# The ATLAS Metadata Interface (AMI) 2.0 metadata ecosystem: new design principles and features.

**J Odier[1], F Lambert[1], J Fulachier[1], on behalf of the ATLAS Collaboration.**

[1]Laboratoire de Physique Subatomique et de Cosmologie, Université Grenoble-Alpes, CNRS/IN2P3, 53 rue des Martyrs, 38026, Grenoble Cedex, FRANCE

E-mail: jerome.odier@lpsc.in2p3.fr

**Abstract**. ATLAS Metadata Interface (AMI) is a generic ecosystem for metadata aggregation, transformation and cataloging. Benefiting from 18 years of feedback in the LHC context, the second major version was recently released. This paper describes the design choices and their benefits for providing high-level metadata-dedicated features. In particular, the Metadata Querying Language (MQL) - a domain-specific language allowing to query databases without knowing the relation between entities - and on the AMI Web framework are described.

## 1. Introduction

The aim of this paper is to give an overview of the second major version of the ATLAS Metadata Interface (AMI), a mature software ecosystem dedicated to scientific metadata. The following presents the design principles and features of the most important sub-projects of the ecosystem. In particular, a focus is done on the Java core, the Metadata Querying Language (MQL) and the Web framework.

*1.1.* What is AMI?

Originally developed for the ATLAS experiment [1] at the CERN Large Hadron Collider (LHC), the second version of AMI is a generic ecosystem for metadata aggregation, transformation and storing. Benefiting from more than 15 years of feedback [2][3][4], it provides a wide array of tools (command line tools, lightweight clients) and Web interfaces for searching data by metadata criteria.

AMI was designed to guarantee scalability, evolutivity and maintainability. It perfectly fits the needs of scientific experiments in big data contexts.

*1.2.* A brief history of AMI

In 2000, primitive PHP [5] metadata bookkeeping software was developed for the ATLAS Liquid Argon (LAr) calorimeter. In 2002 the development of the first Java version was initiated and, since 2006, AMI is the official ATLAS dataset discovery tool.

At the end of 2014, it was decided to rewrite AMI from scratch in order to improve its maintainability and scalability. The project was split into independent sub-projects : this is the new ecosystem.
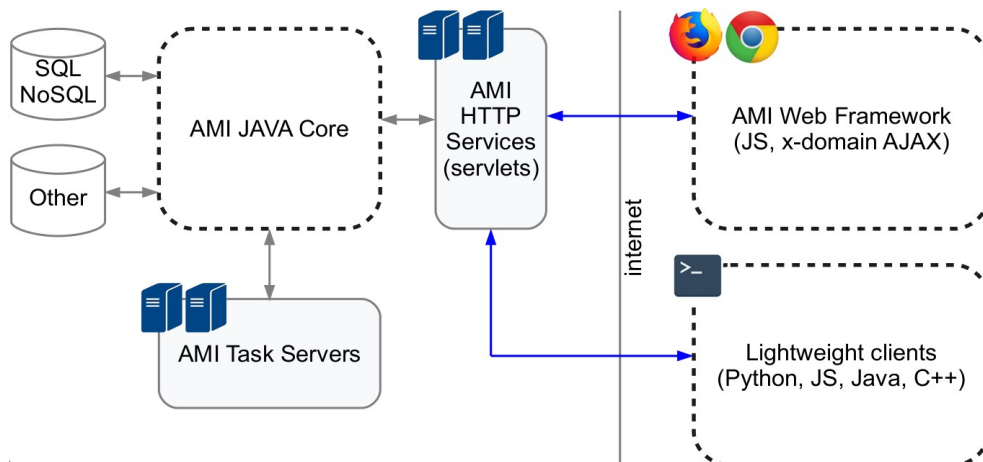
*1.3.* Overview of the AMI ecosystem

AMI is an ecosystem of software dedicated to metadata in a big data context. It consists of:

- AMI Java Core: the server-side core library. It contains high-level primitives for aggregating, processing and storing metadata and for searching data by metadata criteria.

- AMI Web Core: based on AMI Java Core and Java servlet, it provides a proprietary HTTP service and an alternative REST API for accessing the whole AMI content.

- AMI Task Server: a handy scheduler for running metadata aggregation / processing tasks from heterogeneous primary data sources in a distributed way.

- AMI Web Framework: a Web framework for developing metadata-oriented applications. It is designed to be used with the AMI Web Core or in a standalone way.

- A set of lightweight clients for accessing the AMI HTTP service from anywhere (C, C++, Java, JavaScript, Python, …).

Figure 1. shows an overview of the AMI ecosystem.



**Figure 1.** Overview of the AMI ecosystem.

## 2. AMI Java Core

The AMI Java Core is the most important part of the AMI ecosystem. Server-side, it implements high-level primitives for aggregating, processing and storing metadata and for searching data by metadata criteria. It is developed in Java 8 and follows an n-tier architecture in order to improve the global maintainability by isolating each sub-systems.

The sub-systems can be grouped into two layers: i) the "command" layer, the entry point of AMI Java Core, ii) the "metadata" layer, for accessing and manipulating data and metadata in heterogeneous databases (SQL, NoSQL) or from files, locally or remotely.

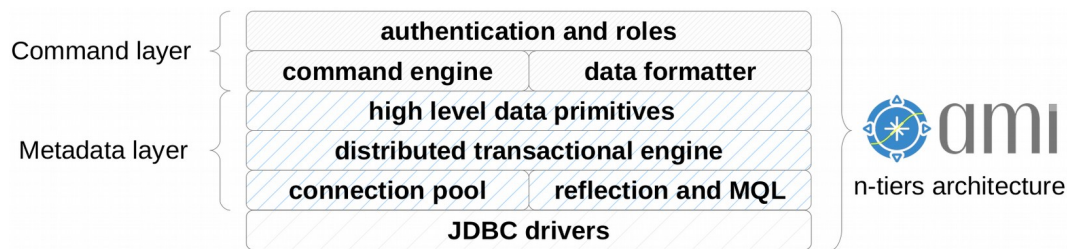Figure 2. shows an overview of AMI Java Core.

**Figure 2.** Overview of AMI Java Core.

## 2.1. *"Command" layer*

The "command" layer is the standard AMI entry point. Querying AMI consists in sending an AMI command to AMI Java Core.

Before executing an AMI command via the HTTP service, an authentication sub-system makes sure the user is registered. It supports login/password, X.509 certificates and Single Sign-On (SSO) methods. Then, an authorization sub-system controls if the user is allowed to execute the AMI command. In case of database access, it is possible to check authorizations with catalog (database), table, row or field granularities.

Basically, an AMI command is a Java class, inheriting from **net.hep.ami.command.AbstractCommand**, benefiting from the AMI "metadata" layer (see next paragraph) and returning a valid XML result. An XML formatter makes it possible to optionally generate TEXT, CSV or JSON outputs.

AMI provides a large set of predefined commands for administrating the service, for performing database operations, for aggregating metadata, and so on. Moreover, AMI is easily extendable by developing new commands in order to fit the experiment needs. For instance, for the ATLAS experiment, there are: "GetDatasetInfo" (for getting the metadata associated with a given dataset), "ListDataset" (for listing datasets by metadata criteria), etc.

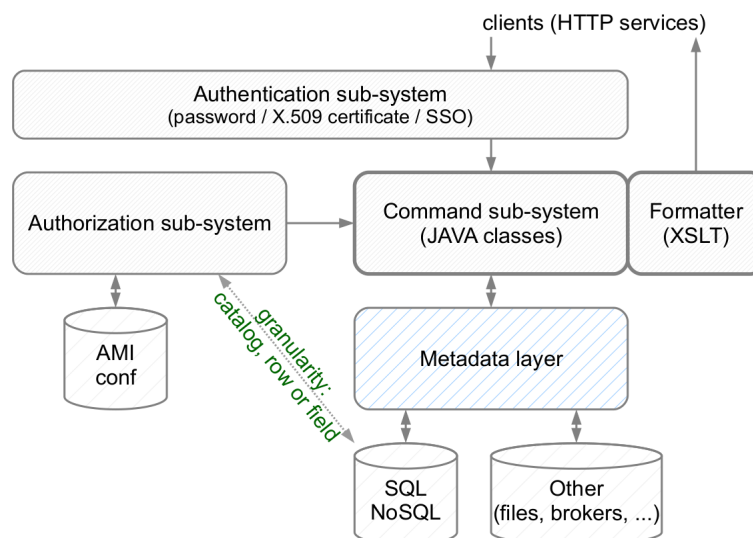Figure 3. shows an overview of the "command" layer of AMI Java Core.



**Figure 3.** Overview of the "command" layer. An AMI command is a java class, inheriting from **net.hep.ami.command.AbstractCommand**, and benefiting from the AMI "metadata" layer.
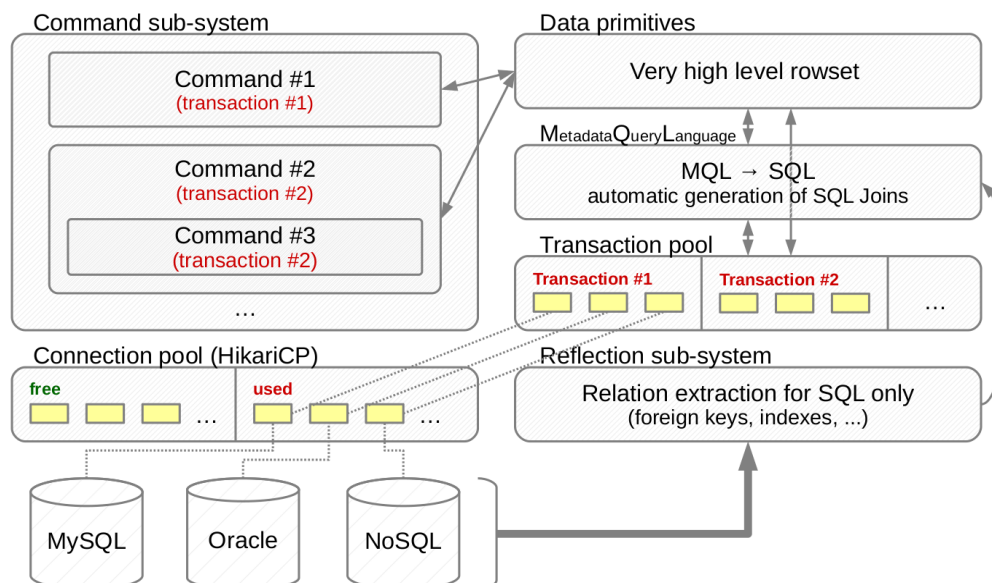
## 2.2. *"Metadata" layer*

The "metadata" layer provides business logic which hides the details of the underlying data sources and optimizes perfomance. Is composed of the following set of sub-systems:

- The first one is the Java Database Connectivity (JDBC) [6] connection pool, a cache of ready-to-use database connections enhancing the global performance of AMI when executing database queries. It is based on HikariCP [7], a quasi "zero-overhead" production-quality connection pool.

- The second sub-system is the JDBC transaction pool. It permits grouping JDBC connections in transactions. When a transaction is committed (resp. rolled back), each JDBC connection is committed (resp. rolled back) and in case of error, each JDBC connection is rolled back.

- The third is the reflection sub-system. Using the reflection capabilities of the Relational Database Management System (RDMS) JDBC drivers, it extracts both foreign keys and indexes and internally builds a global graph of relation between tables and databases.

- The fourth sub-system is the implementation of the Metadata Query Language (MQL). It permits performing database queries without (precisely) knowing relations between tables. It is based on the reflection sub-system. MQL is briefly described in the next paragraph.

- The last is the "data primitives" sub-system. It consists in a set of high-level primitives for querying databases (SQL, MQL, NoSQL); for manipulating the resulting rowset data (metadata, stream, XML serialization, etc.); for manipulating XML, CSV, JSON files locally or remotely (via SSH, HDFS, etc.).
  In AMI Java Core, the "command" layer interacts with the "metadata layer" via the "data primitives" sub-system.

Figure 4. shows an overview of the "metadata" layer of AMI Java Core.



**Figure 4.** Overview of the "metadata" layer with: the connection pool; the transaction pool; the reflection sub-system; the Metadata Query Language (MQL) and the "data primitives" sub-system.

*2.3. Metadata Query Language*

One of the main added value features of AMI is the Metadata Query Language (MQL). Initially proposed by gLite [8], a middleware project for grid computing at LHC experiments, the specification was extended by the AMI team.

MQL is a Domain-Specific Language (DSL) for querying Relational Database Management System (RDMS). But unlike SQL, MQL is able to automatically build joins. So it permits performing queries without (precisely) knowing relations between tables. In other words, it means that MQL only deals with metadata entity names while SQL uses a catalog/table/field paradigm.

The MQL implementation in AMI is based on the reflection sub-system described in the previous paragraph. It is cycle safe because the language permits including or excluding path fragments in the relationship graph. After processing, MQL queries are internally converted to optimized SQL queries and are executed by the adequate JDBC driver. Figure 5. shows an example of MQL query and the corresponding SQL query.
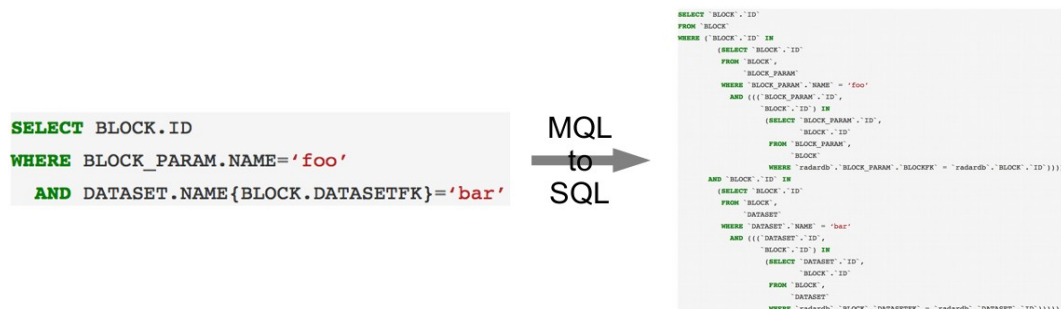


**Figure 5.** Conversion of a Metadata Query Language (MQL) query to a complex Structured Query Language (SQL) query. *{BLOCK.DATASETFK}* means that in case of ambiguity resolving the relation path, the MQL engine has to choice the one that goes through the foreign key *BLOCK.DATASETFK*.

## 3. AMI Task Server

The AMI ecosystem includes dedicated software for executing: i) metadata extraction tasks (from primary data sources), ii) metadata processing tasks, iii) generalist tasks. AMI Task Server is a stand-alone service with no dependency to AMI Java Core. It can be used in a distributed way where instances are synchronized with each other.

*3.1. Features*

AMI Task Server implements a priority lottery scheduler. This probabilistic algorithm solves the problem of starvation giving each process a non-zero probability of being selected at each scheduling operation. In return, AMI Task Server is not real-time. This characteristic is compatible with the purpose of collecting metadata because delays, up to minutes, are tolerated for being up-to-date when primary data sources change. Moreover, AMI Task Server implements a mutual exclusion mechanism for tasks sharing the same exclusion tags.

Each task is parameterized by a priority (0 is the highest priority), a time step (minimum delay between two consecutive executions) and a list of arbitrary exclusion tags. It is possible to run one shot tasks. Tasks can be developed in Java (using or not using AMI Java Core) or in any other language.

A standalone Web interface is provided for administrating a cluster of AMI task servers.

**4. AMI Web Framework**

The AMI ecosystem has a dedicated framework, AMI Web framework (AWF) [9], for developing metadata-oriented Web applications. AWF is based on standard technologies: JavaScript 6 (transpiled to JavaScript 5), JQuery [10], HTML 5, CSS 3, Twitter Bootstrap 4 [11], JSPath [12], and a homemade JavaScript implementation of the Twig template engine [13]. It is designed to be used with the AMI Web Core or in a standalone way.

*4.1.* AMI Web Framework features and patterns

AWF emulates the namespace, interface and class paradigms in JavaScript 5. It manages the life cycle of arbitrary resources (like CSS, JS, JSON, …) and introduces the control (aka. widget or graphical component) and application paradigms. An application is generally built by assembling controls. Each control or application follows a Model View Controller (MVC) pattern in order to simplify developments by decoupling data model, visualization and application specific code. The "model" part is the AMI commands, the "view" part is composed of Twig templates and the "controller" part is composed of a class inheriting form **ami.SubApp** or **ami.Control**.

AWF uses Asynchronous JavaScript And XML (AJAX) for interacting with AMI Web Core in a secured and authenticated way. AMI Web Core is configured to allow requests using the Cross-Origin Resource Sharing (CORS) [14] mechanism. It gives AWF controls the capability of accessing the AMI content from everywhere. More particularly, it is possible to embed AMI controls in external Web pages, personal pages, wikis, etc. See details in a separate conference proceeding [15].

AWF also provides an authentication sub-system (using login/password, X509 certificates or Single Sign-On (SSO)), a URL router, a short URL sub-system and a collection of wizards for generating control or application skeletons. Developing a new control or a new application is quite easy.

It is possible to embed AWF in an existing Web service using the AMI Mini Java/Python/PHP libraries.

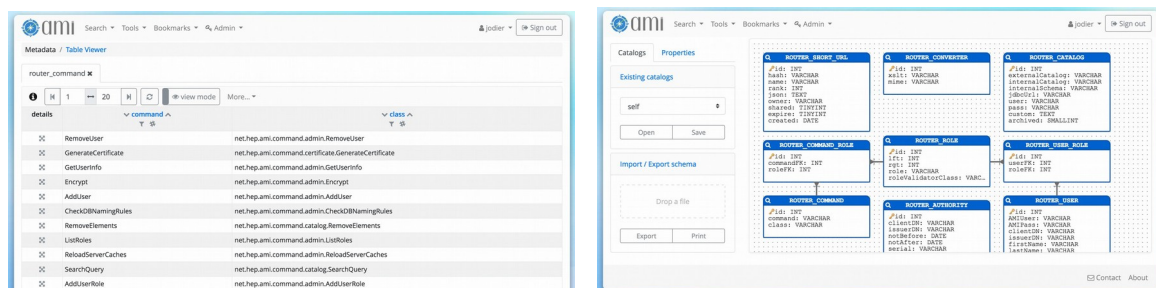Figure 6. shows an example of two web applications based on AWF.



**Figure 6.** At left, an AWF application with a data table. At right, the AWF database schema explorer.

*4.2. Default controls and applications*

AWF provides a set of standard controls:
- Dialog boxes
- Controls for searching (Google-like Search, Criteria Search, etc.)
- Controls for displaying (Schema Viewer, Graph Viewer, Tab, Table, Element Info, etc.)
- Controls for annotating entities (WhiteBoard, etc.)

AWF also provides a set of standard applications:
- AMI command interpreter
- Admin Dashboard and Monitoring
- Basic Content Management System (CMS)
- Schema Viewer, Table Viewer, Simple Search, Criteria Search, Search Modeler, etc.

### 4.3. Screenshots

Figure 7. shows another example of four web applications based on AWF.
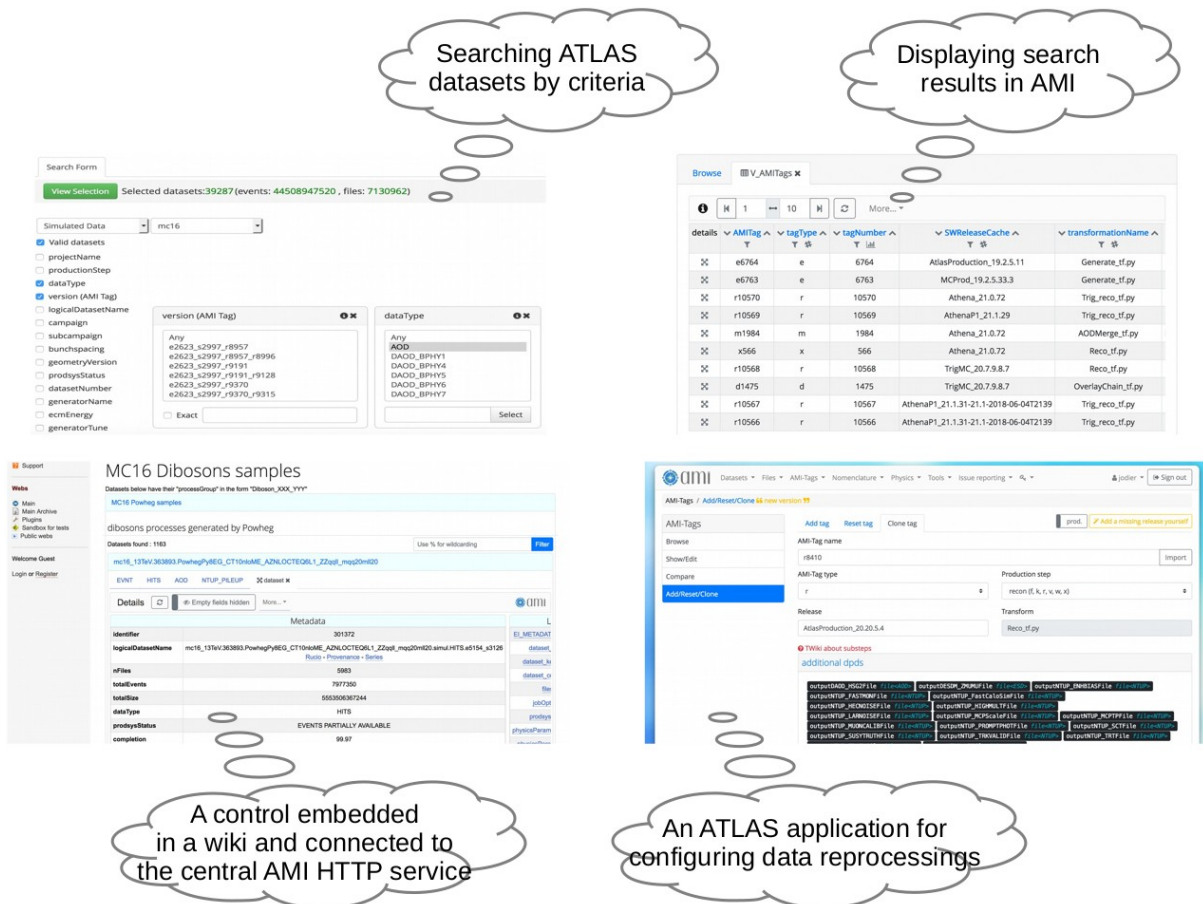


**Figure 7.** At top left, an AWF application with the standard search control. At top right, an AWF application with the standard table control. At bottom left, a custom AMI control, embedded in an external wiki page and connected to the central AMI HTTP service. At bottom right, a custom Web AWF application used by the ATLAS experiment for configuring data reprocessings.

## 5. Conclusion and outlooks

AMI is a well-established and mature metadata ecosystem, proposing services and Web interfaces to more than 2000 active users in the ATLAS collaboration. After four years of development, the second version of the ecosystem is ready for production. It perfectly fits the needs of modern physics experiments in a big data context.

The ecosystem has been used for about one year by the Rosetta/Philae collaboration [16][17] and the migration from AMI version 1 to AMI version 2 is ongoing for the ATLAS experiment.

**References**
[1] ATLAS Collaboration 2008 The ATLAS Experiment at the CERN Large Hadron Collider *JINST* **3** S08003 doi:10.1088/1748-0221/3/08/S08003
[2] Odier J, Aidel O, Albrand S, Fulachier J, Lambert F 2015 Evolution of the architecture of the AT-LAS Metadata Interface (AMI). Proceedings of the 21st International Conference on Computing in High Energy and Nuclear Physics (CHEP2015) *J. Phys.: Conf. Ser. 64 042040*
[3] Odier J, Aidel O, Albrand S, Fulachier J, Lambert F 2015 Migration of the ATLAS Metadata Inter-face (AMI) to Web 2.0 and cloud Proceedings of the 21st International Conference on Computing in High Energy and Nuclear Physics (CHEP2015) *J. Phys.: Conf. Ser. 664 062044*
[4] Fulachier J, Aidel O, Albrand S, Lambert F 2013 Looking back on 10 years of the ATLAS Meta-data Interface. Proceedings of the 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013) *J. Phys.: Conf. Ser.* **513** 042019 doi:10.1088/1742-6596/513/4/042019
[5] PHP: http://www.php.net/
[6] JDBC: https://www.oracle.com/technetwork/java/overview-141217.html
[7] HikariCP: https://brettwooldridge.github.io/HikariCP/
[8] gLite: http://grid-deployment.web.cern.ch/grid-deployment/glite-web/
[9] AWF: https://ami.in2p3.fr/app/?subapp=document&userdata=api.html
[10] JQuery: http://jquery.com/
[11] Twitter Bootstrap: http://getbootstrap.com/
[12] JSPath: https://github.com/dfilatov/jspath/
[13] AMI-Twig.js: http://ami.web.cern.ch/ami/twig/
[14] CORS: https://www.w3.org/TR/cors/
[15] Lambert F, Odier J, Fulachier J 2018 Broadcasting dynamic metadata content to external web pages using AMI (ATLAS Metadata Interface) embeddable components. Proceedings of the 23rd In-ternational Conference on Computing in High Energy and Nuclear Physics (CHEP2018) *J. Phys.: Conf. Ser.*
[16] Rosetta/Philae experiment: http://rosetta.esa.int/
[17] Rosetta/Philae experiment: https://rosetta.jpl.nasa.gov/