Get a list of all registered sources of specific source type
- POST /sourceType/{sourceTypeId}/source Create a new source of specified source type

Source Types ▼

- GET /sourceType/{sourceTypeId} Get details about the specified source type
- PUT /sourceType/{sourceTypeId} Update specified source type
- DELETE /sourceType/{sourceTypeId} Delete specified source type
- GET /sourceType Get a list of all registered source types
- POST /sourceType Create a new source type

Storage ▼

- DELETE /storage/sts/delete Delete already reduced records from the short-term storage older than the specified time

System ▼

- GET / Get global system configuration common to all Information Hub components
- PUT / Update global system configuration common to all Information Hub components

Data Monitor

Data Monitor API 0.8.0

[Base URL: MONITOR_HOST:8020/archivingSystem/monitor/v1]

Schemes

HTTP

Events

GET

/subscribe/event Subscribe to data flow, system and status events

DELETE

/status/delete Delete status records older than the specified time

GET

/subscribe/event/dataflow Subscribe to data flow events

GET

/subscribe/event/system Subscribe to system events

GET

/subscribe/event/status Subscribe to status events

Notifications

GET

/subscribe/notification Get latest notification records (the first page)

DELETE

/notification/delete Delete notification records from the database older than the specified time

GET

/notification/{page} Get specified page of notification records

GET

/notification Get latest notification records (the first page)

Storages

GET

/storage/stsInfo Get statistics for each short-term index present in the Information Hub

GET

/storage/ltsInfo Get statistics for each long-term index present in the Information Hub

Orion Collector Controller

Orion Collector Controller API 0.2.0

[Base URL: CONTROLLER_HOST:8011/archivingSystem/collector/v1/admin]

Schemes

HTTP

Configuration

GET /**config** Export full configuration tree of the Orion Collector component from ZooKeeper configuration service

PUT /**config** Apply new configuration for the Orion Collector component

Instances

GET /**instance/{instanceId}** Get detailed configuration for the specified Collector instance

PUT /**instance/{instanceId}** Update configuration of the specified Collector instance

GET /**instance** Get a list of all Orion Collector instances

Instance Sources

GET /**instance/{instanceId}/source/active** Get a list of sources being collected by the specified Collector instance

GET /**instance/{instanceId}/source/binding** Get a list of bound sources for the specified Collector instance

PUT /**instance/{instanceId}/source/binding** Set list of sources bound to the specified collector instance

Notifier

GET /**config/event** Subscribe to configuration events from the Orion Collector service

Orion

POST /**orion/sync** Retrieve and register sources and source types from Orion Context Broker

Service

GET / Get global Orion Collector instance configuration

PUT / Update the global Orion Collector instance configuration

Sources

GET /**source** Get a list of all collected sources

Data Writer Controller

Data Writer Controller API 0.8.0

[Base URL: CONTROLLER_HOST:8012/archivingSystem/writer/v1/admin]

Schemes

HTTP

Configuration

GET /**config** Export full configuration tree of the Data Writer component from ZooKeeper configuration service

PUT /**config** Apply new configuration for the Data Writer component

Instances

GET /**instance** Get a list of all Data Writer instances

GET /**instance/{instanceId}** Get detailed configuration for the specified Data Writer instance

PUT /**instance/{instanceId}** Update configuration of the specified Data Writer instance

instances

GET /**instance/{instanceId}/source/active** Get a list of sources being consumed by the specified Data Writer instance

Notifier

GET /**config/event** Subscribe to configuration events from the Data Writer service

Service

GET / Get global Data Writer instance configuration

PUT / Update the global Data Writer instance configuration

Sources

GET /**source** Get a list of all sources consumed by the Data Writer

Data Extractor

Data Extractor API 0.8.0

[Base URL: EXTRACTOR_HOST:8080/archivingSystem/extractor/v1]

Schemes

HTTP ▾

Data Interval ▾

POST /data Retrieve time series data from specified data source using provided filters

Devices ▾

GET /device Get a list of all registered devices

GET /device/{deviceName}/{propertyName} Get storage types for specified device property

GET /device/{deviceName} Get properties of the specified device

Query ▾

POST /query/latestCollapseByField Get latest data record for each sensor of specified data source

Sources ▾

GET /sources Get a list of all registered sources

GET /sources/{sourceId} Get details about the specified data source

Source Types ▾

GET /sourceTypes/{sourceTypeId} Get details about the specified source type

GET /sourceTypes Get a list of all registered source types

Time Interval ▾

POST /time Get time intervals where all specified conditions are met

4.2.2. Models

Models	↕
<pre>ArhInstanceList { instances > [...] instanceDescriptions > {...} }</pre>	↕
<pre>ShortInstance > {...}</pre>	↕
<pre>ArhInstance { id string hostname string enabled boolean active boolean status string componentId string }</pre>	↕
<pre>ArhConfigNode > {...}</pre>	↕
<pre>ArhDataFieldList ></pre>	↕
<pre>ShortSourceField ></pre>	↕
<pre>ArhDataField ></pre>	↕
<pre>EventOutput ></pre>	↕
<pre>Type > {...}</pre>	↕
<pre>ArhSystemConnections ></pre>	↕
<pre>ArhSystemContextSettings ></pre>	↕
<pre>ArhReduction ></pre>	↕
<pre>ArhSelectorFilter ></pre>	↕
<pre>ArhSource ></pre>	↕
<pre>ArhSourceList ></pre>	↕

Models	
ArhConfigNode > {...}	↕
ArhCollectorInstance < { id string hostname string enabled boolean active boolean status string componentId string loadBalanced boolean threadCount integer(\$int32) } }	↕
ArhInstanceList > {...}	↕
ShortInstance > {...}	↕
ArhReservedSourceList >	↕
ReservedSource >	↕
ArhBoundSourceList >	↕
EventOutput > {...}	↕
Type > {...}	↕
OrionSourceResult >	↕
OrionSyncResult > {...}	↕
OrionTypeResult >	↕
OrionSyncInput > {...}	↕
ArhService >	↕

4.3. Developer's guide

The Information Hub is developed in Java technology and managed by the Maven tool. It is organized as a multi-module Maven project where each module is located in its own Git repository. The main repository is [information-hub-aggregator](#) which contains a Maven aggregator POM file which specifies all modules of the project. In addition to this aggregator POM the project also contains a parent POM which defines common Maven configuration that is inherited by other modules. The parent POM file is located in the information-hub-parent repository. The information-hub-aggregator repository includes other repositories as Git submodules. A submodule is a Git repository nested inside a parent Git repository at a specific path in the parent repository's working directory. Submodules are configured in the .gitmodules file located in the root of the information-hub-aggregator repository.

The Information Hub is distributed as a set of Docker images and can be deployed using the Docker Compose tool. The [information-hub-docker](#) repository provides Docker Compose projects for installing Information Hub together with Elasticsearch. The POM file contains Maven configuration for building Docker images during the package phase using Spotify Dockerfile Maven plugin.

The Information Hub Management Console is a Java desktop application and is distributed as a ZIP archive containing an executable JAR package with dependencies and a configuration file. The source code is located in app-controller-gui repository and is included in the information-hub-aggregator repository as apps/app-

controller-gui module. The Management Console is developed with the JavaFX software platform. It can be built and run using Oracle JDK or OpenJDK; however, OpenJDK requires additional JavaFX libraries.

4.3.1. Building from sources

Required tools:

- Git
- JDK 8 (OpenJDK or Oracle JDK can be used)
- Maven 3
- Docker

Clone the information-hub-aggregator repository:

```
git clone https://gitpixel.satrdlab.upv.es/xlab/information-hub-aggregator.git
```

Navigate to the information-hub-aggregator directory and initialize Git submodules of the project using following command:

```
git submodule init
```

Update the submodules by running:

```
git submodule update
```

This command will clone missing submodules and checkout the commit specified in the index of the containing repository. This will leave the submodule repositories in a detached HEAD state by default.

To display a list of all submodules, currently checked out commit for each submodule together with its status, run the ‘git submodule status’ command:

```
information-hub-aggregator$ git submodule status
+4b7194ac4e9236a9e98d5200c02be47dda2cdb90 apps/app-controller-gui (v0.8.0-2-g4b7194a)
7d82d77aa512d62e8cd0b4f18ec7f60b3100005c controllers/information-hub-controller (heads/develop)
8ec03d8fd00f7c75da948a9404344e0ba82e26 controllers/orion-data-collector-controller (heads/features/schema-parser)
b62f8a6f0f4448aee0ae38381a8819358e5b660b controllers/srv-data-controller (v0.8.0-1-gb62f8a6)
9fa46a214b63eb60d359273030fd9b31ae7d5e72 libs/lib-broker (v0.8.0-1-g9fa46a2)
63b02b9ac51b7d880312f0cf967e70b8b9948e06 libs/lib-config (v0.8.0-11-g63b02b9)
3fde3d6649bede724cba917a617a289c34877f4c libs/lib-controller (v0.8.0-1-g3fde3d6)
81f4ee295acf71b4b98bf9832cf12c010bbb3816 libs/lib-core (v0.8.0-1-g81f4ee2)
5b8a5c2832c6dd46bbfd56d9f6740c7fbc1d4cd4 libs/lib-extractor (v0.8.0-1-g5b8a5c2)
5cfee669d2f1e38aaeb96d6b1df1d08ee6f983de libs/lib-monitor (v0.8.0)
449a358932fcec7a116e8a6fdad3c9cfd73de17 libs/lib-reductor (v0.8.0)
e7f3ad4250166f936f34b07d06f4005720ce42b8 libs/lib-status (v0.8.0)
941b91d2a620669dd90aeba0a843cfa537998c45 libs/lib-storage (v0.7.0-66-g941b91d)
485999faae7e1bc2791e3762944932a05b7a8d49 optools/operational-tools-core (heads/develop)
569a433d0ca86d655ab81113238cdbffc1c46e81 optools/operational-tools-etc (remotes/origin/develop)
1526fa95317a3d771ee57e283d451c69d342cac pom-parent (v0.8.0-2-g1526fa9)
f9327b6fb02570849807789221161f5835db1cf6 services/ais-data-collector (heads/develop)
e6dfd62be4c2b07b827fb445873fd23d98b5b1de services/orion-data-collector (heads/develop)
b0a653ea96dd7192b57a2d4a1c4227b6b2cc6d6c services/srv-data-extractor (v0.8.0-22-gb0a653e)
91511b40553063b00c72aa662c7e2bac1cf20db8 services/srv-data-monitor (v0.8.0-2-g91511b4)
4529e0a60f62b194942f6723af70f7281dec8a10 services/srv-data-reductor (v0.8.0)
e02c7874c6acc5e7525499a5b5741fb6e52f4197 services/srv-data-writer (v0.8.0-6-ge02c787)
```

Figure 5. List of submodules

Build the project using following command:

```
mvn package
```

To speed up the build process you can use the `-Dmaven.test.skip=true` switch. When done, maven will print the build summary as depicted in the figure below:

```

[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] archiving-parent 0.8.0 ..... SUCCESS [ 2,121 s]
[INFO] lib-config 0.8.0 ..... SUCCESS [ 16,538 s]
[INFO] lib-broker 0.8.0 ..... SUCCESS [ 5,164 s]
[INFO] lib-status 0.8.0 ..... SUCCESS [ 2,456 s]
[INFO] lib-core 0.8.0 ..... SUCCESS [ 2,286 s]
[INFO] lib-storage 0.8.0 ..... SUCCESS [ 6,370 s]
[INFO] lib-extractor 0.8.0 ..... SUCCESS [ 2,835 s]
[INFO] lib-controller 0.8.0 ..... SUCCESS [ 2,564 s]
[INFO] lib-monitor 0.8.0 ..... SUCCESS [ 1,497 s]
[INFO] lib-reductor 0.8.0 ..... SUCCESS [ 2,365 s]
[INFO] orion-data-collector 0.2.0-SNAPSHOT ..... SUCCESS [ 13,786 s]
[INFO] ais-data-collector 0.2.0-SNAPSHOT ..... SUCCESS [ 11,270 s]
[INFO] srv-data-writer 0.8.0 ..... SUCCESS [ 21,784 s]
[INFO] srv-data-extractor 0.8.1-SNAPSHOT ..... SUCCESS [ 25,802 s]
[INFO] srv-data-reductor 0.8.0 ..... SUCCESS [ 20,572 s]
[INFO] srv-data-monitor 0.8.0 ..... SUCCESS [ 22,043 s]
[INFO] elasticsearch-proxy 1.0 ..... SUCCESS [ 11,952 s]
[INFO] data-processor-orchestrator 0.2.0-SNAPSHOT ..... SUCCESS [ 7,666 s]
[INFO] srv-data-controller 0.8.0 ..... SUCCESS [ 0,121 s]
[INFO] srv-data-collector-controller 0.8.0 ..... SUCCESS [ 4,557 s]
[INFO] srv-data-writer-controller 0.8.0 ..... SUCCESS [ 4,628 s]
[INFO] srv-data-reductor-controller 0.8.0 ..... SUCCESS [ 4,333 s]
[INFO] srv-data-extractor-controller 0.8.0 ..... SUCCESS [ 4,682 s]
[INFO] srv-context-service 0.8.0 ..... SUCCESS [ 9,284 s]
[INFO] srv-data-proxy-controller 0.8.0 ..... SUCCESS [ 4,138 s]
[INFO] orion-data-collector-controller 0.2.0-SNAPSHOT ..... SUCCESS [ 11,554 s]
[INFO] information-hub-controller 0.2.0-SNAPSHOT ..... SUCCESS [ 11,389 s]
[INFO] operational-tools-core 0.2.0-SNAPSHOT ..... SUCCESS [ 2,400 s]
[INFO] operational-tools-etc 0.2.0-SNAPSHOT ..... SUCCESS [ 8,681 s]
[INFO] app-controller-gui 0.8.0 ..... SUCCESS [ 13,474 s]
[INFO] information-hub-aggregator 0.1.0-SNAPSHOT ..... SUCCESS [ 0,032 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 04:18 min

```

Figure 6. Build process

The Maven builds all the modules into JAR archives and generates Docker images for all modules representing Information Hub services. To view the generated Docker images, run the following command:

```

information-hub-aggregator$ docker images
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
docker.pixel-ports.eu/information-hub/operational-tools-etc    latest   f3d5eadd4b90 36 minutes ago 132MB
docker.pixel-ports.eu/information-hub/information-hub-controller  latest   45b1a6b69e1d 36 minutes ago 164MB
docker.pixel-ports.eu/information-hub/data-processor-orchestrator  latest   18d16d52de52 37 minutes ago 124MB
docker.pixel-ports.eu/information-hub/elasticsearch-proxy        latest   417120ec464f 37 minutes ago 132MB
docker.pixel-ports.eu/information-hub/srv-data-monitor           latest   72751413fe03 37 minutes ago 159MB
docker.pixel-ports.eu/information-hub/srv-data-reductor          latest   c86f1a85bba1 38 minutes ago 157MB
docker.pixel-ports.eu/information-hub/srv-data-extractor         latest   681413f831d1 38 minutes ago 162MB
docker.pixel-ports.eu/information-hub/srv-data-writer            latest   c1e68ec4a422 38 minutes ago 152MB
docker.pixel-ports.eu/information-hub/ais-data-collector         latest   2dd522705b7f 39 minutes ago 142MB
docker.pixel-ports.eu/information-hub/orion-data-collector       latest   5a3525739267 39 minutes ago 133MB

```

Figure 7. Docker images after building the IH

4.3.2. Development environment

The Information Hub can be run in the local development environment from the Java IDE. First clone and initialize the [information-hub-aggregator](#) project as described in the previous section. The Information Hub requires following third-party software for running:

- Apache Zookeeper
- Apache Kafka
- Elasticsearch
- Orion Context Broker if Orion Data Collector is used

The [information-hub-docker](#) repository provides Docker Compose projects for deploying all the required software. Clone the repository, navigate into kafka-zookeeper, elastic and orion-context-broker directories and run ‘docker-compose up -d’ command to deploy corresponding services.

Add following entries to the /etc/hosts file:

```

172.17.0.1 cscs.archiving.broker
172.17.0.1 cscs.archiving.config
172.17.0.1 cscs.archiving.sts cscs.archiving.its

```

```
172.17.0.1 cisco.archiving.controller
172.17.0.1 cisco.archiving.monitor
```

cisco.archiving.broker represents Apache Kafka hostname, cisco.archiving.config represents Apache Zookeeper hostname and cisco.archiving.sts / Its Elasticsearch. Adapt the IPs accordingly if you deployed these services to a different machine.

Orion Data Collector's configuration is located in the infhub.properties file inside orion-data-collector module:

```
orion.address=http://172.17.0.1:1026
orion.header.fiware-service=
orion.header.fiware-servicepath=
orion-collector.notification.callback.url=http://172.17.0.1:9009
orion-collector.notification.listener.port=9009
```

After finishing the configuration, you can start the Information Hub services and management console by running following Java classes from your IDE:

Table 2. Information Hub services and corresponding Main classes

Service	Main class
information-hub-controller	si.xlab.pixel.infhub.controller.InfHubController
srv-data-monitor	de.gsi.cs.co.sv.archiving.monitor.MonitorService
orion-data-collector	si.xlab.pixel.infhub.collector.orion.OrionDataCollector
srv-data-writer	de.gsi.cs.co.sv.archiving.writer.Writer
srv-data-extractor	de.gsi.cs.co.sv.archiving.extractor.Extractor
app-controller-gui	de.gsi.cs.co.sv.archiving.gui.admin.ArchivingAdminApp

4.3.3. Potential extensions

The Information Hub is designed in a modular and scalable way and as generic as possible. This design allows it to be easily extended with new functionalities. Archiving System, the base of Information Hub which was developed by XLAB for the FAIR (Facility for Antiproton and Ion Research) particle accelerator facility in Darmstadt, Germany was extended for the needs of PIXEL project with following new components:

- Orion Data Collector: new type of the Data Collector component which collects data from data sources on Orion Context Broker.
- AIS Data Collector: new type of the Data Collector component which collects AIS data from AISHub sharing service.
- Data Processor: new type of component which can be plugged into the Information Hub platform and is used for on-the-fly (near real-time) data processing of incoming data streams. Specifically, this type of component was used for ETD (estimated time of departure) calculation on the VesselCall data records.
- Data Processor Orchestrator: new type of component which takes care for managing Data Processors and routing data streams through them by using Apache Kafka topics.

4.3.3.1. Implementing Data Collector for a new data source

The Data Collector is a component responsible for obtaining data records from various devices and data sources, filtering and pre-processing data records and pushing them downstream through the Data Broker for further processing. Multiple types of Data Collector components can be used simultaneously to collect data from different data sources. In addition to the Data Collector component, the Data Collector Controller is also required which provides a REST API for controlling and managing Data Collector instances.

To develop a Data Collector for a new type of data source, you can take the Orion Data Collector project as a sample available in the [orion-data-collector](#) repository. The corresponding Data Collector Controller project is available in the [orion-data-collector-controller](#) repository.

The OrionDataCollector class serves as a Collector's entry point. It gets the Collector's configuration file path provided as a command-line parameter and initializes an OrionCollectorContext instance using this path. The COMPONENT_NAME field is used as a service name for displaying in the Information Hub management console. Finally, the OrionDataCollector instantiates and runs OrionCollectorController which controls the operation of the collector.

The OrionCollectorContext object loads configuration properties from the provided configuration file and exposes them through getter methods. It creates an instance of StatusProducer which is used for reporting the Collector service status information to the Monitoring service which is displayed in the Information Hub management console.

The OrionConfigService serves for managing Collector service's configuration in Apache Zookeeper which is used as a centralized config storage. When Collector starts, it registers in the Zookeeper as depicted in the following figure:

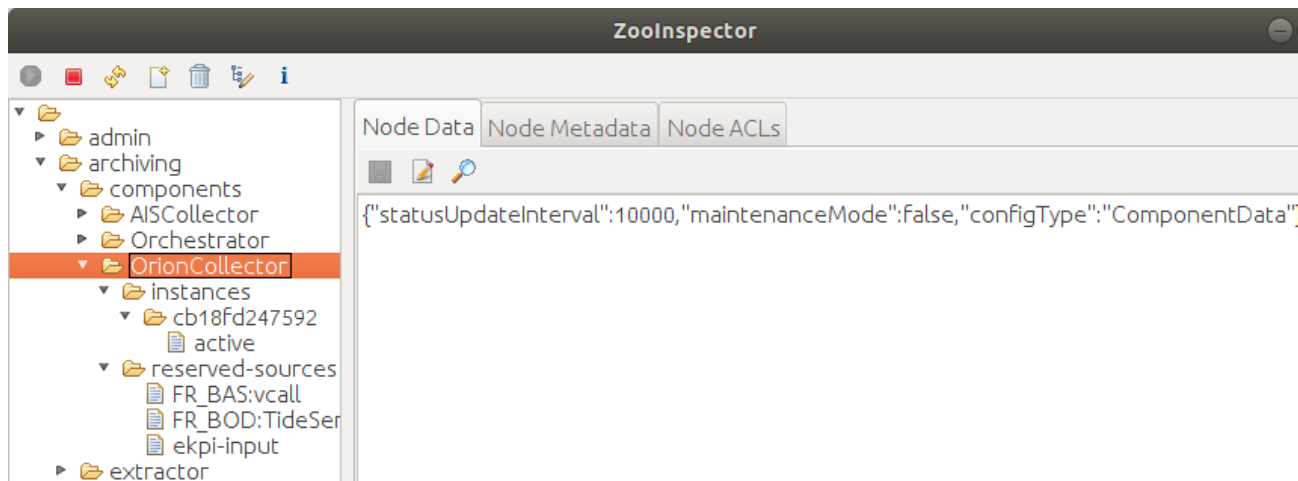


Figure 8. Zookeeper monitoring

All the configuration and management of the Collector goes via the Zookeeper. Collector sets a watch on the corresponding znode in ZooKeeper which triggers on any change. When the Collector Controller (API) updates the configuration in Zookeeper, the Collector gets a notification and takes an appropriate action, for example starts collecting new data source. This way scalability is achieved, and new Collector instances can be added dynamically.

The OrionDataCollectorWorker is responsible for collecting data from Orion Context Broker and managing subscriptions to data sources in Orion. When a new Orion data source is allocated to the Collector instance by the load balancer (through Zookeeper), the OrionDataCollectorWorker subscribes to notifications of data source events and creates a notifications handler. The notification handler accepts notification messages, extracts Orion entities and converts them to JSON objects containing a map of attribute name-value pairs in accordance with the source type schema.

Let’s take as an example the TideSensorObserved source type. The schema of the source type as shown in Information Hub management console is depicted in the following figure:

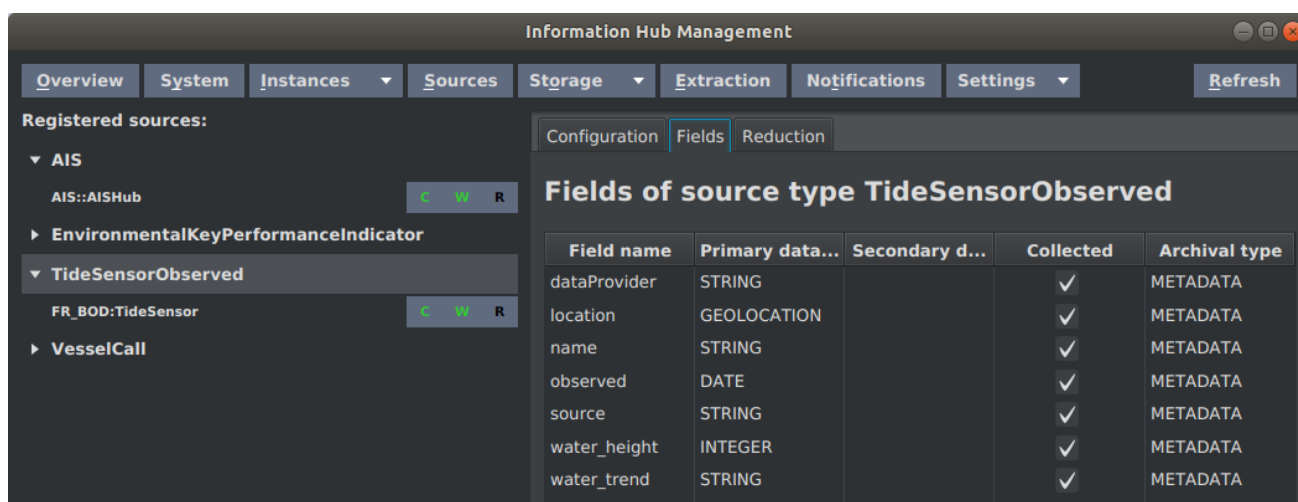


Figure 9. Information Hub Management (TideSensorObserved example)

Orion sends data entities in the following format as depicted in figure below:

```

1 {
2   "id": "FRB0D:TideSensor:Bordeaux:2019-12-28T08:42:00.000Z",
3   "type": "TideSensorObserved",
4   "dataProvider": {
5     "type": "Text",
6     "value": "https://nami.bordeaux-port.fr/hauteurs",
7     "metadata": {}
8   },
9   "location": {
10    "type": "geo:json",
11    "value": {
12      "type": "Point",
13      "coordinates": [
14        -0.548,
15        44.859
16      ]
17    },
18    "metadata": {}
19  },
20  "name": {
21    "type": "Text",
22    "value": "Bordeaux",
23    "metadata": {}
24  },
25  "observed": {
26    "type": "DateTime",
27    "value": "2019-12-28T08:43:00.000Z",
28    "metadata": {}
29  },
30  "source": {
31    "type": "Text",
32    "value": "FR_B0D:TideSensor",
33    "metadata": {}
34  },
35  "water_height": {
36    "type": "Integer",
37    "value": "471",
38    "metadata": {}
39  },
40  "water_trend": {
41    "type": "Text",
42    "value": "down",
43    "metadata": {}
44  }
45 }

```

Figure 10. Orion entity (*TideSensorObserved* example)

The Collector converts this Orion entity from Orion format to the following JSON object containing map of attribute name-value pairs:

```

1 {
2   "dataProvider": "https://nami.bordeaux-port.fr/hauteurs",
3   "location": {
4     "lat": -0.548,
5     "lon": 44.859
6   },
7   "name": "Bordeaux",
8   "observed": 1577522580000,
9   "source": "FR_B0D:TideSensor",
10  "water_height": "471",
11  "water_trend": "down"
12 }

```

Figure 11. Orion entity conversion by the Collector (*TideSensorObserved* example)

The Collector then converts the JSON object into a JSON string and creates an ArchiveRecord object:

```

String recordJson = record.toString();
ArchiveRecord archiveRecord = new ArchiveRecord(
    source.getSourceTypeId(),
    source.getSourceId(),
    System.currentTimeMillis());
archiveRecord.setRecordId(orionRecord.get("id").asText());
archiveRecord.setData(recordJson.getBytes());

```

The ArchiveRecord constructor has three parameters: source type ID, source ID and timestamp. The current time is used as the record timestamp in this case. The Orion entity ID is used as the record ID which makes

possible to provide the updating records functionality. If a record with the same ID already exists in Elasticsearch, the existing record will be updated instead of a new one created. The content of archiveRecord is set using the setData method to the JSON object shown above converted to a byte array.

The ArchiveRecord is then pushed downstream through the Data Broker (Apache Kafka is used as a data broker). Each data source uses its own data publisher (IDataPublisher object) because sources might be routed to different Kafka topics according to the routing table which enables additional on-the-fly data processing in Data Processor components. The OrionDataCollectorWorker takes suitable publisher from the publishers' map and publishes the data record:

```
IDataPublisher publisher = this.publishers.get(source.getSourceId());
publisher.send(source, record);
```

4.3.3.2. Implementing a new component

To support some custom functionality, it may be required to develop a new component type for the Information Hub. For example, a Data Processor component was developed to support on-the-fly data processing functionality.

Following the project structure, it is recommended for developing a new component:

4.3.3.2.1. Main class

The main class (e.g. OrionDataCollector) provides an entry point to the application (main method) which is used for starting the service. The main class accepts configuration file path as a command-line parameter (if needed), initializes the component context and creates and starts the component's controller.

4.3.3.2.2. Controller

The Controller class (e.g. OrionCollectorController) contains logic for controlling the instance life cycle. At start up, it connects to the configuration service (Zookeeper), retrieves the component configuration and initializes and starts the operation of the service. Afterwards it continues to monitor the values in the configuration service and applies any changes there to the local operation.

The controller class must extend ServiceController abstract class defined in lib-core library:

```
public abstract class ServiceController<T extends IServiceContext> {
    protected abstract void start(T context, InstanceData instanceConfig);
    protected abstract void stop(boolean maintenance);
    protected void applyInstanceConfig(InstanceData instanceConfig){
        // Override
    }
    protected void applyComponentConfig(ComponentData componentConfig){
        // Override
    }
}
```

The start method is called when the service is started. It provides an InstanceData object containing the instance configuration in Zookeeper. The stop method is called when the stop of the service has been triggered. The maintenance parameter reveals if the stop was triggered by the operator activating the maintenance mode. The applyInstanceConfig method is called when the component configuration in Zookeeper is changed and the method has to apply those changes to the service operation. The applyComponentConfig method is called when common Information Hub components configuration has been changed.

4.3.3.2.3. Context

The Context class (e.g. OrionCollectorContext) contains all the runtime configuration of the currently running instance. It reads settings from the configuration file, environment variables and config service of the Information Hub. Furthermore, it creates an instance of StatusProducer for reporting service status information and provides this instance through getter method.

The Context class must extend ServiceContext class defined in lib-core library and may extend following methods to provide customized implementation:

```
InstanceConfigService getConfigService();
StatusProducer getStatusProducer();
InstanceData getDefaultInstanceConfiguration();
boolean doInit()
```

The getConfigService method returns an InstanceConfigService corresponding to the component for managing the component configuration in Zookeeper. The getStatusProducer method provides a StatusProducer instance. The getDefaultInstanceConfiguration method creates and returns an InstanceData (or appropriate child class) object containing default component configuration which is used when component is started for the first time to initialize the instance configuration node in Zookeeper. The doInit method can be overridden to load component specific configuration from configuration file or environment variables.

4.3.3.2.4. InstanceConfigService

The InstanceConfigService instance (e.g. OrionConfigService) provides methods for retrieving and managing component's configuration in the Zookeeper as well as paths to the appropriate nodes in the ZooKeeper. To use standard znodes structure for generic Information Hub components, extend the ComponentConfigService class and provide suitable ID which will be used as a node name:

```
public class MyConfigService extends ComponentConfigService {
    private static String COMPONENT_ID = "MyComponent";
    public MyConfigService() {
        super(COMPONENT_ID);
    }
}
```

The figure below depicts nodes structure for Orion Data Collector component:

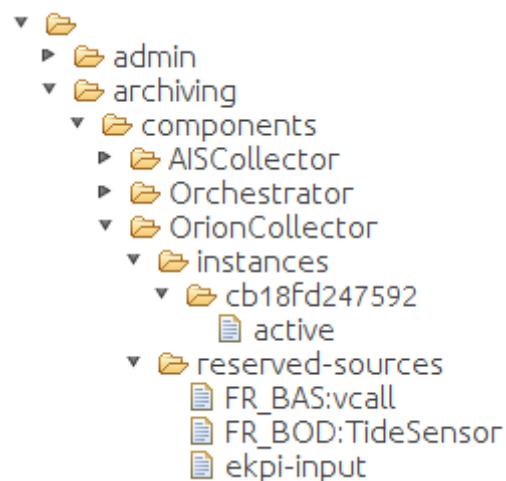


Figure 12. Nodes structure (Orion Data Collector)

4.3.3.2.5. Worker

The Worker class (e.g. OrionDataCollectorWorker) contains logic for the core operation of the component. It is instantiated and started by the Controller.

4.3.3.2.6. Configuration file and Log4j configuration

Static configuration properties (which don't change in runtime) can be put into configuration properties file which is loaded at the component start up by the Context class.

The Information Hub uses Apache Log4j 2 logging framework. The log4j2.xml file contains the logging configuration.

4.3.3.2.7. Dockerfile

The Dockerfile contains instructions for the Docker tool to build the Docker image of the component. The Information Hub is distributed as a set of Docker images; each component is packed into its own image. 'openjdk:8-jre-alpine' is used as a base image for Information Hub components. The figure below depicts Dockerfile for Orion Data Collector component:

```
FROM openjdk:8-jre-alpine
ARG JAR_FILE
COPY target/${JAR_FILE} /opt/inf-hub/lib/
COPY target/lib /opt/inf-hub/lib/
COPY src/main/resources/log4j2.xml /etc/inf-hub/
COPY src/main/resources/infhub.properties /etc/inf-hub/
ENTRYPOINT ["java", "-Dlog4j.configurationFile=/etc/inf-hub/log4j2.xml", "-cp", "/opt/inf-hub/lib/*", \
            "si.xlab.pixel.infhub.collector.orion.OrionDataCollector", "/etc/inf-hub/infhub.properties"]
```

Figure 13. Dockerfile for Orion Data Collector

5. PIXEL Operational Tools

5.1. Introduction

5.1.1. Main concepts and architecture

The Operational Tools (OT) are mainly in charge of bringing closer to the user both the models and predictive algorithms developed within the PIXEL project. By user here we mean administrators and managers analysing port operations by means of simulation models and predictive algorithms. In order to reach that goal, a set of high-level tasks are defined:

- Publish models and/or predictive algorithms
- Edit and configure the models and/or predictive algorithms
- Execute models and/or predictive algorithms
- Schedule models and/or predictive algorithms to be executed at a specific time once or periodically
- Define different operational and environmental Key Performance Indicators (KPIs), based on specific data available in the information hub for tracking and monitoring purposes
- Establish some pattern detection mechanism. The most basic one is the use of triggers.
- Get the trends of a model and/or predictive algorithm (e.g. historical data)

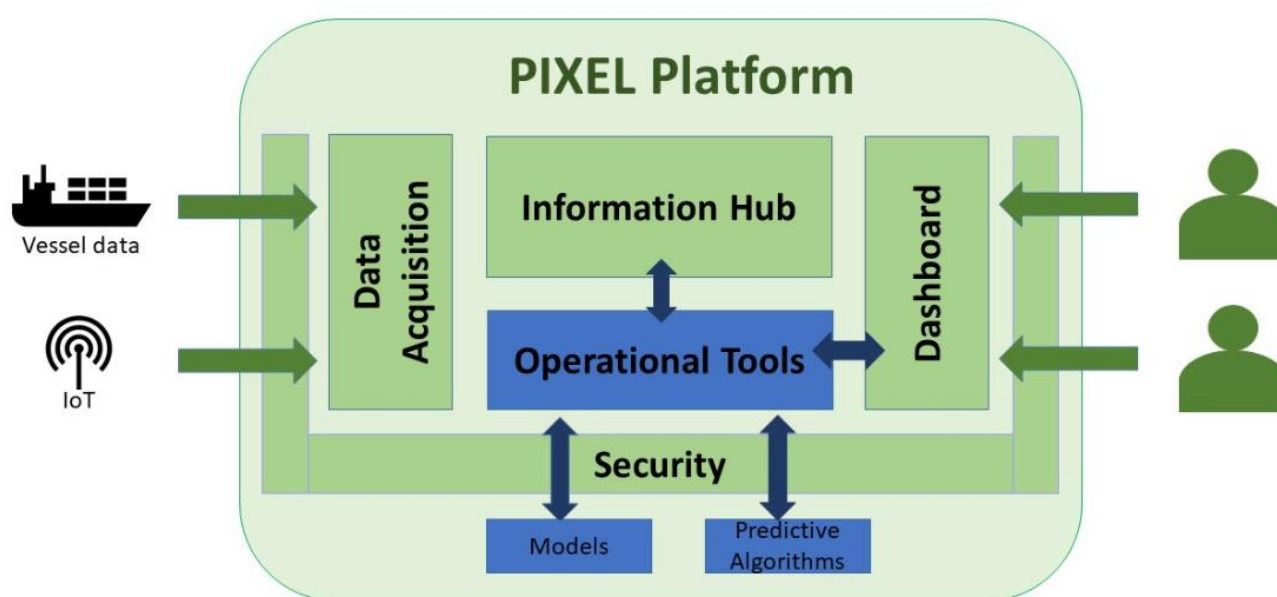


Figure 14. Operational Tools - Architecture overview

The functional overview of the Operational Tools is depicted in the Figure below. Several internal components can be identified:

- OT UI: this is the graphical interface to access (most of) the underlying functionalities. This component provides independence and autonomy, but it can be later integrated as part of the PIXEL dashboard to provide a single-entry point for administrators
- OT API: backend API implementing the functionalities needed. This component is aligned with PIXEL security framework in order to fulfil all required security policies (e.g. authentication, authorization, etc.)

- **Publication component:** it allows publishing both models and predictive algorithms. By publishing it may be necessary to deploy the models as Docker containers. Besides, the models ‘and predictive algorithms’ configurations can also be edited.
- **Engine:** this component is responsible for executing the different models and predictive algorithms. The execution can be invoked in real time or scheduled.
- **Data processing:** it is responsible for managing trends from specific data (KPIs) and also for some internal data adaptations required.
- **Event processing:** this component is responsible for real-time monitoring of indicators and trigger specific actions depending on previously configured rules. It includes a connector (bridge) to be integrated with an external notification system.
- **Database:** the database includes description of the models and predictive algorithms that can be used, KPI description, rules as well as other configuration and output related parameters necessary for the correct behaviour of the internal building blocks.

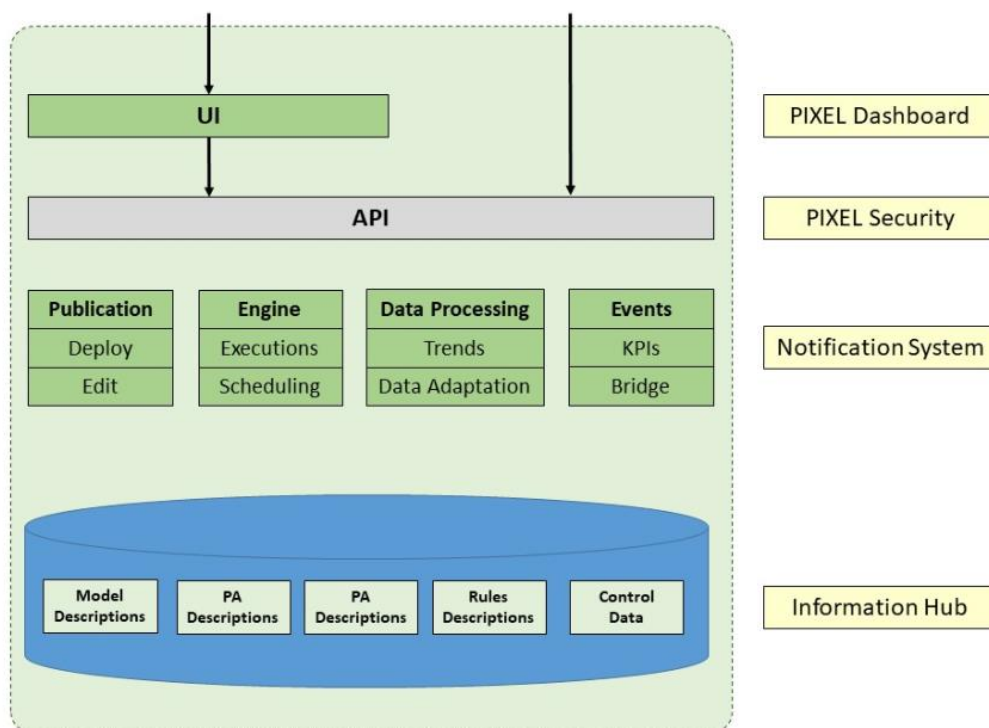


Figure 15. Operational Tools - Functional overview

5.1.2. Models

Models are entities in the PIXEL platform that will be used by port administrators to run and simulate models and predictive algorithms with different input parameters. As every model and predictive algorithm is different from each other and has its own internals, there is a need to homogenize a common abstract model entity to be the internal representation in the PIXEL platform. It encompasses two different types of developments that have been done within the PIXEL project:

- **Models:** models relate to energy, traffic and environment. A specific model is the Port Environmental Index (PEI). For more information about the models, please check the PIXEL main documentation repository by clicking [here](#).

- Predictive algorithms: predictive algorithms relate to estimating time of arrival in ports, traffic at gates and use of AIS data. For more information about the models, please check the PIXEL main documentation repository by clicking here.

The Figure below shows the process experienced by any model or predictive algorithm that is going to be used inside the PIXEL platform:

- The model or predictive algorithm is first drafted as algorithm and implemented as program.
- The model is encapsulated into a Docker container to convert it into a portable component. Additionally, an OT adaptor is attached to his Docker container in order to be integrated into the PIXEL platform.
- Through the publication process the model or predictive algorithm becomes aware into the PIXEL platform. The Docker image is pulled from the (open) GitHub repository and can be used internally.
- After published, the model or predictive algorithm can be executed by passing the appropriate arguments (parameters) as JSON file. The description of this JSON file will be described in future sections. The execution can run immediately (real time) or it can be scheduled to be performed periodically (e.g. every day or week).
- The results of the model are stored into the PIXEL Information Hub, which can be queried by the PIXEL dashboard to visualize them in form of particular graphs depending on the model or predictive algorithm.

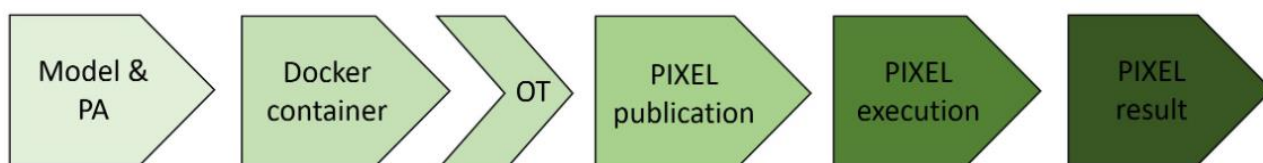


Figure 16. Link between models and the Operational Tools

5.1.3. Key Performance Indicators

According to Wikipedia a Key Performance Indicator (KPI) is a type of performance measurement. KPIs evaluate the success of an organization or of a particular activity in which it engages. For the PIXEL project, we envision that basic KPIs will mostly refer to:

- Sensors: the PIXEL platform encompasses an IoT network and can therefore monitor any integrated sensor. Some of the sensors may represent an important impact on the decision made from port authorities (e.g. depending on the tide level or the wind speed some cargo type is not recommended to be loaded/unloaded).
- Models and Predictive algorithms: models and predictive algorithms are typically complex and provide various different outputs; however, some specific items of the output can be considered of crucial importance and be characterized as KPIs.

More complex KPIs can be potentially defined by combining previous ones, but there is a need to define a common format for them as data entity. PIXEL has followed the FIWARE Data model, which specification can be accessed here. Some extensions have been added, when needed, to particularize it to port and model needs (e.g. environmental KPIs for the PEI calculation). You can find more information on the main documentation repository of PIXEL, clicking here, as there is a section dedicated to Data Models.

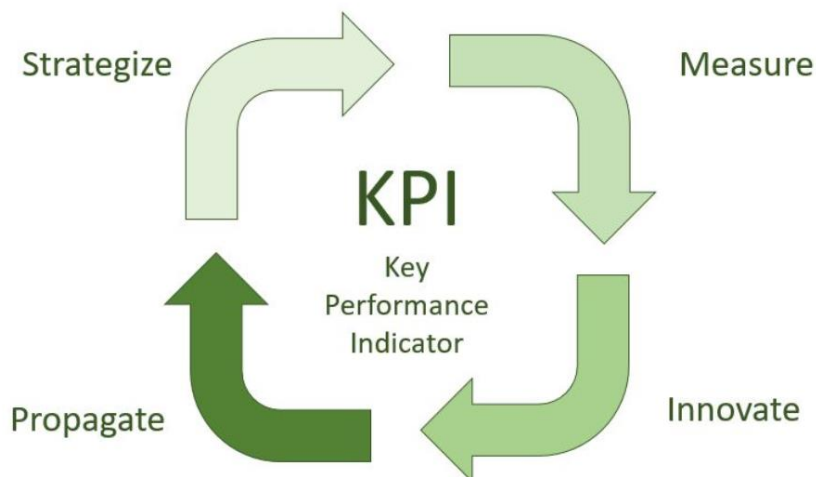


Figure 17. Key Performance Indicators

5.1.4. Event processing

According to Wikipedia a Event Processing is a method of tracking and analysing (processing) streams of information (data) about things that happen (events), and deriving a conclusion from them. Complex event processing, or CEP, consists of a set of concepts and techniques developed in the early 1990s for processing real-time events and extracting information from event streams as they arrive. The goal of complex event processing is to identify meaningful events (such as opportunities or threats) in real-time situations and respond to them as quickly as possible.

Considering that the PIXEL platform uses as main database Elasticsearch, the selected and natural choice as CEP engine refers to ElastAlert. You can find detailed information about ElastAlert by clicking [here](#). Some of its main features are reliability, modularity and easiness to set up and configure.

From the perspective of the Operational Tools, and considering the current needs of the target ports, this will mainly be related to monitored KPIs where some thresholds are reached. For these situations, rules and alerts are 'templated' to facilitate the configuration to port operators and define proper actions. More complex actions are possible and supported through ElastAlert; this will be commented in the Developer's Guide subsection, explaining possible extensions.

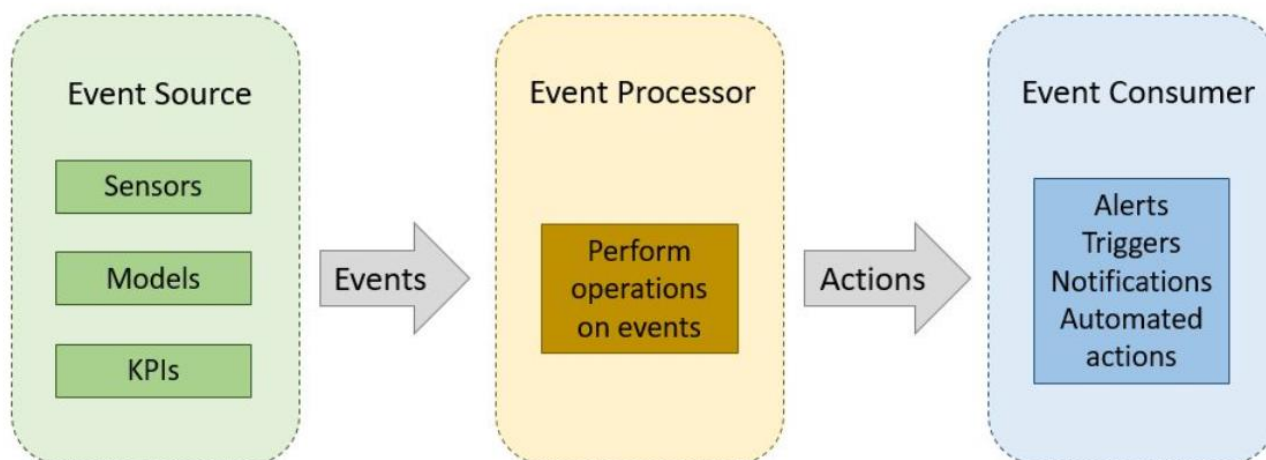


Figure 18. Operational Tools- Event Processing overview

5.2. Developer's guide

5.2.1. Identification of interfaces

As commented in the previous section, the OT interacts mainly with 3 external entities, as shown in the Figure below:

- Interface 1: This is the interface used by the Operational Tools to obtain information (e.g. eKPIs) from the Information Hub. Here the OT act mainly as user. Therefore, this interface will not be explained in this section, but in the Information Hub chapter as part of the PIXEL documentation.
- Interface 2: This is the main interface used, typically by the Dashboard, to publish and execute models, instances and scheduledInstances, as well as manage KPIs. This interface is developed as part of the Operational Tools and will be described below as Management Interface.
- Interface 3: The Operational Tools are somehow divided into a main component, being part of the PIXEL architecture, and also a OT adaptor integrated in each of the models and predictive algorithms to allow its integration and management inside the platform. This interface is developed as part of the Operational Tools and will be described below as Execution Interface.

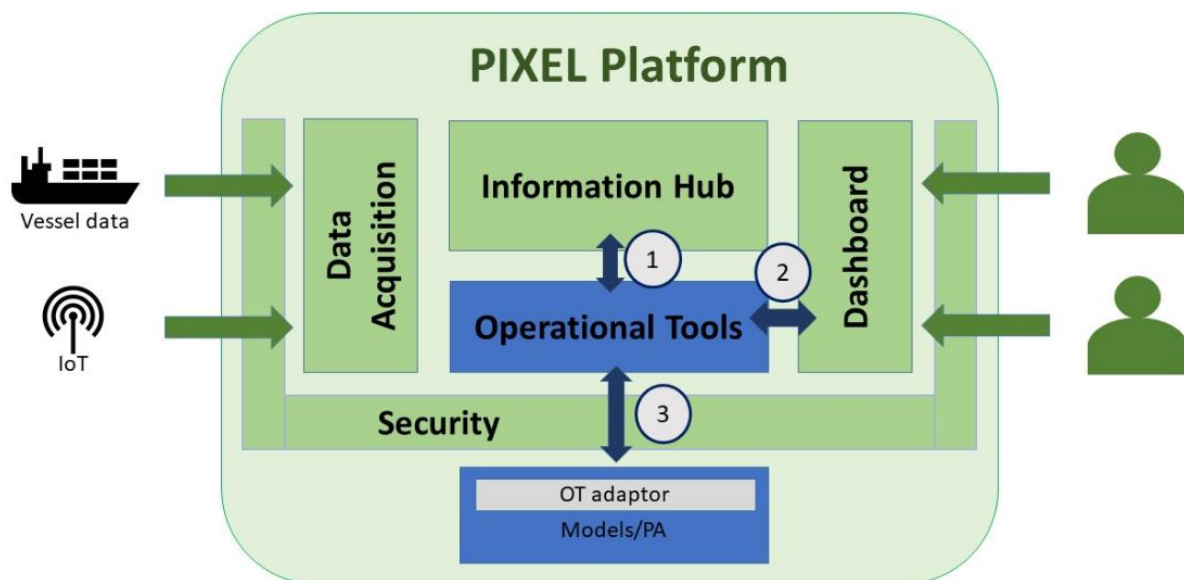


Figure 19. Operational Tools- Identification of interfaces

The Figure below depicts these 3 interfaces from the point of view of the internal blocks of the main components of the Operational Tools. As can be observed, the PIXEL Dashboard will invoke Interface 2 to manage the publication and execution of models. The Engine block of the OT, whenever a model or predictive algorithm needs to be executed, invokes the corresponding Docker instance, which incorporates an OT adaptor component able to understand the exchange of parameters through the Interface 3. The Interface 1 refers to the use of the Information Hub API to retrieve information. Storage of information as output of the execution of models and predictive algorithms is done by the Docker instance by means of the OT adaptor.

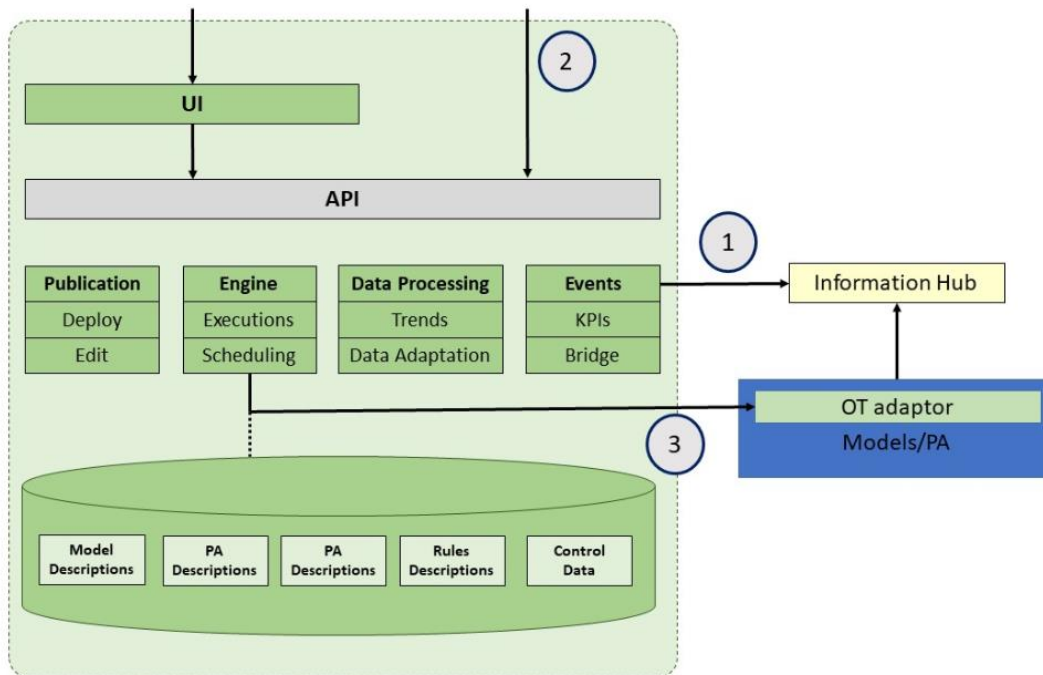


Figure 20. Operational Tools – Link interfaces and internal components

5.2.2. Management interface

This specification is intended for service consumers (with development skills). It provides a full specification of how to interoperate with the OT Management Service API.

The API user should be familiar with: - RESTful web services - HTTP/1.1. - JSON data serialization formats.

Users can perform the following actions through the CRUD (Create, Read, Update, Delete) API: - Manage models (both PIXEL models and predictive algorithms) - Manage instances (executions of models and predictive algorithms) - Manage scheduled instances (scheduled executions of models and predictive algorithms) - Manage KPIs (following the FIWARE KPI data format)

All endpoints require authentication. The Authorization HTTP header can be specified with ApiKey <your-key> to authenticate as a user and have the same permissions that the user itself. Example:


```
GET / HTTP/1.1
Host: ot_host
Authorization: ApiKey <your-key>
```

Once the OT main component is deployed, it provides an Swagger (OpenAPI) endpoint under the path http://<your_server>:8080/otpixel/doc/#!/ where you have a Swagger UI to test the API

Instance Resource		
PUT	/instances/create	Create an Instance
DELETE	/instances/delete/{id}	Delete an instance
GET	/instances/get/{id}	Get an Instance by id
GET	/instances/list	List all instances
POST	/instances/update	Update an instance
KPI Resource		
PUT	/kpis/create	Create a KPI
DELETE	/kpis/delete/{id}	Delete a kpi
GET	/kpis/get/{id}	Get a KPI by id
GET	/kpis/get/{id}/lastKPI	Get the last value of a KPI by id
GET	/kpis/get/{id}/stats	Get stats a KPI
GET	/kpis/list	List all models
POST	/kpis/update	Update a kpi
Model Resource		
PUT	/models/create	Create a model
DELETE	/models/delete/{id}	Delete a model
GET	/models/get/{id}	Get a model by id
GET	/models/get/{id}/info	Get information of a model by id
GET	/models/list	List all models
POST	/models/update	Update a model
Scheduled Instance Resource		
PUT	/scheduledInstances/create	Create a scheduled instance
DELETE	/scheduledInstances/delete/{id}	Delete a scheduled instance
GET	/scheduledInstances/get/{id}	Get a scheduled Instance by id
GET	/scheduledInstances/list	List all scheduled instances
POST	/scheduledInstances/update	Update a scheduled instance
POST	/scheduledInstances/updateStatus/{id}	Update only the status of a scheduled instance

Figure 21. Operational Tools – Management Swagger API

A complete list of all methods is available as a standalone HTML page (<https://docs-hub-ot.readthedocs.io/en/latest/ot-api.html>), containing examples of code for various programming languages (Java, JS, PHP, C#, Python, etc.). One can also see the different fields of the data formats as well as the response codes.



API SUMMARY

API METHODS - INSTANCE

createInstance
deleteInstance
getInstance
listInstances
updateInstance

API METHODS - KPI

createKPI
deleteKPI
getKPIById
getKPIStatsById
getLastKPIById
listKPIs
updateKPI

API METHODS - MODEL

createModel
deleteModel
getModel
getModelInfo
listModels
updateModel

API METHODS - SCHEDULEDINSTANCE

createScheduledInstance
deleteScheduledInstance
getScheduledInstance
listScheduledInstances
updateScheduledInstance
updateStatusScheduledInstance

OPERATIONAL TOOLS

API and SDK Documentation

The OT uses a REST API. All endpoints require authentication. The Authorization HTTP header can be specified with ApiKey to authenticate as a user and have the same permissions that the user itself.

InstanceResource

createInstance

Create an instance

If id is not provided, it will be randomly generated

PUT

```
/instances/create
```

Usage and SDK Samples

Curl Java Android Obj-C JavaScript C# PHP Perl Python

```
curl -X PUT -H "Authorization: [[apiKey]]" "http://<your_tomcat_server>:8080/olpixel/api/instances/create"
```

Parameters

Body parameters

Name	Description
body *	<div style="background-color: #f9f9f9; padding: 5px;"> <pre style="margin: 0;">{ id: string idRef: string name: string description: string mode: string user: string input: > [] forceinput: > [] output: > [] logging: > [] creation: integer (int64) start: integer (int64) otStatus: string dockerId: string }</pre> </div>

Responses

Status: 200 - successful operation

Schema

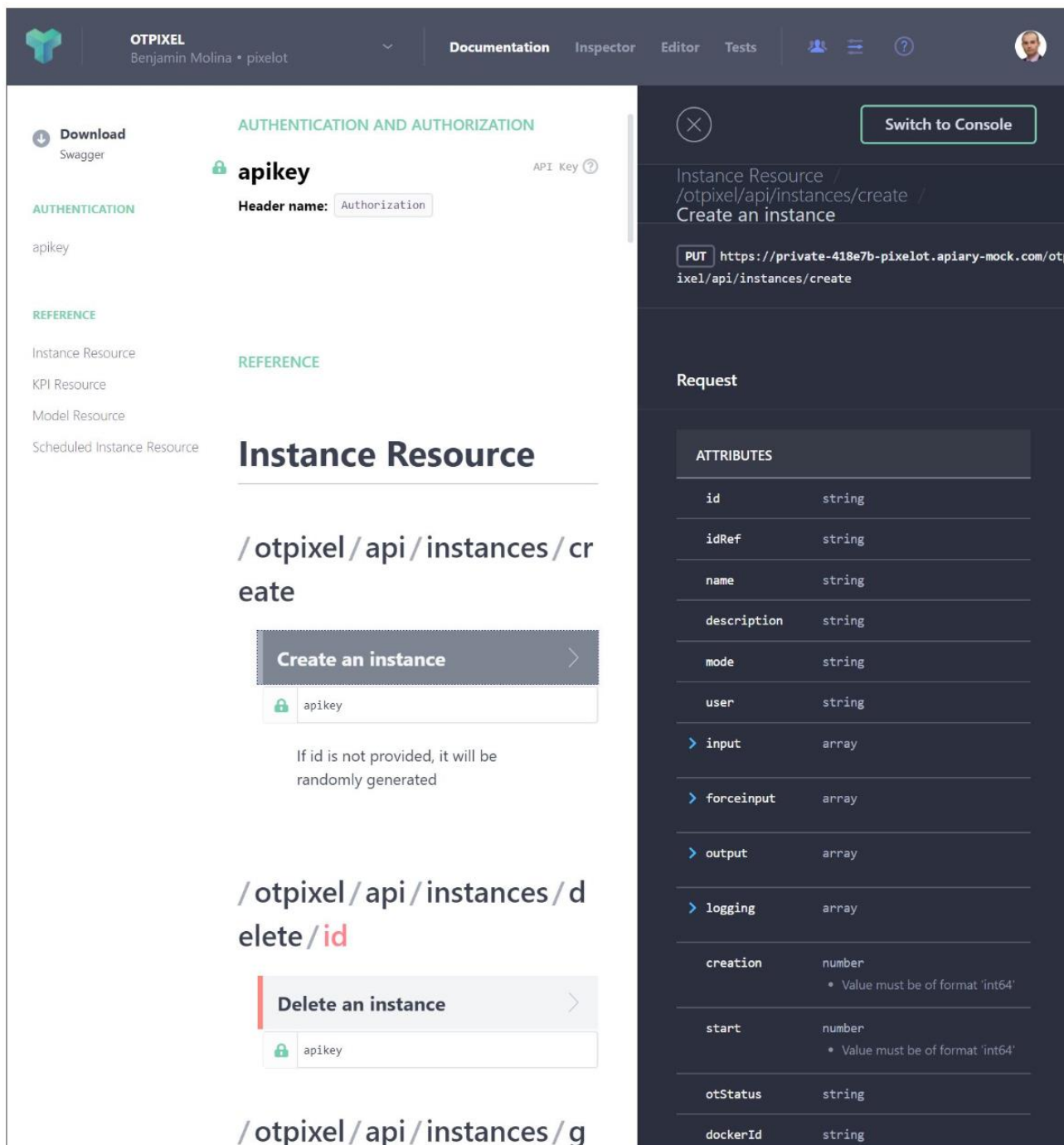
```
{ }
```

Status: 400 - Instance already exists

Status: 500 - Internal error

Figure 22. Operational Tools- Management APIs with code samples

Exploiting the potential of Swagger (OpenAPI) specifications, the OT management API has also been ported to apiary (<https://pixelot.docs.apiary.io/#>). Note that there is no proper real backend server to test the data, but you can see the functions as well as the JSON datatypes.



The screenshot displays the API documentation for the Management API, specifically the 'Instance Resource' section. The interface is dark-themed and includes a navigation bar at the top with options like 'Documentation', 'Inspector', 'Editor', and 'Tests'. The main content area is split into two columns. The left column shows the 'Instance Resource' section with endpoints for creating and deleting instances, each with a corresponding 'Create an instance' or 'Delete an instance' button and a form for the API key. The right column shows a console view with a 'Switch to Console' button and a PUT request to the create endpoint, along with a table of request attributes.

Instance Resource

`/otpixel/api/instances/create`

Create an instance

apikey

If id is not provided, it will be randomly generated

`/otpixel/api/instances/delete/id`

Delete an instance

apikey

`/otpixel/api/instances/g`

Request

ATTRIBUTES	
id	string
idRef	string
name	string
description	string
mode	string
user	string
input	array
forceinput	array
output	array
logging	array
creation	number • Value must be of format 'int64'
start	number • Value must be of format 'int64'
otStatus	string
dockerId	string

Figure 23. Operational Tools- Management API ported to Apiary

5.2.3. Execution interface

The OT Engine block is able to run Dockerized models and predictive algorithms if they include an specific OT adaptor to allow the integration. The execution flow is depicted in the Figure below, where several steps can be identified:

- Step 1: the main OT component launches the model via instantiating the corresponding Docker and passing an instance JSON file with all needed parameters.
- Step 2: The controller manages the whole internal execution of the model inside the Docker container following several steps. In step 2 it gets all inputs via the Input retriever module. This module should

be able to use both the Extractor and the Broker (Kafka) API of the Information Hub to obtain all needed data.

- Step 3: If there is a need to transform the input data, the Input Transformer is invoked. This might happen when the input data formats are not natively supported by the model itself, and some adaptation is needed.
- Step 4: The model algorithm is launched passing all the obtained inputs from the Information Hub. The controller shall monitor stdout, stderr to check whether the execution is going well or some errors appear.
- Step 5: If there is a need to transform the output data, the Output Transformer is invoked. The required transformations (if any) are mainly conditioned by latter efficient queries (e.g. visualization in the Dashboard).
- Step 6: The resulting (transformed) output is written in the IH via the Output writer. This module should be able to use the Extractor and/or the Broker (Kafka) API.

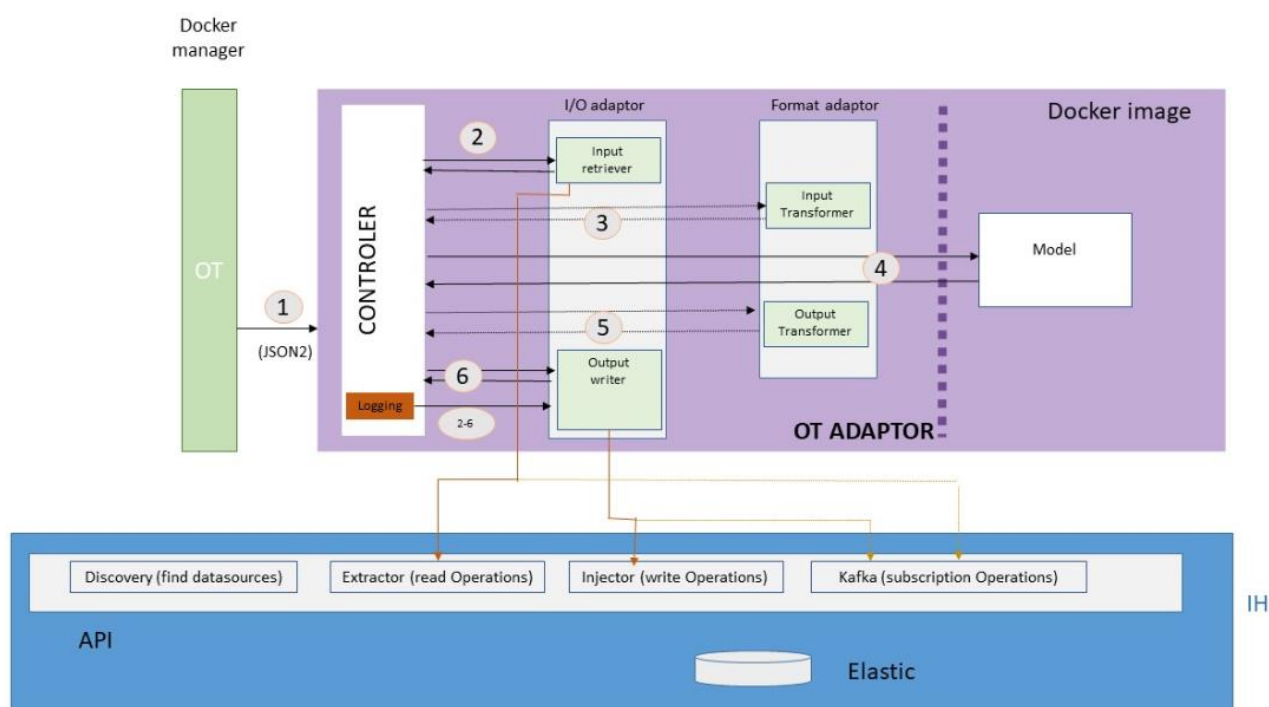


Figure 24. Operational Tools- Execution interface overview

The controller includes a logging functionality in order to monitor all steps. It should log: start, end (with status), and any intermediate information during the process (if any). The latter might be conditioned (level of logging) by some (possible) input verbose parameter.

In case of error, a typical logging table after an execution will look like

Table 3. Logging format example for model execution (error)

Timestamp	id_model (UUID)	id_execution (UUID)	type (String)	message (String)
2020-04-16T18:51:17+00:00	JJUSG676531	JJH6757423	start	-
2020-04-16T18:51:19+00:00	JJUSG676531	JJH6757423	error	error message 1
2020-04-16T18:51:20+00:00	JJUSG676531	JJH6757423	error	error message 2
2020-04-16T18:51:21+00:00	JJUSG676531	JJH6757423	end	error

In case of success, a typical logging table after an execution will look like

Table 4. Logging format example for model execution (success)

Timestamp	id_model (UUID)	id_execution (UUID)	type (String)	message (String)
2020-04-16T18:51:17+00:00	JJUSG676531	JJH6757423	start	-
2020-04-16T18:51:19+00:00	JJUSG676531	JJH6757423	info	info message 1
2020-04-16T18:51:20+00:00	JJUSG676531	JJH6757423	info	info message 2
2020-04-16T18:51:21+00:00	JJUSG676531	JJH6757423	end	success

5.2.4. Software Extensions

There are several potential extensions to be added to the existing implementation of the Operational Tools. Some of them are commented below

5.2.4.1. Include an additional resource in the Management API

There are several potential extensions to be added to the existing implementation of the Operational Tools. One of these consists in extending the API to include a new resource in its Management API, in case you need it to your specific needs. In this case, and assuming that you have already imported the code from github, you should follow these steps:

1. Add POJO class: Add new POJO class that represents the new resource in Java resources eu.pixel.otpixel.model. It is advisable that the class extends the utility class eu.pixel.otpixel.model.IdentifiableObject. For example, just copy Model.java into YourClass.java and adapt it accordingly, then generate Setters/Getters with Eclipse.

2. Create provide: Create a new provider interface in the package `eu.pixel.otpixel.datasource.dao.providers` with methods to interact with the new resource. If you want to add CRUD capabilities to the interface it is easier if the interface extends the utility interface `eu.pixel.otpixel.datasource.dao.CRUD<T>` where T is your new POJO resource class. Then you can add specific methods to that interface (see `eu.pixel.otpixel.datasource.dao.providers.ModelProvider`). For example, copy `ModelProvider.java` into `YourClassProvider.java` and change the basics to have something like

```
import eu.pixel.otpixel.model.YourClass;

public interface YourClassProvider extends CRUD<YourClass>{.. }
```

3. Modify interface: Modify the interface `eu.pixel.otpixel.datasource.DataSource` and add a method that forces the `DataSource` provider implementations to return an implementation of your provider interface created in the previous step. Eclipse will complain at this moment. Don't worry, keep on with the following steps and the problem will be solved (just an issue about dependencies)

```
public YourClassProvider getYourClassProvider();
```

4. 4Modify `DataSource` interfaces: After you modify the `DataSource` interface all the existing implementations will fail to compile. You should modify all `DataSource` implementations (`MongoDB`, `Memory`) since they have to return a specific implementation for that kind of `Datasource` of your provider interface. For example, to return a `MongoDB` implementation of that provider, create a new class in `eu.pixel.otpixel.datasource.impl.mongodb` that implements your provider interface. You will have to implement all the methods of that interface. In this case, if your provider interface extended `eu.pixel.otpixel.datasource.dao.CRUD<T>` it would be easier if this `MongoDB` provider class extended the utility class `eu.pixel.otpixel.datasource.impl.mongodb.AbstractMongoDBCRUDProvider` (see `eu.pixel.otpixel.datasource.impl.mongodb.MongoDBModelProvider`). For example, copy `MongoDBModelProvider.java` into `MongoDBYourClassProvider.java` and change the import and classes from `Model` to `Yourclass`. Adapt also `MongoDBDataSource.java` to include `MongoDBYourClassProvider`. You have to do the previous step in all different `DataSource` implementations (also for `memory`). Once you create your provider implementation you have to modify all `DataSource` implementations to return your provider implementation (in this case would be `eu.pixel.otpixel.datasource.impl.mongodb.MongoDBDataSource`).
5. Create resource: Once the DAO (Database Access Objects) are all well-defined and the project compiles again, create a new API resource access class in `eu.pixel.otpixel.api.resources`. You should follow REST compliance guidelines for that and keep consistency throught the project. To ensure that, the most easiest approach is to copy an already existing class such as `eu.pixel.otpixel.api.resources.ModelResource` and modify it for your new resource
6. Create converters: There is still something to add: the converters, for `MongoDB` implementation it is located at `eu.pixel.otpixel.datasource.impl.mongodb.converters`. First add the converter, similar to `eu.pixel.otpixel.datasource.impl.mongodb.converters.ModelConverter.java`, and then add a new method to `eu.pixel.otpixel.datasource.impl.mongodb.converters.MongoDBConverters.java`. For example, create `eu.pixel.otpixel.datasource.impl.mongodb.converters.YourClassConverter.java` from `eu.pixel.otpixel.datasource.impl.mongodb.converters.ModelConverter.java` and adapt it accordingly. Then add a new method in `eu.pixel.otpixel.datasource.impl.mongodb.converters.MongoDBConverters.java` in the static list of methods:

```
static{  
    converters.put(YourClass.class, new YourclassConverter());  
}
```

5.2.4.2. Enhance the Dockerized model

There are several ways in which you may enhance the provided Dockerized models and/or predictive algorithms, or just add new functionalities to your newly created ones. Some examples will be:

- Add new connector: currently all models are obtaining the information via the Information Hub, which stores all needed information under a common place. This requires that all needed information to be placed in the Information Hub via NGSI Agents and a connected Data Acquisition Layer (DAL). However, for a certain model, you are able to add a new connector able to retrieve directly opendata from external data sources. In that case, it is the Input Retriever component who is in charge of implementing this functionality. From the point of view of the Dashboard and the OT main component it should be a seamless upgrade, as long as the connector is well defined in the GetInfo.json and instance.json files.

The connector is defined in the GetInfo.json file. A possible example will be something like

```
"supportExecAsync": true,  
"type": "model",  
"category": "environment",  
"system": {  
    "connectors": [{  
        "type": "opendata-api",  
        "description": "this connector allows connecting to Opendata repo X",  
        "options": [{  
            "name": "url",  
            "type": "string",  
            "description": "url",  
            "required": true  
        }], {  
            "name": "reqParams",  
            "type": "string",  
            "description": "request parameters (if any)",  
            "required": false  
        }], {  
            "name": "headers",  
            "type": "headersObject",  
            "description": "necessary headers (if any)",  
            "required": false  
        }  
    }  
}
```



```

    }
  ] }
]
},

```

- Add verbosity level: Typically, the model implementation has a way to log and trace the execution of the model, with some log4j or similar functionality to store such information into a file. However, this is stored inside the Docker container and is lost once the execution finishes. Currently Dockerized models are mainly logging start and end of execution, without any intermediate trace being mandatory (only errors). However, you could add additional levels in the in the verbose option field of the logging element. For example, the GetInfo.json file could look like

```

"logging": [{
  "name": "your-name",
  "supportedConnectors": ["ih-api"],
  "type": "default-logging-format",
  "description": "",
  "required": true,
  "options": [{
    "name": "verbose",
    "type": "string",
    "description": "verbosity level (error, warning, info, debug)",
    "required": false
  },
  {...}
]
}]

```

5.2.5. Compilation from the sources

5.2.5.1. Development environment

The following requirements apply to this software component (Java part):

- **JDK 1.8+**: you should be able to compile the code both in Linux and Windows environments.
- **Eclipse IDE 2019**: download and import the project into this IDE and (if not already) convert it into a Maven project.

- **Apache Tomcat8:** the software component is compiled as a WAR file to be deployed on a Tomcat8 server (tested SO: Ubuntu 18.04 LTS with OpenJDK 1.8).
- **Mongo database:** the WAR application makes use of Mongo to store/persist information. It is supposed to be located on the same server as Tomcat 8; otherwise, change the configuration files accordingly.

The following requirements apply to this software component (Javascript part for the UI):

- **Nodejs:** v10.16.0
- **npm:** v6.9.0
- **VUE:** v3.9.2

5.2.5.2. Configuration

Before compiling, you will have to create various configuration files from the given templates:

- **build.local.properties:** It should server as template to create a file **build.properties** with the configuration parameters of your project. You can leave everything as it is and just change localhost with your server's IP or hostname.

```
#BUILD CONFIGURATION

jdk.version: 1.8
#put the same war.name as webapp.path to avoid possible conflicts
war.name: otpixel

#APP CONFIGURATION
webapp.name: OTPIXEL
webapp.path: /otpixel
webapp.appclass: eu.pixel.otpixel.App
webapp.api.package: eu.pixel.otpixel.api
webapp.api.path: /api

#KONGCHEN CONFIGURATION
webapp.kongchen.scheme: http
#change 'localhost' with your server's name
webapp.kongchen.host: localhost:8080
webapp.kongchen.basePath: /otpixel/api
```

- **server.local.properties:** It should server as template to create a file **server.properties** with the configuration parameters of your project. Here you can set the credentials for accessing your Tomcat8 server for deployment purposes. Please ensure that your Tomcat8 server is able to allow such operation (set up tomcat-user.xml properly if not).

```
server.scheme: http
# change parameters (Server's name/IP, usr, pass) to accommodate to your server
server.host: localhost:8080
tomcat.username: tomcat
tomcat.password: s3cret
```

- **log4j.xml:** Located under Java Resources --> resources. Configuration file for Log4j. You may adapt it to your needs (on single log file, or many).

- **default.local.configuration.xml**: Located under Java Resources --> resources. It should serve as template to create a file **default.configuration** with the configuration parameters of your project. You will have to set here your apiKey, your Elastic configuration and your Mongo (database) configuration mainly.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xml>
<configuration>
  <server>
    <headers>
      <header enabled="true" key="Access-Control-Allow-Origin" value="*" />
      <header enabled="true" key="Access-Control-Allow-Headers" value="origin,
content-type, accept, authorization" />
      <header enabled="true" key="Access-Control-Allow-Credentials" value="true" />
      <header enabled="true" key="Access-Control-Allow-Methods" value="GET, POST,
PUT, DELETE, OPTIONS, HEAD" />
    </headers>
  </server>

  <!-- API key when invoking the Swagger API. Change it according to your test
environment -->
  <security>
    <apikey>apikey</apikey>
  </security>

  <!-- Elastic server configuration. Change it according to your test environment --
>
  <elastic>
    <host>localhost</host>
    <port>9200</port>
    <scheme>http</scheme>
    <username>username</username>
    <password>password</password>
  </elastic>

  <!-- Frequency (in seconds) to search for new models added to the platform -->
  <ot-engine>
    <frequency>30</frequency>
  </ot-engine>

  <datasource>
    <className>eu.pixel.otpixel.datasource.impl.mongodb.MongoDBDataSource</className>
    <!-- <uri>mongodb://mongo:27017/otpixel</uri> this could be used in docker-
compose with mongo as a docker instance-->
    <uri>mongodb://localhost:27017/otpixel</uri>
  </datasource>
</configuration>

```

Additionally, for the UI, which is developed in Vue (javascript), you will need to configure the following:

- **settings.local.js**: Located under extra --> ui --> cfg. It should serve as template to create a file settings.js with the configuration parameters of your project. Just change localhost with your server's IP or network hostname.

```
(function(window) {  
  window.__env = window.__env || {};  
  
  window.__env.otpixelapi = {  
    "endpoint": "http://localhost:8080/otpixel/api",  
    "apiKey": "apikey"  
  };  
  
  window.__env.debug = true;  
})(this);
```

5.2.5.3. Compilation

STEP 1: Compile the UI code

If you don't need nor want to compile it, there is already a precompiled version under the folder 'www/ui'. In this case, just adapt the configuration file:

- **settings.local.js**: Located under www --> ui --> cfg. It should serve as template to create a file settings.js with the configuration parameters of your project.

If you want to compile the UI, just open a command line window on the location of the code (extra/ui) and type

```
npm install  
npm run build
```

If everything goes well (several warnings might appear) then just replace the content of the 'www/ui' of your Eclipse project with the content of the 'dist' folder you have just compiled.

STEP 2: Compile the WAR application

In order to package the program into a WAR file, just right click on the pom.xml file --> Run As --> maven build:

The goal should be **mvn clean compile tomcat7:redeploy**

If you have configured the files properly, the code should be compiled and uploaded directly to your Tomcat server. The process will also generate a Swagger environment to test the backend. Open a browser and check if it works:

http://<your_tomcat_server>:8080/otpixel/ui (vue UI)

http://<your_tomcat_server>:8080/otpixel/doc (Swagger UI to test backend API)

6. PIXEL Dashboard and Notification

6.1. Overview

The Dashboard & Notifications is the component that has the capability of representing data stored in the IH through **meaningful combined visualizations in real time**. It also provides the capability to **send notifications based on the status of the data** received from sensors. Finally, this module **provides (aggregates and homogenises) all the UIs for the different functional blocks** (e.g. Operational Tools). The Dashboard component is divided in two subcomponents:

- *Frontend*: It offers a web application based on the VueJS Framework. This component exposes the UI through which the user interacts.
- *Backend*: It exposes all the services needed for the dashboard. Moreover, it connects to IdM (identity management service) to ensure users are authorized. It has a non-relational database and communicates with the **PIXEL Operational Tools** for the management of the containers. The **Backend** also has a component responsible of alerts.
 - *Backend of alerts*: It has a service that exposes a REST API to create different types of alerts. Once launched they are sent directly to the backend. It requires connection to the **PIXEL Information Hub**.

The Dashboard has a **Proxy** whose functionality is to maintain a single entry point to the dashboard. There are redirects for all the PIXEL components. All the services must be exposed through this component.

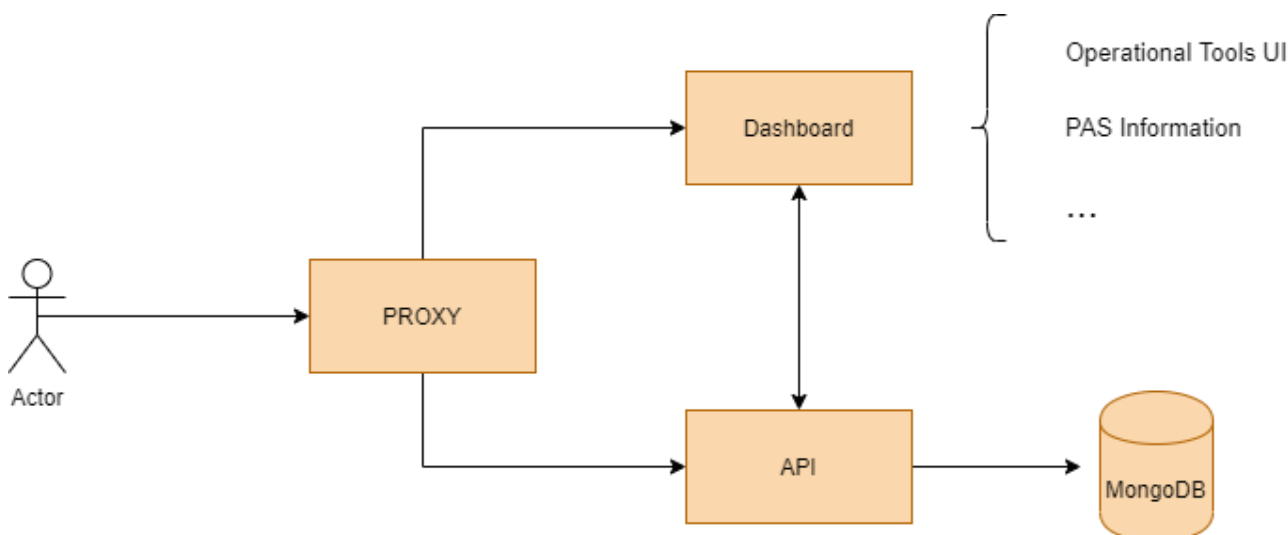


Figure 25: Dashboard diagram

The functional overview of the different options that the Dashboard has are:

- *Overview*: Section where the visualizations created by the end-user (and published) are shown. In this way, they are accessible as soon as the user accesses to the platform.
- *Views*: Component responsible for creating the different types of visualizations (**Gant diagram, Table**, etc.) of the data coming from the sensors.
- *Dashboard*: UI responsible for creating dashboards using visualizations created in the previous section.
- *Permission*: Component aligned with the **PIXEL Security & Privacy** module in order to fulfil all required security policies (e.g. authorization, authentication, roles, permission, etc.).
- *PAS Information*: UI to fill in the different entities (**resources, rules and supply chain**) needed as input for the PAS Model (**Port Activity Scenario**).
- *Map*: Component that will show geolocated data (sensors, devices, etc.) from the different ports.

- *Alerts*: Component responsible for **real-time monitoring of data** and triggering of alerts depending on their value.
- *Operational Tools*: User interface to access the functionalities of the Operational Tools.

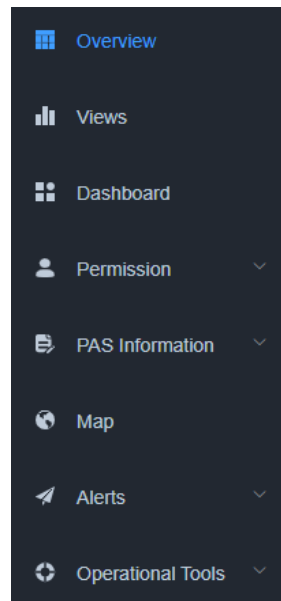


Figure 26: Dashboard menu options

6.2. Developer's guide

6.2.1. Introduction

For the development of the Dashboard & Notifications component of PIXEL it has been necessary to create two solutions:

1. *Client solution*. Main component of this module and accessible via web to show the results obtained in the platform. It has been developed in **Element UI (Open source framework based on Vue.JS)**. There is a complete guide of how to develop with this framework [here](#). The next picture depicts the more important features of this solution.

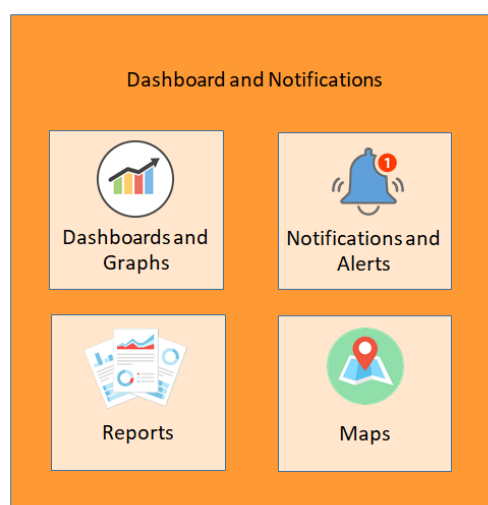


Figure 27: Dashboard features (client solution)

2. *Server solution*. **REST API** created to interact from the client solution with the different entities needed in certain processes (Visualizations, Dashboard, Alerts, etc.). It is a **CRUD API** from which access to

the No-SQL DB that is being used (**MongoDB**) are made. This API follows the **MVC pattern**. The element View would be the view of the client solution itself. Within the **API** there will be only **Model** and **Controller**. It has been developed in **Node.JS**.

6.2.2. Folder structure (Client solution)

The folder structure is marked by the chosen framework (**Element UI**). The following picture depicts the folders and a brief description of their functionality.

```
sh
├─ build                # build config files
├─ mock                # mock data
├─ plop-templates      # basic template
├─ public              # pure static assets (directly copied)
│  └─ favicon.ico      # favicon
│  └─ index.html       # index.html template
├─ src                 # main source code
│  └─ api              # api service
│  └─ assets           # module assets like fonts,images (processed by webpack)
│  └─ components       # global components
│  └─ directive        # global directive
│  └─ filters          # global filter
│  └─ icons            # svg icons
│  └─ lang             # i18n language
│  └─ layout           # global layout
│  └─ router           # router
│  └─ store            # store
│  └─ styles           # global css
│  └─ utils            # global utils
│  └─ vendor           # vendor
│  └─ views            # views
│  └─ App.vue          # main app component
│  └─ main.js          # app entry file
│  └─ permission.js    # permission authentication
├─ tests               # tests
├─ .env.xxx            # env variable configuration
├─ .eslintrc.js       # eslint config
├─ .babelrc           # babel config
├─ .travis.yml        # automated CI configuration
├─ vue.config.js      # vue-cli config
├─ postcss.config.js  # postcss config
└─ package.json       # package.json
```

Figure 28: Folder structure

6.2.3. Add new views

If you want to add a new view to the platform the developer has to create the view inside the views folder (see next figure), creating its container folder (alerts will be the container folder for the views related with this functionality).

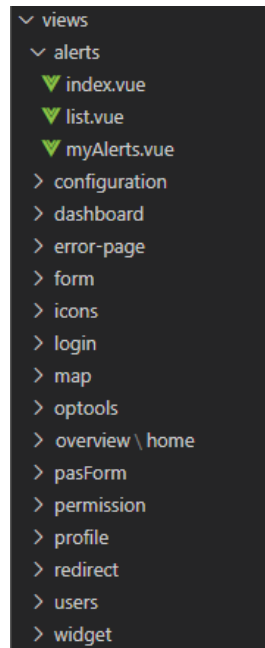


Figure 29: Example of views inside its container folder

Once the view has been created it is time to add the view to the file responsible for managing the routes in the platform (*router/index.js*).

In this file, in addition to the paths for the different menu entries, it is configured:

- **Navigability** between views. Even if the views don't have a menu entry.
- **Icons** of the views that have menu entry.
- **Nesting** of views within the same menu entry.

The next figure depicts the configuration in the *index.js* for the Map menu entry.

```
{
  path: '/map',
  component: Layout,
  children: [
    {
      path: 'map',
      component: () => import('@/views/map/index'),
      name: 'map',
      meta: { title: 'map',
              icon: 'international',
              affix: false }
    }
  ]
}
```

Figure 30: Example of how to fulfil a menu entry in *index.js* file

6.2.4. Internationalization

The PIXEL platform has the i18n configurations necessary to give support for the following languages:

- *English*
- *Spanish*

- *French*
- *Italian*
- *Greek*

The solution includes a folder called **lang** where the files for each supported language are located. To identify these languages, they are called according to their **ISO Language Codes**.

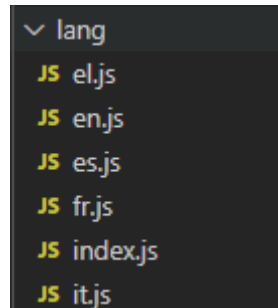


Figure 31: Files used to translate PIXEL (called according to their ISO Language Codes)

The entries in these files are structured in regions according to the functionality to which the tag to be translated belongs. The figure below depicts the entries within the region alerts for the English file (**en.js**, according to the nomenclature mentioned above).

```
alerts: {
  listOfAlerts: 'Alerts List',
  myAlerts: 'My Alerts',
  emptyAlerts: 'No Alerts Available',
  alertDescription: 'Description',
  alertTitle: 'Title',
  alertType: 'Type',
  alertSubscriptions: 'Subscription',
  alertRead: 'Read',
  alertFecha: 'Date',
  searchAlert: 'Search Alert'
}
```

Figure 32: English translations for alerts functionality

If PIXEL must **support a new language**, the developer should go to the `index.js` file (in the language folder) and generate the necessary files (according to the content of `index.js` file). Finally, it will be necessary to create a JavaScript file named with the ISO Code for the new language.

The **syntax of the tag** to be created will be different depending on its location within the view:

- *HTML Code*. The next figure illustrates the syntax of the label in this case. This is the title tag within the widget region.

```
<span>{{ widget.title }}</span>
```

Figure 33: Syntax, HTML Code

- *JavaScript Code*. Syntax is different in this case (see next figure). It is necessary to use the **‘\$t’ method** that injects the necessary code to translate the tags by going to the corresponding file and region to retrieve the text of the tag.

```

this.$message({
  type: pixelConstants.WARNING_MESSAGE_TYPE,
  message: this.$t('common.machinesNotImplemented')
})

```

Figure 34: Syntax, JavaScript Code

6.2.5. Notifications

The PIXEL Dashboard & Notifications module includes two options to show notifications or messages to the end-user (without the need to build a custom popup or modal dialog).

1. **Message:** Method used to **notify a message to the end-user**. For example, the result of a validation. There are several levels of notifications: **warning, error or successful**. It is also possible to indicate the time interval during which the message will be visible. The syntax of this type of notification is depicted in the next figure.

```

this.$message({
  type: pixelConstants.SUCCESS_MESSAGE_TYPE,
  message: this.$t('common.deleteSucced')
})

```

Figure 35: Message syntax

The previous figure will show a popup notifying that the delete action has been successful.

The next picture illustrates the appearance of this type of message. In that case, the result of a validation: a warning message.



Figure 36: Message appearance

2. **Notification:** Method used to show the result of an action (entity created, updated, etc.). There are different levels of messages depending on the result or priority of the action: warning, successful, error. The syntax of this type of message is depicted in the next figure.

```

this.$notify({
  title: this.$t('common.success'),
  message: this.$t('common.createdSuccessfully'),
  type: 'success',
  duration: 2000
})

```

Figure 37: Notification Syntax

The previous figure will show a popup notifying that the entity has been created successfully (see example below).

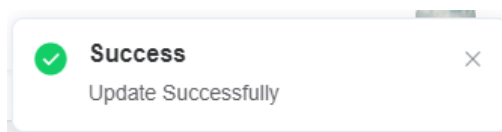


Figure 38: Notification appearance

6.2.6. Access to APIs

The Dashboard interacts with the APIs of other components (as well as with the API of the server solution). These requests are not made directly from the corresponding view where necessary.

There is a folder called API containing a JavaScript file for each of the entities or functionalities accessed via API.

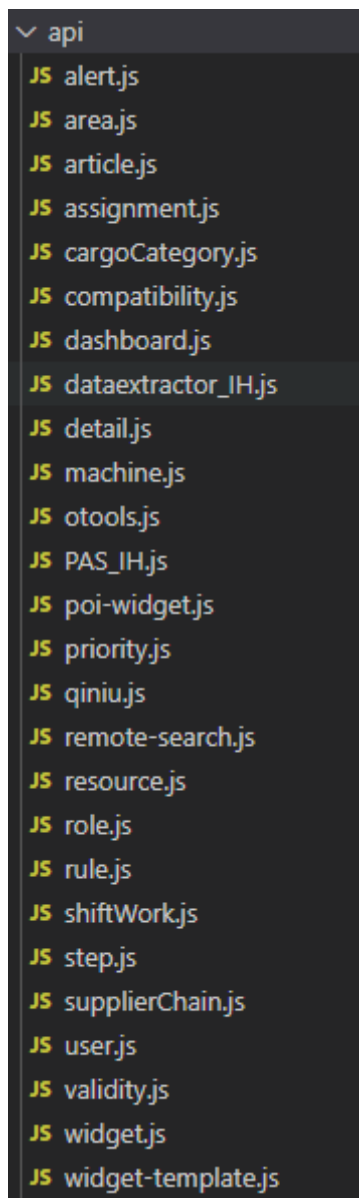


Figure 39: Content of the API folder

For example:

- *otools.js*. This is the file where the **PIXEL Operational Tools API endpoints** will be located.

```
import pixelConstants from '@/utils/constants' // import class for constants
import request from '@/utils/request_otools'

// #region Models

export function getModelsByType(type) {
  return request({
    url: pixelConstants.URL_PIXEL_GET_MODELS + '?type=' + type,
    method: pixelConstants.METHOD_GET,
    headers: {
      Accept: pixelConstants.Content_Type_Application_Json,
      Authorization: pixelConstants.API_KEY_SWAGGER_PIXEL_DATAMODEL
    }
  })
}

export function getModels(query) {
  return request({
    url: pixelConstants.URL_PIXEL_GET_MODELS,
    method: pixelConstants.METHOD_GET,
    headers: {
      Accept: pixelConstants.Content_Type_Application_Json,
      Authorization: pixelConstants.API_KEY_SWAGGER_PIXEL_DATAMODEL
    }
  })
}
}
```

Figure 40: OTools endpoints

- *dataextractor_ih.js*. This file centralises all the **IH dataextractor API endpoints**.

```
import pixelConstants from '@/utils/constants' // import class for constants
import request from '@/utils/request_IH'

export function dataExtractor(dataString) {
  return request({
    url: pixelConstants.URL_PIXEL_DATA_EXTRACTOR_IH + '?split=false',
    method: pixelConstants.METHOD_POST,
    data: dataString,
    headers: {
      Accept: pixelConstants.Content_Type_Application_Json,
      'Content-Type': pixelConstants.Content_Type_Application_Json
    }
  })
}
}
```

Figure 41: dataextractor API endpoints

- *PAS_IH.js*. It contains the necessary methods to complete the PAS (Port Activity Scenario) forms. This information is stored in the **PIXEL Information Hub** and in this case there is no specific API. Queries are performed directly using the **Elasticsearch REST API**.

```
import pixelConstants from '@/utils/constants' // import class for constants
import request from '@/utils/request_IH_PAS'

export function existIndex(indexName) {
  return request({
    url: '/' + indexName,
    method: pixelConstants.METHOD_HEAD,
    headers: {
      'Content-Type': pixelConstants.Content_Type_Application_Json
    }
  })
}

export function deleteIndex(indexName) {
  return request({
    url: '/' + indexName,
    method: pixelConstants.METHOD_DELETE,
    headers: {
      'Content-Type': pixelConstants.Content_Type_Application_Json
    }
  })
}
```

Figure 42: Elasticsearch endpoints

- *resource.js*. It contains the endpoints exposed by the **CRUD API** created in the server solution.

```
import request from '@/utils/request_dashboard'

export function resourceFetchList(query) {
  return request({
    url: '/resource',
    method: 'get',
    params: query
  })
}

export function resourceCreate(data) {
  console.log(data)
  return request({
    url: '/resource',
    method: 'post',
    data
  })
}

export function resourceUpdate(id, data) {
  console.log('Llego al fichero resource.js')
  return request({
    url: `/resource/${id}`,
    method: 'put',
    data
  })
}
```

Figure 43: Endpoints for resource entity

Each of these files has in common the import that is made in the first line. This imports the request class that will be used for each set of endpoints.

This request class contains:

- *Base url*: Used in the request for each endpoint.
- *Timeout configuration*: Can be different for each API.
- *Construction of the response*: Object for each request.

The next picture depicts an example of request class (for the Operational Tools endpoints in this case).

```
import axios from 'axios'
import pixelConstants from '@/utils/constants' // import class for constants

// create an axios instance
const service = axios.create({
  baseURL: pixelConstants.baseURL_PIXEL_DATAMODEL_API,
  timeout: 5000 // request timeout
})

// request interceptor
service.interceptors.request.use(
  config => {
    // do something before request is sent
    return config
  },
  error => {
    // do something with request error
    console.log(error) // for debug
    return Promise.reject(error)
  }
)

// response interceptor
service.interceptors.response.use(
  response => {
    // const res = JSON.stringify(response.data, null, 4)
    const res = response.data
    return res
  },
  error => {
    console.log('err' + error) // for debug
    return Promise.reject(error)
  }
)

export default service
```

Figure 44: Request class for Operational Tools endpoints

6.2.7. Add a new entity to the server solution

The server solution has been developed following the **MVC pattern (Model-View-Controller)**. Therefore, this pattern will be followed in the case of adding a new entity that will interact with the Dashboard.

The next figure depicts the folder structure of the server solution.

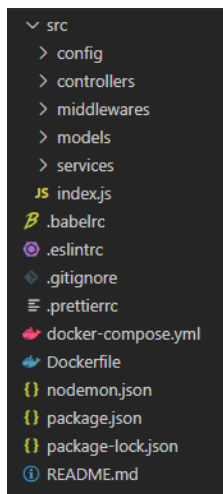


Figure 45: Structure of server solution

Among these folders, the following stand out:

- *controllers*. There will be controllers for each entity. **API entry point**. It is where the redirection of the method exposed to the internal method of our API is done. It makes use of the service classes.

```
import express from 'express';
import logger from '../config/winston';
import areaService from '../services/area.service';
// import resourceService from '../services/resource.service';

const router = express.Router();

// routes
router.get('/filterByResource/:idResource', getAreasFilterByResource);
router.get('/:id', getOne);
router.put('/:id', update);
router.post('/', create);
router.delete('/:id', deleteNode);
router.get('/', getAll);

export default router;

// Implementation

async function getAreasFilterByResource(req, res) {
  const { params: { idResource } = { idResource: null } } = req;

  try {
    const data = await areaService.getAll({ idResource });
    res.json(wrapperOk(data));
  } catch (error) {
    logger.error(error);
    return res.status(404).json({
      error: {
        code: 54910,
        message: 'An error occurred'
      }
    });
  }
  return null;
}
```

Figure 46: Controller file

- *services*. It performs the queries against the database for this purpose makes use of the models.

```
import Area from '../models/Area.model';

const service = {};

service.getOne = getOne;
service.getAll = getAll;
service.create = create;
service.update = update;
service.deleteOne = deleteOne;
service.getAllFiltered = getAllFiltered;

export default service;

// Implementation

async function getOne(id) {
  let data = [];
  try {
    data = await Area.findOne({ _id: id });
  } catch (error) {
    throw new Error(error);
  }
  return data;
}

async function getAll(query) {
  let data = [];
  try {
    data = await Area.find(query);
  } catch (error) {
    throw new Error(error);
  }
  return data;
}
```

Figure 47: Service file

- *models*. It is in these classes that the object to be used for our entity will be defined (properties, relations with other entities, etc.).

```
import { model, Schema } from 'mongoose';

const mySchema = new Schema(
  {
    idArea: {
      type: 'String'
    },
    idResource: {
      type: 'String'
    },
    label: {
      type: 'String'
    },
    type: {
      type: 'String'
    },
    owner: {
      type: 'String'
    },
    terminal: {
      type: 'String'
    }
  },
  {
    timestamps: true
  }
);

mySchema.index({ '$**': 'text' });

export default model('Area', mySchema);
```

Figure 48: Example of model

There is a configuration file (index.js) where the developer must add a few lines for each of the entities to be exposed. These lines are related to the controller of the entity. This is because the access point to the API is through the controller.

This file is formed by two blocks:

- *First block.* Where the **import of the controller's entity** is done.

```
import 'dotenv/config';
import express from 'express';
import cors from 'cors';
import logger from './config/winston';
import config from './config/config';
import util from './middlewares/util';
import auth from './middlewares/auth';
import widgetController from './controllers/widget.controller';
import widgettemplate from './controllers/widgettemplate.controller';
import notificationsubscription from './controllers/notificationsubscription.controller';
import notificationtemplate from './controllers/notificationtemplate.controller';
import notificationlog from './controllers/notificationlog.controller';
import dashboard from './controllers/dashboard.controller';
import resource from './controllers/resource.controller';
import area from './controllers/area.controller';
import machine from './controllers/machine.controller';
import rule from './controllers/rule.controller';
import cargoCategory from './controllers/cargoCategory.controller';
import shiftWork from './controllers/shiftWork.controller';
import priority from './controllers/priority.controller';
import assignment from './controllers/assignment.controller';
import validity from './controllers/validity.controller';
import supplierChain from './controllers/supplierChain.controller';
import step from './controllers/step.controller';
import detail from './controllers/detail.controller';
import compatibility from './controllers/compatibility.controller';
import connectToDb from './config/mongodb';
```

Figure 49: Section for the import of the controller's entity

- *Second block.* Where the **path of the entity** of that controller is indicated.

```
const app = express();
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(util.languageParser);
app.use(cors());

// Routes
app.use('/widget', auth.authUser, auth.isAuthorized, widgetController);
app.use('/widgettemplate', auth.authUser, auth.isAuthorized, widgettemplate);
app.use('/notification/subscription', auth.authUser, auth.isAuthorized, notificationsubscription);
app.use('/notification/log', auth.authUser, auth.isAuthorized, notificationlog);
app.use('/notification', auth.authUser, auth.isAuthorized, notificationtemplate);
app.use('/dashboard', auth.authUser, auth.isAuthorized, dashboard);
app.use('/resource', auth.authUser, auth.isAuthorized, resource);
app.use('/area', auth.authUser, auth.isAuthorized, area);
app.use('/machine', auth.authUser, auth.isAuthorized, machine);
app.use('/rule', auth.authUser, auth.isAuthorized, rule);
app.use('/cargoCategory', auth.authUser, auth.isAuthorized, cargoCategory);
app.use('/shiftWork', auth.authUser, auth.isAuthorized, shiftWork);
app.use('/priority', auth.authUser, auth.isAuthorized, priority);
app.use('/assignment', auth.authUser, auth.isAuthorized, assignment);
app.use('/validity', auth.authUser, auth.isAuthorized, validity);
app.use('/supplierChain', auth.authUser, auth.isAuthorized, supplierChain);
app.use('/step', auth.authUser, auth.isAuthorized, step);
app.use('/detail', auth.authUser, auth.isAuthorized, detail);
app.use('/compatibility', auth.authUser, auth.isAuthorized, compatibility);
app.listen(config.PORT, () => logger.info(`listen on ${config.PORT} port`));
```

Figure 50: Section responsible for the routing of the controllers

6.2.8. Add new visualizations

The **src/components** folder contains the various components of the dashboard client solution.

Among them, the widget folder stands out. This is where the different widgets/visualizations used to represent the information within the PIXEL Platform are created (displays associated with the execution of a model).

Therefore, in case you want to add a new type of visualization, this is where it should be done.

Currently, this folder is structured in the following subfolders:

- *Amcharts*: Visualizations created using this JavaScript library.
- *Echart*: Visualizations created using this JavaScript library.
- *Custom*: Customs visualizations without the specific use of any JavaScript library.
- *Mixins*: Folder with files that help representing the visualizations. In this case, the file responsible for resizing them (*resize.js*).

7. PIXEL Security

7.1. Overview

The main function of the security layer is to secure the access to the API of the other components from outside the platform and to provide a solution for identity management.

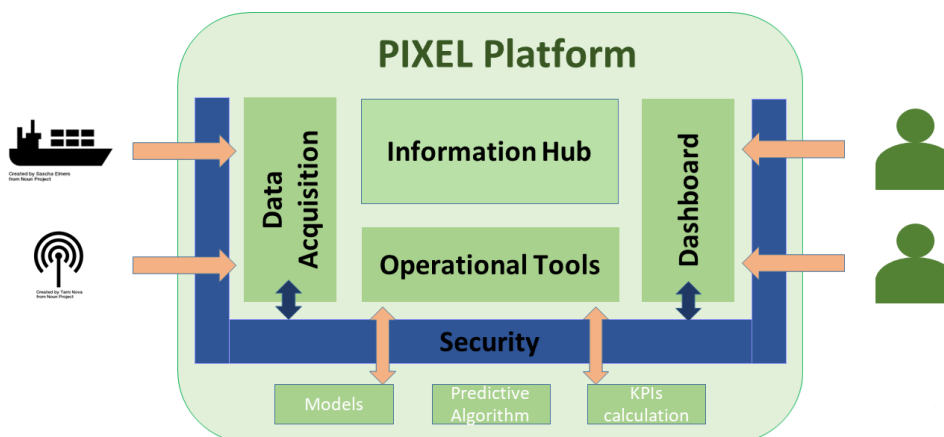


Figure 51. . Operational Tools - Architecture overview

The security layer not only secures the access to the NGSI Agents that exposed an API in the Data Acquisition Layer, but it also provides security to the dashboard UI to access the PIXEL’s API (Dashboard, Information Hub and Operational Tools). We rely on the FIWARE architecture and solution to implement those features in PIXEL, using the FIWARE Generic Enablers:

- KeyRock: The Identity Manager
- Wilma (PEP Proxy): The OAuth2 proxy that check the access
- AuthzForce: An XACML authorization solution

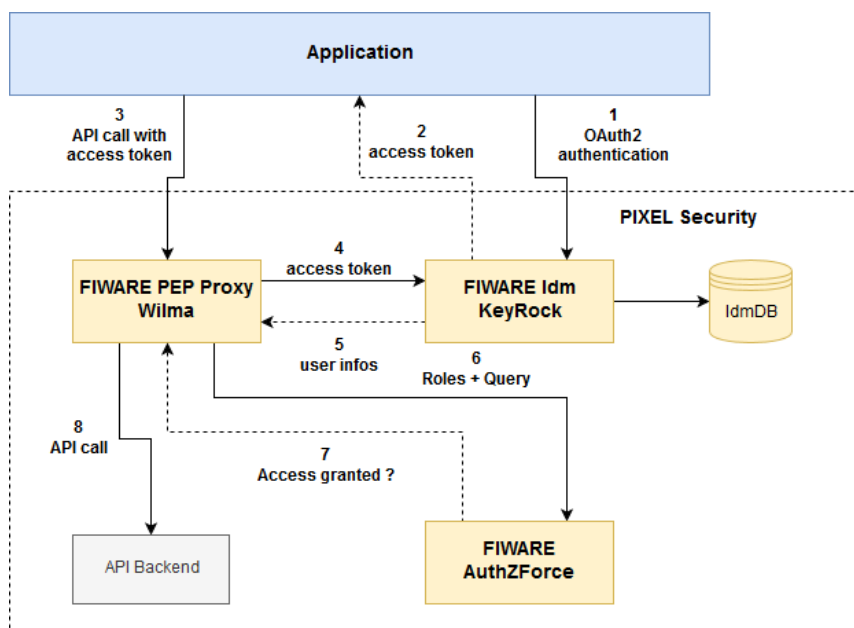


Figure 52. PIXEL security scheme

7.2. (REST) API

Wilma and AuthZForce interact directly with KeyRock to ensure that the access token provided by the user is granted with enough permissions to access the request URI. The PIXEL components have no need to access directly those components.

- AuthZForce provides a [SOAP API](#)
- Wilma doesn't provide an API and consumes AuthZForce and Keyrock ones.

7.2.1. Paths

The complete Identity Management API for KeyRock is fully [documented by FIWARE](#) and allows managing all the objects of the Identity Management models:

- Authentication.
- Manage Applications.
- Manage Users.
- Manage Organizations.
- Manage Roles.
- Manage Permissions.
- Manage IoT Agents.
- Manage Pep Proxies.

KeyRock also implements the standard OAuth2 protocol. The different API usages are [documented by FIWARE](#). Those APIs provide all the solution needed to generate a valid X-AuthToken to request an URI access.

In PIXEL most of the tokens are requested using the [Resource Owner Password Credentials Grant](#) method.

7.2.2. Models

The full API models are implemented in the MySQL Database that helps to show the relation between the different Identity Management objects.

A complete database structure is available on [github](#).

The main objects are:

- User: It has credentials, and can be assigned to Organization and Application.
- Organization: It represents a user's group. It can be assigned to Application.
- Application: We can assign one PEP Proxy to protect a backend. Users and Organizations have assigned Roles in the context of the application.
- Roles: A set of Permissions
- Permission: Defines an allowed request on an Application

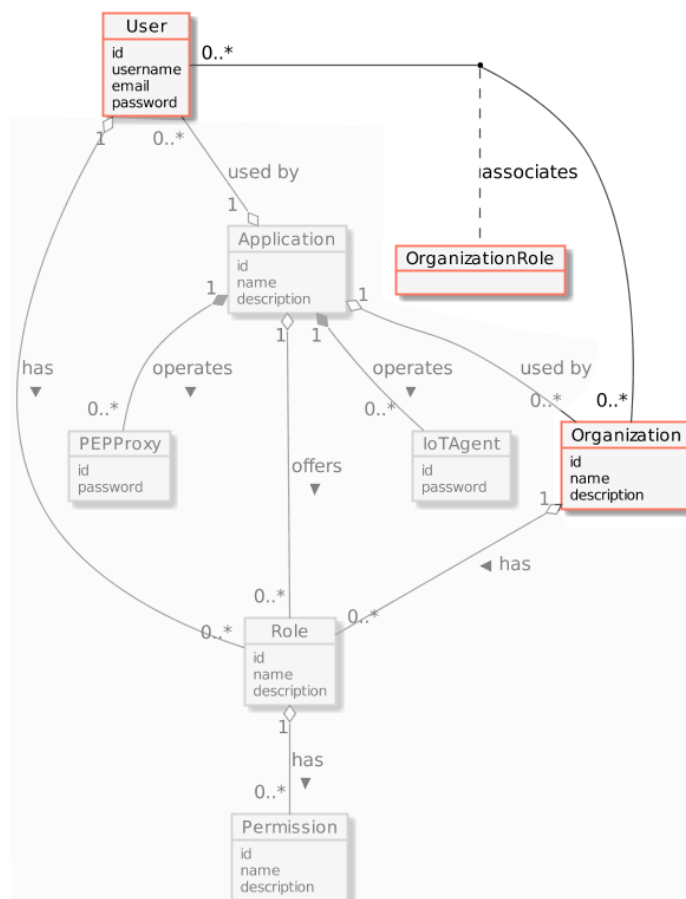


Figure 53. Related models for managing permissions

7.3. Developer's guide

The Security Layer relies on FIWARE components that are all Open Source, fully documented with an existing community managing them. It's always possible (as in any open source project) to propose evolution and corrections.

- Keyrock : <https://github.com/ging/fiware-idm>
- Wilma : <https://github.com/ging/fiware-pep-proxy>
- AuthZForce : <https://github.com/authzforce/core>

7.3.1. Potential extensions

The current version of KeyRock uses a MySQL Database to manage the objects. An evolution could be to propose a different driver database to allow KeyRock to work with an external IdM solution like LDAP to be more closely integrated in the port IT solution.

7.3.2. Additional notes

FIWARE propose documentation to easily integrate their solutions:

- KeyRock documentation guide : <https://fiware-idm.readthedocs.io/en/latest/index.html>
- Tutorials : <https://github.com/FIWARE/tutorials.Identity-Management>