D2.1.1 SPECIFICATION OF INGESTION SERVICES

Advanced Search Services and Enhanced Technological Solutions for the European Digital Library

Grant Agreement Number: 250527

Funding schema: **Best Practice Network**

Deliverable D2.1.1 WP2.1

Deliverable
V.1.0 – 13 May 2011
Document. ref.: ASSETS.D2.1.1.CNR.WP2.1.V1.0

Programme Name: .... ICT PSP
Project Number: 250527
Project Title:.................................. ASSETS
Partners:.. Coordinator: ENG (IT)
Contractors:
Document Number: D2.1.1
Work-Package:............................... WP2.1
Deliverable Type: .... Deliverable
Contractual Date of Delivery: 31-March-2011
Actual Date of Delivery: 13-May-2011
Title of Document: .... Specification of Ingestion Services
Author(s): .... Stefano Baccianella, Andrea Esuli, Diego Marcheggiani, Fabrizio Sebastiani (CNR);
.... Sergiu Gordea (AIT)
Approval of this report ....

Summary of this report: .... see Executive Summary

History:.. see Change History

Keyword List: .... ASSETS, Ingestion, Classification, Extraction, Cleaning, Metadata

Availability .... This deliverable is:
X public
### limited to ASSETS consortium distribution
### limited to EU Programme distribution
### restricted
### internal

Change History

Version	Date	Status	Author	Description
0.1	02/05/11	Draft	AE, SB, DM, FS (CNR)	Initial draft
0.2	11/05/11	Draft	AE, SB, DM, FS (CNR)	Incorporating feedback from reviewers
1.0	13/05/11	Final	AE, SB, DM, FS (CNR)	Final release

Table of Contents

Executive Summary

This document contains a specification of the services to be developed within tasks “T2.1.1 Metadata cleaning”, “T2.1.2 Knowledge extraction”, and “T2.1.3 Metadata classification”, all of them under the responsibility of CNR. For each activity, a scientific analysis and a detailed specification of the API level is provided[1].

1 Part of the content of this deliverable already appears in Deliverable 2.0.4 “The ASSET APIs”.

1. Introduction

The planned objective of WP2.1 is to implement a service for the enrichment of metadata records that accounts (i) for the removal of various sources of noise from these records (“metadata cleaning”), (ii) for the automatic identification and annotation, within metadata records, of text strings that denote relevant entities (“knowledge extraction from metadata records”), and (iii) for the automatic classification of the metadata records according to a set of categories, possibly organized into a taxonomy, relevant for the domain (“metadata classification”).

This task is made complex by the presence, within the ASSETS consortium and within Europeana, of different content providers, concerned with different types of content, and whose content is described by metadata records expressed in different languages. There is a need thus to implement the above-mentioned services in a way that addresses this diversity of content providers, content types, and languages, and in a way that allows possible new content providers, with new content types described by metadata expressed in new languages, to be also addressed with minimum additional effort.

As a consequence, the objective is to implement these services according to a supervised learning methodology. Essentially, this means that a new content provider will be able to set up a system for enriching its own metadata by providing to the system a “training” set of enriched metadata records. The system would use these enriched metadata records as indications, or examples, of what enriching metadata records from this content provider means, and would then generate an “automatic enricher” of metadata records from this content provider. Essentially, this mechanism allows to set up automatic metadata enrichers for any type of content provider, any type of content, and any language, provided adequate training sets of manually enriched metadata records are given as input.

This supervised learning metaphor underlies all three services tackled within WP2.1. However, its algorithmic realization for the different services is different, since the individual tasks are different in nature. For instance, T2.1.3 is a task that purports to enrich the metadata record as a whole, by classifying it, and will thus be tackled via automatic text classification technologies. Instead, T2.1.2 is a task that purports to enrich the metadata record not by annotating the record in its entirety, but by annotating individual sequences of words within the record, and will thus be tackled via automatic sequence learning (“information extraction”) technologies.

The next section will give a concise scientific introduction to these tasks and to the algorithms that we are going to use to solve them.

2. Scientific background

2.1 Metadata Cleaning

In the ASSETS proposal, T2.1.1 has to do with the removal of noise from metadata records, where by "noise" we mean any kind of spurious text that may be present in the metadata record. Spurious text may be present for a variety of reasons, ranging from erroneous typing on the part of the person who performed the data entry, to erroneous conversion from a source metadata format to a target metadata format. The original goal of T2.1.1 was to tackle this problem via supervised learning, using "information extraction" technology (see Section 2.2 for a definition) of a type similar to the one used in T2.1.2 "Knowledge extraction from metadata records". The rationale for using information extraction technology was that noise in the metadata record field would consist of text strings properly contained in the individual fields of the record, and that as such would lend themselves to automatic annotation in the same way as other types of text strings properly contained in the individual fields of the record, such as person and organization names, do lend themselves to such automatic annotation (see T2.1.2). In the intention of the proposers this approach to metadata cleaning based on supervised learning was meant to be a radically novel piece of research, since we do not know of any other research work that applies supervised learning, or information extraction, technology to (meta)data cleaning.

Unfortunately, the goals of T2.1.1 have turned out to be very difficult to achieve in practice, since feedback received from the content providers has revealed that the amount of noise in the metadata is much smaller than it had been foreseen at the time of writing the proposal. This is especially problematic for an approach to this task based on supervised learning, since a supervised learning approach would have required a sizeable amount of noisy metadata records (to be used as training and test data) in which the content providers had manually marked the noisy substrings and had clearly indicated what the corrective action had been (e.g., deletion, replacement with another correct substring, etc.). It was in this light that the partner responsible for T2.1.1 (CNR) provided all the content providers of the ASSETS consortium with a set of guidelines on how to annotate metadata records that be used as training and test data for T2.1.1, and requested each content provider to provide a sizeable amount of such annotated metadata records. Unfortunately, no content provider was really able to identify a sizeable enough amount of noisy metadata records (only one CP provided 12 such records, and another CP provided 1), due to the fact that content providers in the ASSETS consortium already enforce quality checking protocols that ensure that the metadata records they produce are relatively noise-free. In other words, metadata cleaning turned out to be a much less important problem than the proposers had thought among the content providers in the ASSETS consortium.

As a consequence, most of the effort in T2.1.1 has been moved to T2.1.2 and T2.1.3, which have required much more effort than it had initially been foreseen due to the complexity of the Europeana infrastructure.

The design and implementation activities were planned to be executed in parallel with the CPs' manual annotation activities; as a result, in the first year some effort has been devoted to task T2.1.1 anyway. This has resulted in the definition of the API for the service and its implementation at the interface level, as documented in Section 3 of this deliverable.

In the 2nd year of the project we will also try to refocus this task as an enrichment and normalization task (via the use of dictionaries and authority files); this means, e.g., recognizing certain types of expressions of special significance (e.g., temporal expressions), normalizing them and/or linking them to authority file entries. We are currently investigating the existence of training and test data that are necessary to carry out these tasks.

2.2 Knowledge Extraction from Metadata Records

2.2.1 Knowledge extraction from metadata records based on conditional random fields

T.2.1.2 has to do with automatically annotating the text of which metadata records consist of, by tagging specific substrings of this text according to a pre-specified set of tags that denote concepts of interest in the domain the metadata records, and the corresponding content, refer to. This task is usually referred to as *information extraction* (IE), or *knowledge extraction,* in the literature [Be.Dov and Feldman, 2010, McCallum 2005, Sarawagi 2008]. In other words, information extraction is the discipline concerned with the extraction of natural language expressions from free text, where these expressions instantiate concepts of interest in a given domain; if there are *n* different concepts of interest, information extraction is a a bit like highlighting the text via *n* highlight markers of *n* different colours. For instance, given a corpus of job announcements, one might want to extract from each announcement the natural language expressions that describe the nature of the job, the promised annual salary, the job location, etc. Another very popular instance of IE is searching free text for *named entities*, i.e., names (or mentions) of persons, locations, geopolitical organizations, and the like [Nadeau and Sekine, 2007]. Put yet another way, IE may be seen as the activity of populating a structured information repository (such as a relational database, where "job", "annual salary", "job location" are attributes) from an unstructured information source such as a corpus of free text. As such, IE is important for enriching digital libraries by making implicit semantics explicit, and is a prerequisite for *concept normalization*, i.e., the linking of the mention of a concept to an entry of a controlled vocabulary so that different linguistic manifestations of the same concept link to the same controlled vocabulary entry.

There are two main approaches to designing an IE system. The former is the *rule- based approach*, which consists in manually writing a set of rules which relate natural language patterns with the concepts to be extracted from the text. This approach, while potentially effective, is too costly, since it requires a lot of human effort for writing the rules, which must be jointly written by a domain expert and a natural language engineer. In T2.1.2 we follow the alternative approach, which is based on *supervised machine learning*. According to this approach, a general-purpose learning software learns to relate natural language patterns with the concepts to be instantiated, from a set of manually annotated free texts, i.e., texts in which the instances of the concepts of interest have been marked by a domain expert. The most important advantage of this approach is that the human effort required for annotating the texts needed for training the system is much smaller than the one needed for manually writing the extraction rules. After all, this is just a manifestation of the fact, well-known in the cognitive sciences, that defining a concept *intensionally* (i.e., specifying a set of rules for recognizing the instances of this concept – say, a set of rules for recognizing red objects) is cognitively much harder for a human that defining the same concept *ostensively* (i.e., pointing to a set of instances of the concept – say, pointing to a set of red objects). A consequence of the machine learning approach is that a system for information extraction may be easily updated to reflect new needs, such as e.g., the addition of a new concept to the set of concepts to be identified, or the replacement of the concept set with a completely different concept set. While the rule-based approach would require, in these cases, the manual update of the extraction rules via the joint work of a knowledge engineer and a domain expert, the machine learning approach just requires the provision of new training examples annotated according to the new concepts of interest. In T2.1.2 this is extremely advantageous, since the ASSETS consortium (and, *a fortiori*, the set of Europeana content providers) harbours a variety of different content providers, working on different types of content (and thus likely requiring the annotation of text according to different concepts of interest) and describing this content via metadata records formulated in different languages. In the rule-based approach this diversity would entail the need to tackle each combination of <content provider + type of content + language> individually, by manually writing rules for each such combination, while in the machine learning approach each such combination may be tackled by simply providing appropriate training examples.

In the following sections we will first give a formal definition of information extraction, a brief description of “conditional random fields”, the supervised learning algorithm that we have adopted for T2.1.2. Conditional random fields have widely been studied, and are widely used in information extraction applications, ranging from named entity recognition [Zeng et al., 2009], to the analysis of medical reports [Esuli et al., 2011], to medical record anonymisation [Szarvas et al, 2007], and even word hyphenation [Trogkanis and Elkan, 2010]. We will then give a detailed description of the evaluation protocol that we will follow in order to ascertain how accurately the system performs on the metadata records of the ASSETS and Europeana content providers.

A formal definition of information extraction

Let a text $U = \{t_1 < s_1 < ... < s_{n-1} < t_n\}$ consist of a sequence of *tokens* (i.e., word occurrences) $t_1, ..., t_n$ and *separators* (i.e., sequences of blanks and punctuation symbols) $s_1, ..., s_{n-1}$, where ”<” means “precedes in the text”. We use the term *textual unit* (or simply *t-unit*), with variables $u_1, u_2, ...$, to denote either a token or a separator. Let $C=\{c_1, ..., c_m\}$ be a predefined set of *tags* (aka *labels,* or *classes*), or *tagset*. Let $A=\{\sigma_{11}, ..., \sigma_{1k}, ..., \sigma_{m1}, ..., \sigma_{mk}\}$ be an *annotation* for *U*, where a segment σ_{ij} *for U* is a pair (st_{ij},et_{ij}) composed of a start token $st_{ij} \in$ U and an end token $et_{ij} \in U$ such that $st_{ij} \leq et_{ij}$ (“≤” obviously means “either precedes in the text or coincides with”). Here, the intended semantics is that, given segment $□_{ij}=(st_{ij}, et_{ij}) \in$ A, all t-units between st_{ij} and et_{ij}, extremes included, are tagged with tag c_i.

Given a universe of texts $\mathcal{U}$ and a universe of segments $\mathcal{A}$, we define *information extraction* (IE) as the task of estimating an unknown target function $□ : \mathcal{U} \times C \rightarrow \mathcal{A}$, that defines how a text $U \in \mathcal{U}$ ought to be annotated (according to a tagset C) by an annotation A $\in \mathcal{A}$; the result $□(□)$$: \mathcal{U} \times C \rightarrow \mathcal{A}$ of this estimation is called a tagger. Consistently with most mathematical literature we use the caret symbol □() to indicate estimation. Note that the notion of IE we have defined allows a given t-unit to be tagged by more than one tag, and is thus dubbed multi-tag IE. The multi-tag nature of our definition essentially means that, given tagset $C=\{c_1, ..., c_m\}$, we can split our original problem into m independent subproblems of estimating a target function $□_i : \mathcal{U} \rightarrow \mathcal{A}_i$ by means of a tagger $□(□_i) : \mathcal{U} \rightarrow \mathcal{A}_i$, for any $i \in \{1, ..., m\}$. Likewise, the annotations we will be concerned with from now on will actually be *c*-annotations, i.e., sets of c_i*-segments* of the form $A_i = \{\sigma_{i1}, ..., \sigma_{ii}\}$. Hereafter we will often drop the prefix c_i- when the context makes it implicit.

Conditional random fields

As a learning algorithm we have used conditional random fields} [Lafferty et al, 2001, Sutton and McCallum, 2007]. Conditional random fields are graphical models that model a conditional distribution *p(**y**|**x**)*, in which the variable ***y***$=\langle y_1,\ldots, y_t\rangle$ represents the labels to be predicted, and the variable ***x***$=\langle x_1,\ldots, x_t\rangle$ represents the observed knowledge. In our case ***y*** are the tags to be assigned to the tokens and separators in the text, and ***x*** is the information about these tokens and separators that we will input to the system.

Conditional random fields are often used in classification tasks in which the entities to be classified have highly dependent features (sequence labeling, IE, etc.). Conditional random fields differ from other graphical models, such as Hidden Markov Models}, that use a joint probability distribution *p(**y**,**x**)* and therefore require to know the prior probability distribution *p(**x**)*. In conditional random fields the input variables ***x*** do not need to be represented, thus avoiding the non trivial modeling of the prior probability distribution *p(**x**)*, and allowing the use of rich and dependent features of the input.

CRF++ is the implementation of *linear-chain conditional random fields*, that define the conditional probability of ***y*** given ***x*** as:

$$P\langle y|x:\theta\rangle=\frac{1}{Z(x)}\exp\left(\sum_{t=1}^{T}\sum_{k=1}^{K}\theta_k f_k\left(y_{t-1},y_t;x_t\right)\right)$$

where *Z(x)* is a normalization factor, θ_k is one of the *K* model parameter weights corresponding to a feature function $f_k(y_{t-1},y_t;\mathbf{x}_t)$.

Each feature function f_k describes the sequence ***x*** at position *t* with label y_t observed with a transition from label y_{t-1} to y_t.

CRF++ allows to define feature functions f_k by using information about the token to be labeled, and about the tokens around the token to be labeled; this is possible by defining the size of the window of tokens to be considered around the one to be labeled. The window can be composed by information belonging to tokens that precede the token to be labeled or belonging to tokens that folllow the token to be labeled. Having a wide window is important in tasks that require to identify long annotated sequence of tokens. For more details about conditional random fields see [Sutton and McCallum, 2007].

A conditional random field learner needs each t-unit either in a training document or in a test document to be represented in vectorial form. In this work we have used a set of features consisting of the original token as it appears in the text, its part of speech, and the relative lemma, plus information about capitalization, prefixes, suffixes and stemming. To give the learner more robustness over typographical and orthographical errors, we use as features the token lemma, the token prefixes (the first character of the token, the first two, the first three, the first four) and suffixes (the last character of the token, the last two, the last three, the last four), the token stem, and token capitalization information. With token capitalization we identify 4 types of capitalization: “all capital”, indicating that all the letters in the word are uppercased, “first letter capital”, indicating that just the first letter of the word is uppercased and the rest of the letters are all lowercased, “all lower”, indicating that none of the letters in the word are uppercased, and “mixed case”, indicating that there are some uppercased letters and some lowercased letters. We also include as a feature the part of speech of the token.

As the evaluation measure we use the recently proposed *token & separator F_1 model* [Esuli and Sebastiani, 2010]. According to this model, a tagger is evaluated according to the well-known F_1 measure on an event space consisting of all t-units in the text. In other words, each t-unit u_k (rather than each segment, as in the traditional “segmentation F-score” model) counts as a true positive, true negative, false positive, or false negative for a given tag c_i, depending on whether u_k belongs to c_i or not in the predicted annotation and in the true annotation. As argued in [8], this model has the advantage that it credits a system for partial success, and that it penalizes both overtagging and undertagging.

As is well-known, F_1 combines the contributions of precision (π) and recall (ρ), and is defined as $F_1 = \frac{2\pi\rho}{\pi+\rho} = \frac{2\mathrm{TP}}{2\mathrm{TP}+FP+FN}$, where *TP*, *FP*, and *FN* stand for the numbers of true positives, false positives, and false negatives, respectively. Note that F_1 is undefined when TP=FP=FN =0; in this case we take F_1 to equal 1, since the tagger has correctly tagged all t-units as negative.

We compute F_1 across the entire test set, i.e., we generate a single contingency table by putting together all t-units in the test set, irrespective of the text they belong to. We then compute both microaveraged F_1 (denoted by $F_1\mu$) and macroaveraged F_1 (F_1M). $F_1\mu$ is obtained by (i) computing the tag-specific values TP_i, FP_i and FN_i, (ii) obtaining TP as the sum of the TP_i’s (same for *FP* and *FN*), and then (iii) applying the $F_1 = \frac{2\mathrm{TP}}{2\mathrm{TP}+FP+FN}$ formula. F_1M is obtained by first computing the tag-specific F_1 values and then averaging them across the c_j ’s.

An advantage of using F_1 as the evaluation measure is that it is symmetric, i.e., its values do not change if one switches the roles of the human annotator and the automatic tagger. This means that F_1 can also be used as a measure of agreement between any two annotators/taggers, regardless of whether they are human or machine, since it does not require one to specify who among the two is the “gold standard” against which the other needs to be checked. For this reason, in the following section we will use F_1 both (a) to measure the agreement between our system and the human annotators, and (b) to measure the agreement between the two human annotators. This will allow us to judge in a direct way how far our system is from human performance.

References

Ben-Dov, M., Feldman, R.: Text Mining and Information Extraction. In Oded Maimon, Lior Rokach (Eds.): Data Mining and Knowledge Discovery Handbook, 2nd ed. Springer, 2010, pp. 809-835

Esuli, A., Marcheggiani, D., Sebastiani, F.,: Information Extraction from Radiology Reports. Presented at the 7th Italian Conference on Digital Libraries, Pisa, Italy, 2011

Esuli, A., Sebastiani, F.: Evaluating information extraction. In: Proceedings of the Conference on Multilingual and Multimodal Information Access Evaluation (CLEF’10), Padova, IT (2010) 100–111

Lafferty, J., McCallum, A., Pereira, F.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proceedings of the 18th International Conference on Machine Learning (ICML’01), Williamstown, US (2001) 282–289

Li, L., Zhou, R., Huang, D.: Two-phase biomedical named entity recognition using CRFs. Computational Biology and Chemistry 33(4):334-338 (2009)

McCallum, A.: Information extraction: Distilling structured data from unstructured text. Queue 3(9) (2005) 48–57

Nadeau, D., Sekine, S.: A survey of named entity recognition and classification. Linguisticae Investigationes 30(1) (2007) 3—26

Sarawagi, S.: Information extraction. Foundations and Trends in Databases 1(3) (2008) 261--377

Sutton, C., McCallum, A.: An introduction to conditional random fields for relational learning. In Getoor, L., Taskar, B., eds.: Introduction to Statistical Relational Learning. The MIT Press, Cambridge, US (2007) 93–127

Szarvas, G., Farkas, R., and Busa-Fekete, R.: State-of-the-art anonymisation of medical data with an iterative machine learning model/framework. Journal of the American Medical Informatics Association, 14(5):574–580, 2007.

Trogkanis, N., Elkan, C.: Conditional Random Fields for Word Hyphenation. Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden, 2010, pp. 366-374.

Zeng, G., Zhang, C., Xiao, Bo., Lin, Z.: CRFs-Based Chinese Named Entity Recognition with Improved Tag Set. Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering, 2009, Los Angeles, US, 2009:519-522

2.3 Automatic Classification of Metadata Records

As part of their routine information management protocols, many organizations and content providers classify their content (or the metadata that describe this content) according to a set of categories (or “classification scheme”) that effectively describe the domain this content is about. It is often the case that, unless the domain is trivial in nature, this classification scheme has a hierarchical structure, since a non-hierarchical, flat structure would be too clumsy to accommodate the many categories that describe this domain. We will indeed assume that content providers do structure their content according to a hierarchically shaped classification scheme. This assumption is non-restrictive, since a flat classification scheme may also be seen as a hierarchical classification scheme consisting of two levels only, the root (level 0) and all the categories (level 1) appended to the root as children.

The field of supervised learning that tackles the classification of textual items (as metadata records are) under hierarchically structured classification schemes is called *hierarchical text categorization* (HTC). Notwithstanding the fact that most large-sized classification schemes for text (e.g. the ACM Classification Scheme, the MESH thesaurus, the NASA thesaurus) indeed have a hierarchical structure, the attention of text classification (TC) researchers has mostly focused on algorithms for “flat” classification. These algorithms, once applied to a hierarchical classification problem, are not capable of taking advantage of the information inherent in the class hierarchy, and may thus be suboptimal, in terms of efficiency and/or effectiveness. On the contrary, many researchers have argued that by leveraging on the hierarchical structure of the classification scheme, heuristics of various kinds can be brought to bear that make the classifier more efficient and/or more effective. This is the reason why, for the purposes of T2.1.3, we will focus our attention on algorithms explicitly devised for HTC.

An important intuition that underlies HTC algorithms is that, by viewing classification as the identification of the paths that, starting from the root, funnel the document down to the subtrees where it belongs (in “Pachinko machine” style), entire other subtrees can be pruned from consideration. That is, when the classifier corresponding to an internal node outputs a negative response, the classifiers corresponding to its descendant nodes need not be invoked any more, thus reducing the computational cost of classifier invocation exponentially [Chakrabarti et al. 1998; Koller and Sahami 1997]. A second important intuition is that, by training a binary classifier for an internal node category on a well-selected subset of training examples of local interest only, the resulting classifier may be made more attuned to recognizing the subtle distinctions between documents belonging to that node and those belonging to its sibling nodes. While this technique promises to bring about more effective classifiers, it is also going to improve efficiency, since a smaller set of examples is used in training, thereby making classifier learning speedier. Many of these intuitions have been used in close association with several learning algorithms; the most popular choices in this respect have been naïve Bayesian methods, neural networks, support vector machines, and example-based classifiers.

In T2.1.3 we will use an HTC algorithm based on *boosting* technology, called TreeBoost.MH [Esuli et al, 2008]; the reasons for this choice include the fact that TreeBoost.MH has proved to be highly efficient and, above all, highly accurate in a number of applications we have previously applied it to, including the classification of newswire reports [Esuli et al, 2008], of medical discharge reports [Esuli et al, 2008], and of radiology reports [Baccianella et al, 2011]. TreeBoost.MH is a multi-label (ML) HTC algorithm that consists of a hierarchical variant of AdaBoost.MH [Schapire and Singer, 2000], the most important member of the family of boosting algorithms; here, *multi-label* (ML) means that a document can belong to zero, one, or several categories at the same time. TreeBoost.MH embodies several intuitions that had arisen before within HTC, e.g., the intuitions that both feature selection and the selection of negative training examples should be performed "locally", i.e. by paying attention to the topology of the classification scheme. TreeBoost.MH also incorporates the intuition that the weight distribution that boosting algorithms update at every boosting round should likewise be updated "locally". All these intuitions are embodied within TreeBoost.MH in an elegant and simple way, i.e. by defining TreeBoost.MH as a recursive algorithm that uses AdaBoost.MH as its base step, and that recurs over the tree structure.

In the next two sections we give a concise description of TreeBoost.MH.

2.3.1 TreeBoost.MH: A hierarchical version of AdaBoost.MH for multi-label TC

When discussing an HTC application it is always important to specify what the *semantics of the hierarchy* is, i.e., to specify the semantic constraints that a supposedly perfect classifier would enforce; which constraints are in place has important consequences on which algorithms we might want to apply to this task, and, more importantly, on how we should evaluate these algorithms. For instance, one should specify whether a document can in principle belong to zero, one, or several categories (which is indeed our assumption within T2.1.3), or whether it always belongs to one and only one category. No less importantly, one should specify whether it is the case that

1. a document *d* that is a positive example of a category is also a positive example of all its ancestor categories. We assume this to be the case.
2. a document *d* can in principle be a positive example of an internal node category and at the same time *not* be a positive example of any of its descendant categories. We assume this to be the case.

Assumption 2 is indeed useful for tackling datasets in which documents with these characteristics do occur, while at the same time not preventing us to deal with datasets with the opposite characteristics. A consequence of these two assumptions is that the set of the positive training examples of a nonleaf category is a (possibly proper) superset of the union of the sets of positive training examples of all its descendant categories.

TreeBoost.MH embodies several intuitions that had arisen before within HTC.

The first, fairly obvious intuition (which lies at the basis of practically all HTC algorithms proposed in the literature) is that, in a hierarchical context, the classification of a document is to be seen as a descent through the hierarchy, from the root to the (internal or leaf) categories where the document is deemed to belong. In ML classification this means that each non-root category has an associated binary classifier which acts as a “filter” that prevents unsuitable documents to percolate to the descendants of the category. All test documents that a classifier deems to belong to a category are passed as input to all the binary classifiers corresponding to its children categories, while the documents that the classifier deems not to belong to the category are “blocked” and analysed no further. Note that it may well be the case that a document is deemed to belong to a category by its corresponding classifier and is then rejected by all the binary classifiers corresponding to its children categories; this is indeed consistent with assumption (2) above. In the end, each document may thus reach zero, one, or several (leaf or internal node) categories, and is thus classified as belonging to them.

The second intuition is that the training of a classifier should be performed “locally”, i.e. by paying attention to the topology of the classification scheme. To see this, note that, during classification, if the classifier for a category has performed reasonably well, the classifier for the children categories will only (or mostly) be presented with documents that belong to the subtree rooted in that category. As a result, the training of a classifier for a given category should be performed by using, as negative training examples, the positive training examples of its sibling categories, with the obvious exception of the documents that are also positive training examples of the category itself. In particular, training documents that only belong to categories other than those mentioned above need not be used. The rationale of this choice is that the negative training examples thus selected are “quasi-positive” examples of the category [Fagni and Sebastiani, 2010], i.e. are the negative examples that are closest to the boundary between the positive and the negative region of the category (a notion akin to that of “support vectors” in SVMs), and are thus the most informative negative examples that can be used in training. This is beneficial also from the standpoint of (both training and classification time) efficiency, since fewer training examples and fewer features are involved.

The third intuition is similar, i.e. that feature selection should also be performed “locally”, by paying attention to the topology of the classification scheme. As above, if the classifier for the category has performed reasonably well, the classifiers for its children categories will only (or mostly) be presented with documents that belong to the subtree rooted in the category itself. As a consequence, for the classifiers corresponding to the children categories, it is cost-effective to employ features that are useful in discriminating (only) among themselves; features that discriminate among categories lying outside the subtree rooted in the category are too general, and features that discriminate among the subcategories of the children categories are too specific. This intuition, albeit in the slightly different context of single-label classification, was first presented in [Koller and Sahami, 1997].

TreeBoost.MH also embodies the novel intuition that the weight distribution that boosting algorithms update at every boosting round should likewise be updated “locally”. In fact, the two previously discussed intuitions indicate that hierarchical ML classification is best understood as consisting of several independent (flat) ML classification problems, one for each internal node of the hierarchy. In a boosting context, this means that several independent distributions, each one “local” to an internal node, should be generated and updated by the process. In this way, the “difficulty” of a category will only matter *relative* to the difficulty of its sibling categories. This intuition is of key importance in allowing TreeBoost.MH to obtain exponential savings in the cost of training over AdaBoost.MH.

TreeBoost.MH incorporates these four intuitions by factoring the hierarchical ML classification problem into several “flat” ML classification problems, one for every internal node in the tree. TreeBoost.MH learns in a recursive fashion, generating a binary classifier for each non-root category, by means of which hierarchical classification can be performed in “Pachinko machine” style.

Learning in TreeBoost.MH proceeds by first identifying whether a leaf category has been reached, in which case nothing is done, since the classifiers are generated only at internal nodes. If an internal node has been reached, a ML feature selection process may (optionally) be run to generate a reduced feature set on which the ML classifier for the node will operate. This may be dubbed a “glocal” feature selection policy, since it takes an intermediate stand between the well-known “global” policy (in which the same set of features is selected for all the categories) and the “local” policy (in which a different set of features is chosen for each different category). The glocal policy selects a different set of features for each (maximal) set of sibling categories. We use information gain as the feature selection function, and Forman’s [2004] round robin as a feature score globalization method. After the reduced feature set has been identified, TreeBoost.MH calls upon AdaBoost.MH to solve a ML (flat) classification problem for the set of sibling categories; again, in order to implement the “quasi-positive” policy discussed above, the negative training examples of a category are taken to be the set of the positive training examples of its sibling categories minus the positive training examples of the category itself. Note that this implements the view, discussed above, of several independent, “local” distributions being generated and updated during the boosting process.

Finally, after the ML classifier for a maximal set of sibling categories has been generated, for each such category a recursive call to TreeBoost.MH is issued that processes the subtree rooted in the category in the same way. The final result is a hierarchical ML classifier in the form of a tree of binary classifiers, one for each non-root node, each consisting of a committee of decision stumps.

2.3.2 Related work

HTC was first tackled in Wiener et al. [1995], in the context of a TC system based on neural networks and latent semantic indexing. The intuition that it could be useful to perform feature selection locally by exploiting the topology of the tree is originally due to Koller and Sahami [1997]. However, this work dealt with single-label text categorization, which means that feature selection was performed ‘‘collectively’’, i.e., relative to the set of children of each internal node; given that in T2.3.1 we are in an ML classification context, we instead do it ‘‘individually’’, i.e., relative to each child of any internal node. The intuition that the negative training examples for training the classifier for a given category could be limited to the positive training examples of categories topologically close to it is due to Ng et al. [1997] and Wiener et al. [1995]. The notion that, in an ML classification context, classifiers at internal nodes act as ‘‘routers’’ informs much of the HTC literature, and is explicitly discussed in Ruiz and Srinivasan [2002], which proposes a HTC system based on neural networks.

Other works in hierarchical text categorization have focused on other specific aspects of the learning task. For instance, the ‘‘shrinkage’’ method presented in McCallum et al. [1998] is aimed at improving parameter estimation for data-sparse leaf categories in a single-label HTC system based on a naive Bayesian method; the underlying intuitions are specific to naive Bayesian methods, and do not easily carry over to other contexts. Incidentally, the naive Bayesian approach seems to have been the most popular among HTC researchers, since several other HTC models are hierarchical variations of naive Bayesian learning algorithms [Chakrabarti et al. 1998; Gaussier et al. 2002; Toutanova et al. 2001; Vinokourov and Girolami 2002]; SVMs have also recently gained popularity in this respect [Cai and Hofmann 2004; Dumais and Chen 2000; Liu et al. 2005; Yang et al. 2003].

References

Baccianella, S., Esuli, A., & Sebastiani, F. (2011). Single-Label Classification of Radiology Reports under the ACR Classification Scheme. Presented at the 7th Italian Research Conference on Digital Libraries, Pisa.

Cai, L., & Hofmann, T. (2004). Hierarchical document categorization with support vector machines. In Proceedings of the 13th ACM International Conference on Information and Knowledge Management (CIKM'04), pp. 78–87.

Chakrabarti, S., Dom, B. E., Agrawal, R., & Raghavan, P. (1998). Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies. Journal of Very Large Data Bases, 7(3), 163–178.

Dumais, S. T., & Chen, H. (2000). Hierarchical classification of web content. In Proceedings of the 23rd ACM International Conference on Research and Development in Information Retrieval (SIGIR'00) (pp. 256–263). Athens, GR.

Esuli, A., Fagni, T., & Sebastiani, F. (2008). Boosting Multi-label Hierarchical Text Categorization. Information Retrieval, 11(4):287-313.

Fagni, T. & Sebastiani, F. (2010). Selecting Negative Examples for Hierarchical Text Classification: An Experimental Comparison. Journal of the American Society for Information Science and Technologies, 61(11):2256-2265.

Forman, G. (2004). A pitfall and solution in multi-class feature selection for text classification. In Proceedings of the 21st International Conference on Machine Learning (ICML'04). Banff, CA.

Gaussier, E., Goutte, C., Popat, K., & Chen, F. (2002). A hierarchical model for clustering and categorising documents. In Proceedings of the 24th European Colloquium on Information Retrieval Research (ECIR'02) (pp. 229–247). Glasgow, UK.

Koller, D., & Sahami, M. (1997). Hierarchically classifying documents using very few words. In Proceedings of the 14th International Conference on Machine Learning (ICML'97) (pp. 170–178). Nashville, US.

Liu, T. Y., Yang, Y., Wan, H., Zeng, H. J., Chen, Z., & Ma, W. Y. (2005). Support vector machines classification with a very large-scale taxonomy. SIGKDD Explorations, 7(1), 36–43.

McCallum, A. K., Rosenfeld, R., Mitchell, T. M., Ng, A. Y. (1998). Improving text classification by shrinkage in a hierarchy of classes. In Proceedings of the 15th International Conference on Machine Learning (ICML'98) (pp. 359–367). Madison, US.

Ng, H. T., Goh, W. B., Low, K. L. (1997). Feature selection, perceptron learning, and a usability case study for text categorization. In Proceedings of the 20th ACM International Conference on Research and Development in Information Retrieval (SIGIR'97) (pp. 67–73). Philadelphia, US.

Ruiz, M., & Srinivasan, P. (2002). Hierarchical text classification using neural networks. Information Retrieval, 5(1), 87–118.

Schapire, R. E., & Singer, Y. (2000). BOOSTEXTER: A boosting-based system for text categorization. Machine Learning, 39(2/3), 135–168.

Toutanova, K., Chen, F., Popat, K., & Hofmann, T. (2001). Text classification in a hierarchical mixture model for small training sets. In Proceedings of the 10th ACM International Conference on Information and Knowledge Management (CIKM'01) (pp. 105–113). Atlanta, US.

Vinokourov, A., & Girolami, M. (2002). A probabilistic framework for the hierarchic organisation and classification of document collections. Journal of Intelligent Information Systems, 18(2/3), 153–172.

Wiener, E. D., Pedersen, J. O., & Weigend, A. S. (1995). A neural network approach to topic spotting. In Proceedings of the 4th Annual Symposium on Document Analysis and Information Retrieval (SDAIR'95) (pp. 317–332). Las Vegas, US.

Yang, Y., Zhang, J., & Kisiel, B. (2003). A scalability analysis of classifiers in text categorization. In Proceedings of the 26th ACM International Conference on Research and Development in Information Retrieval (SIGIR'03) (pp. 96–103). Toronto, CA.

2. Ingestion-related ASSETS Needs and Constraints

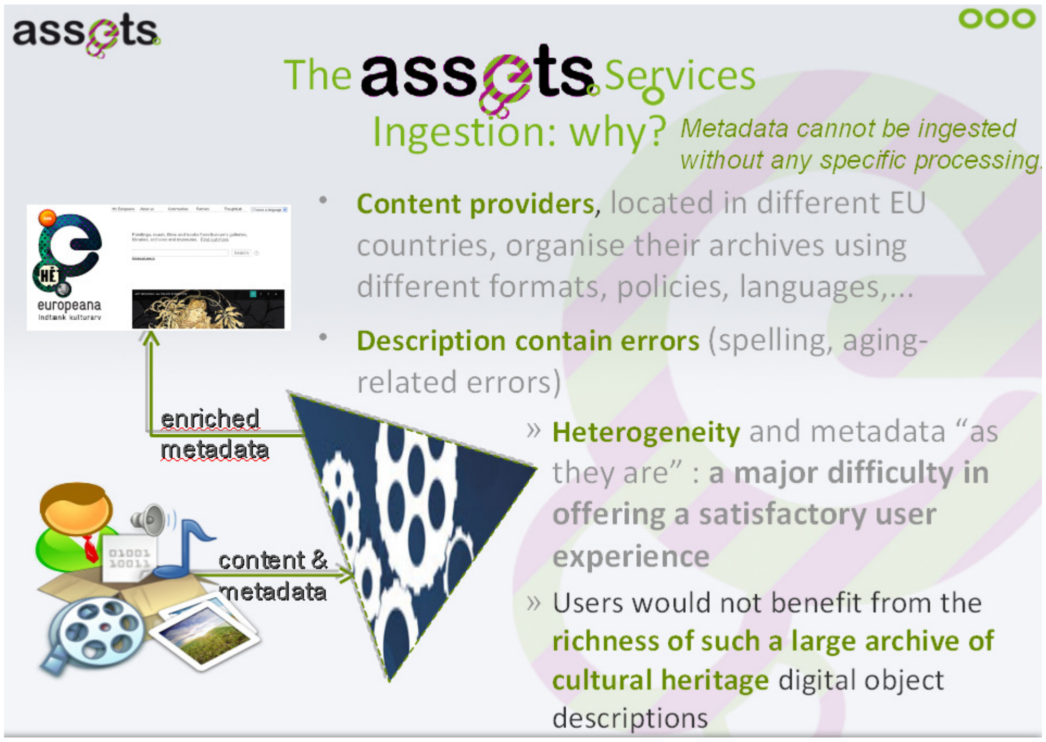
This section identifies the set of ingestion-related needs and constraints, by describing the issues of Metadata Enrichment, Heterogeneity Reduction, Information Extraction and Classification. This allows describing the rationale behind the decisions for the system and its services, the models, the constraints and the features.

2.1 Metadata Enrichment, Heterogeneity Reduction, Information Extraction and Classification

The **ASSETS portal** will act as an integration system acquiring information, i.e., metadata content, from different entities (e.g. museums, libraries, archives, etc.) which are called with the general term of Content Providers (CPs) located in different European countries. Since different CPs organize their digital archives using different formats, the metadata gathered within the ASSETS project will suffer from a great heterogeneity. In fact, the metadata submitted by different CPs are expected to have the following characteristics:

- They will be expressed in different languages. Basically, ASSETS will process metadata containing words and texts written in the most of the European languages.
- They will use different formats for the textual representation of many values, such as, for example, the dates (e.g. dd-mm-yyyy or mm-dd-yyyy) or the dimensions of a physical object (e.g. different units measurement) .
- They will be formatted according to different XML schemas (e.g., Dublin core-based or entirely proprietary formats).
- They are likely to contain errors in the textual representations and descriptions. There are several types of errors the metadata can suffer from. We report two example cases that we plan to study. :
    - **Spelling errors.** Authors can be easily misnamed if they do no have a well known name in the annotator's native language. There might be spelling errors even if the author's name (or other texts) does not change among languages. In one of the first examples received (by DW), Mozart has been written as *Mozzart*.
    - **Aging-related errors and problems**. The metadata datasets of cultural heritage institutions have been produced over a long period of time by different archivists. Without digital preservation actions, it is likely that different archivists used (over a long period of time) different terms and ontologies for annotating similar or related metadata records. Furthermore, metadata annotated many years ago might refer to ontologies that are no longer used. This last type of errors will be managed according to the **guidelines coming from *WP2.3 Digital Preservation***.

Such heterogeneity represents a major difficulty in offering a satisfactory user experience on the ASSETS web site. In fact, if the metadata are ingested 'as they are', without any specific processing, the users would not benefit from the richness of such a large archive of cultural heritage digital object descriptions. The results returned for a specific query might not include several interesting results (for example, an Italian-speaking user might not receive the link to an Italian book whose metadata record is not expressed in Italian). The results might as well contain irrelevant result due to false matches among different languages or formats.


The assets Services

Ingestion: why? *Metadata cannot be ingested without any specific processing.*

- **Content providers**, located in different EU countries, organise their archives using different formats, policies, languages,...
- **Description contain errors** (spelling, aging-related errors)
  - » **Heterogeneity** and metadata "as they are" : **a major difficulty in offering a satisfactory user experience**
  - » Users would not benefit from the **richness of such a large archive of cultural heritage** digital object descriptions

Figure 1 - The rationale of Ingestion Service

Europeana Foundation realised the need of integrating the tools that collectively cover the ingestion functionality into a unique framework. In particular, the need for having a unified ingestion workflow with a single access point was identified as a requirement for the Assets Project. Since this is a joint collaboration, the extended description for the design of this service will be provided within the Europeanalabs documentation.

Europeana metadata contains both structured and unstructured information. Structured information is provided by those metadata fields that identify well-specified type of information, e.g., "date", "creator", "language". Unstructured information is provided by those metadata fields that act as containers of generic information, e.g., "description".

The aim of the knowledge extraction service is provide Europeana ingestion and enrichment process with automatic information extraction functionalities that enable to extract relevant structured information from unstructured metadata fields contained in Europeana records.

Finally, the aim of "Metadata classification" is to develop a service for the automated classification of metadata records under a taxonomy of semantic categories.

The classification process consists of linking a record to zero, one, or several from a (taxonomically organized) set of predefined categories (aka "classes", or "concepts", or "codes"). The set of predefined categories is called the classification scheme. Classification is thus akin to "populating" a taxonomy with instances of the concepts in the taxonomy.

Europeana records are provided by many different content provider, which may (i) not use any classification schema for their data, (ii) use a very specific classification scheme custom tailored for specific local purposes of the content provider, (iii) use a standard well-know classification schema for their data, either general-purpose (e.g., Library of the Congress

Subject Headings, LCSH) or discipline-specific (e.g., Medical Subject Headings). Among these three cases the last one is certainly the preferred one for Europeana.

The metadata classification service will enable Europeana to automatically classify the new records provided by content providers, and those already acquired by Europeana, following a set of general-purpose and/or discipline-specific classification schema.

The ultimate goal of the task is making the searching and browsing experience on the part of the user more satisfactory; e.g.:

- user can navigate from record to concept and to other records belonging to same concept or sibling concepts;
- user can restrict search to records belonging to a specific concept;
- user can ask to group the search results according to the concepts they belong to.

3. Ingestion-related ASSETS Services

This section allows the reader to understand the approach adopted by each ASSETS research activity (often according to the state of the art and standards) for identifying the proposed solution to the issues described in the previous section.

3.1 The Ingestion Services

Most of algorithms that will be experimented and implemented will be based on a Machine Learning (ML) approach and, more in details, on supervised learning methods. In that approach a learning machine induces a classifier by observing a set of metadata records that have already been associated with one or more categories (such set is called training set).

In particular:

- for the Metadata Cleaning service, the training set should contain examples of typical error with the correspondent corrections;
- for the Knowledge Extraction service, the training set should contain annotated descriptions. Basically, the relevant entities (e.g. names of persons or places) should be surrounded with specific tags;
- for the Metadata Classification service, the training set should contain metadata records associated to the correct categories;
- the Ingestion team has already prepared XML schemas the CPs need to use in order to provide training sets before the Europeana Data Model (EDM) is finalized. Once the EDM has been finalized, the Ingestion team will modify the current XML schemas to take into account the fresh common data format.

ML methods have already been proven to be very effective in knowledge extraction and classification tasks. Even if some established algorithms will have to be tailored for the specific needs of the ASSETS project, it is reasonable to expect very good results if enough annotated metadata are provided by the CPs.

However, supervised ML algorithms are not the only methods that will be experimented in the Ingestion module. Possible alternatives include unsupervised learning methods (that do not require a training set) and non-adaptive methods for the simpler cases.

In general, we will limit the number of cases where a non-adaptive approach will be used. In fact, even if in some cases a rule-based system would be quite effective and much simpler to implement, it would not adapt to new datasets provided by CPs that decide to join ASSETS (or Europeana).

The ASSETS proposal for the ingestion issues is to provide implementations of advanced services with functionalities able to clean, enrich, extract knowledge, and classify the metadata records coming from the CPs involved in the ASSETS project and the metadata records currently indexed by Europeana.

The ASSETS ingestion services will:

- clean and perform a basic enrichment of the metadata (using URIs pointing to controlled vocabularies and authority files) through the **Metadata Cleaning** service;
- extract knowledge and enrich the original metadata with the new information through

the **Knowledge Extraction** service;

- classify the metadata under a well-defined classification taxonomy through the **Metadata Classification** service;

3.1.1 Metadata Cleaning

The service has the following set of goals:

1. correct part of the errors in the metadata;
2. normalize the values of specific fields by using the same textual representation in all of the datasets received by the ASSETS project;
3. perform a basic enrichment of specific elements of the metadata records.

3.1.2 Knowledge Extraction

Usually the metadata records have one or more descriptive fields containing free-form, unstructured textual description of a physical object. The second step ***Knowledge Extraction*** will extract information by such long textual description. The type of extracted information will depend on the metadata domain and cannot be specified at this moment.

3.1.3 Metadata Classification

Even after the cleaning and the extraction of knowledge have been accomplished, the metadata remain a largely disorganized set. The ASSETS users might benefit from an organization of the metadata under a well defined semantic taxonomy, like, for example, the Library of Congress Classification scheme. The third step ***Metadata Classification*** will associate each metadata record to one or more categories in classification schemes that are yet to be chosen.

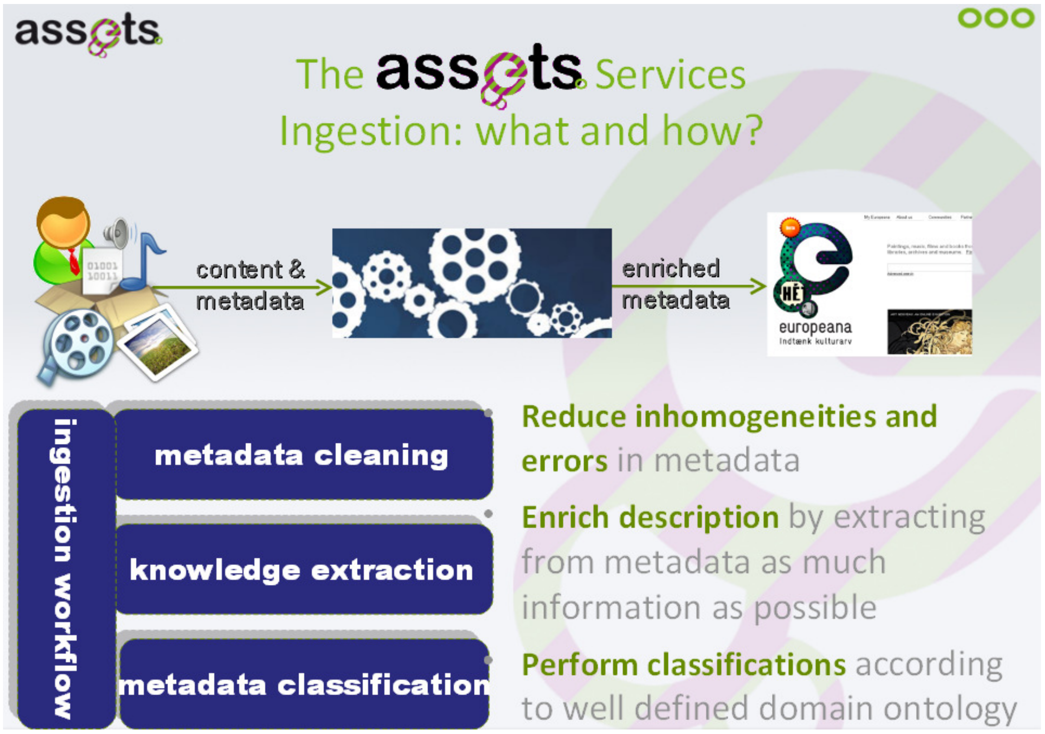

assets
The assets Services
Ingestion: what and how?
content & metadata
enriched metadata
europeana
ingestion workflow
metadata cleaning
knowledge extraction
metadata classification
Reduce inhomogeneities and errors in metadata
Enrich description by extracting from metadata as much information as possible
Perform classifications according to well defined domain ontology

Figure 2 – The ASSETS Ingestion Services

For further details on how CPs have to format their data in order to submit training data to the enrichment services of WP2.1 see Appendix 1 - Enrichment Services Training Data Format.

The ASSETS proposal for the ingestion issues is to provide implementations of advanced services with functionalities able to clean, enrich, extract knowledge, and classify the metadata records coming from the CPs involved in the ASSETS project and the metadata records currently indexed by Europeana.

The ASSETS ingestion services will:

- clean and perform a basic enrichment of the metadata (using URIs pointing to controlled vocabularies and authority files) through the Metadata Cleaning service.
- extract knowledge and enrich the original metadata with the new information through the Knowledge Extraction service.
- classify the metadata under a well-defined classification taxonomy through the Metadata Classification service

3.1.4 Ingestion Workflow

The Europeana web portal implements a search engine over the European cultural heritage. In order to provide this functionality, an index with the description of the masterpieces was created. This information is retrieved from the Content Providers (CPs). The model used for metadata aggregation is sketched in the following figure.

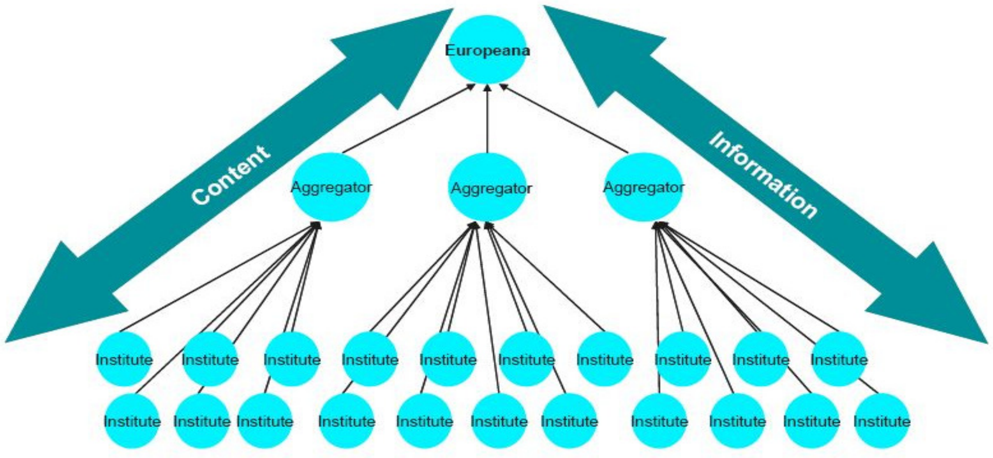


Figure 3 – Aggregators in the Europeana organisation model

Where the aggregators are organizations which integrate the data retrieved from content providers, and transform it into a representation compatible with the Europeana search index. See also Europeana Aggregator's Handbook [6].

The harvesting of the metadata is based on the Open Archives Initiative Protocol for Metadata Harvesting (for further details see http://www.openarchives.org/pmh). This protocol standardises the harvesting of metadata by defining a web service interface which provides descriptions of the collection objects in XML format.

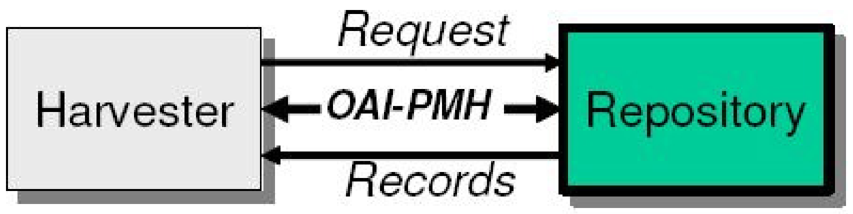


Figure 4 - OAI-PMH Harvesting

These XML files, retrieved through the OAI-PMH interface, are used as input for the ingestion workflow.

The ASSETS proposal for the ingestion workflow management will:

- Integrate execution of the metadata enrichment services in a standardized workflow - Ingestion Workflow Management;
- Build the multimedia index used for content based search functionality - Post-Ingestion Processing;

The ingestion framework will perform the necessary processing steps as much as possible in an autonomous way. The ingestion team is involved in the scheduling process, where the team needs/can prioritise certain collections or inform the system about a new collection which needs to be processed. After processing, the records remain in the acceptance point until a member of the ingestion team confirms them.

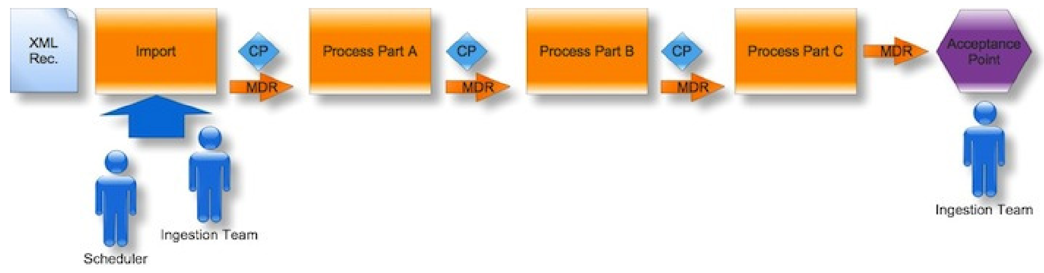


Figure 5 - The Ingestion Process flow

4. The ASSETS Data Models and Interfaces for Ingestion Services

The information available from the above sections allows us to depict the context and the application scenarios for the ASSETS services, and consequently, this allows us to identify concepts which are presented below. These concepts are modelled as UML diagrams which emphasize the relationships between and the properties of objects, which represent the ASSETS Data Models.

The first part of this section introduces the Common Data Model, and afterwards it and describes the specific Data Models for Ingestion Services.

We have adopted the following table templates to identify important aspects of each service such as: the interfaces, the dependencies, the responsibilities, the key concepts managed and the operations supported.

Service Name	*The name of the service*
Responsibility	*List of items for the responsibility of the service*
Provided Interfaces	*List of the interfaces through whom the service provides its features and manages key concepts*
Dependencies	*List of dependencies with other ASSETS services, if any. If this information is not available, provides the expected key concepts which represent inputs for the service from other ASSETS services*

Interface Name	*The name of the service interface*
Key Concepts	*Identification of the key concepts (data model) managed by the interface*
Operations	*List of Item for the operations of the interface*

It is important to remark that ASSETS project is adopting an iterative and incremental development process. For that reason, the interfaces and models presented here are a picture of the current development phase, that will evolve as the development of services proceeds toward the release of the prototypes, which are expected by the month 16 of the project.

4.1 System Architecture Overview

Differently from Europeana project, which stores exclusively the items' metadata within its database, the ASSETS services will need to index and store multimedia content, too. Moreover, the "Video summarisation, adaptation, indexing and retrieval" service will generate video summaries which need to be made available to the end user. These requirements enforce the enhancement of ASSETS architecture with the usage of heavyweight-technologies, in comparison to Europeana architecture.

Anyway, one of our project goals is to implement high quality services and to integrate as many as possible into the Europeana portal. Therefore, the ASSETS architecture needs to follow as long as possible the Europeana architecture, technologies and implementation guidelines.

The proposed system architecture is sketched in the following figure.

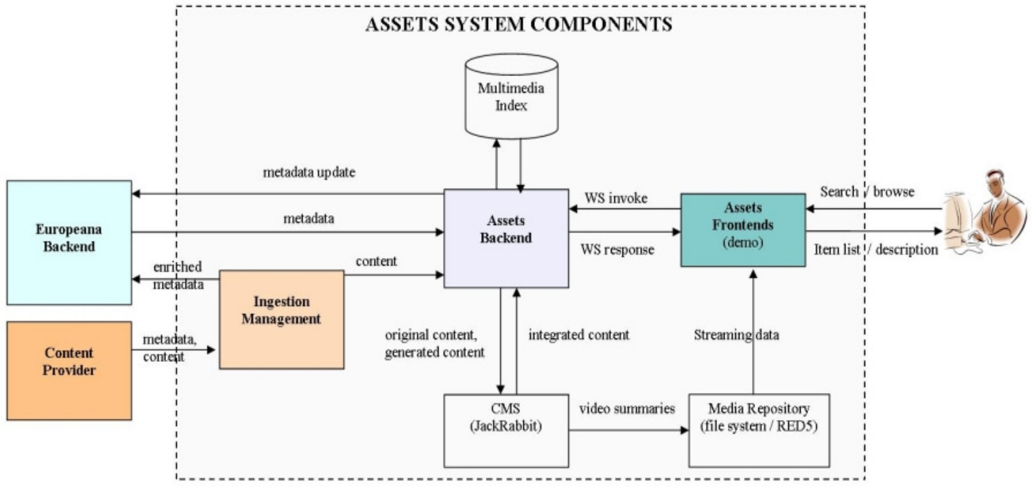


Figure 6 – Overview of the ASSETS System Architecture

The dashed line marks the border of the ASSETS system and its external interfaces. The named arrows represent the dataflows exchanged between the ASSETS system components (internally or with the outer world).

The assets services needs to communicate with:

- Content Providers Portal: which need to provide on OAI-PMH interface for metadata harvesting and an URL for content harvesting;
- Europeana Backend: which will be accessed through its Web API for updating the Europeana metamodel and metadata with the one created by ASSETS services;
- End Users: access the ASSETS search and browsing services from their browser.

4.1.1 System Components

The Assets internal architecture is composed from 3 main components.

First of them has the role of collecting the metadata information and the content from the content providers and submitting it for storage into the Assets&Europeana databases (Ingestion Management).

The second one implements the business functionality (Assets Backend) and makes it available on Internet through a Web API.

The third component implements the Graphical User Interface (Assets Frontend) which offers a rich set of browsing and searching functionality for end users.

The rest of the section will describe the Ingestion Management components, and the Common components on which they rely.

4.2 Common Models and Interfaces

The common components were designed for implementing the basic functionality that is common for all Assets services. This functionality includes the access to the information stored into the Europeana database and Solr index, the unified concept for application configuration, the common data-model used by Assets components, the ORM framework for data storage based on MongoDb.

The Assets common architecture layer is implemented in 5 components: Assets data-model, Assets common-api, Assets common-server-api, Assets common-server, Assets common-client.

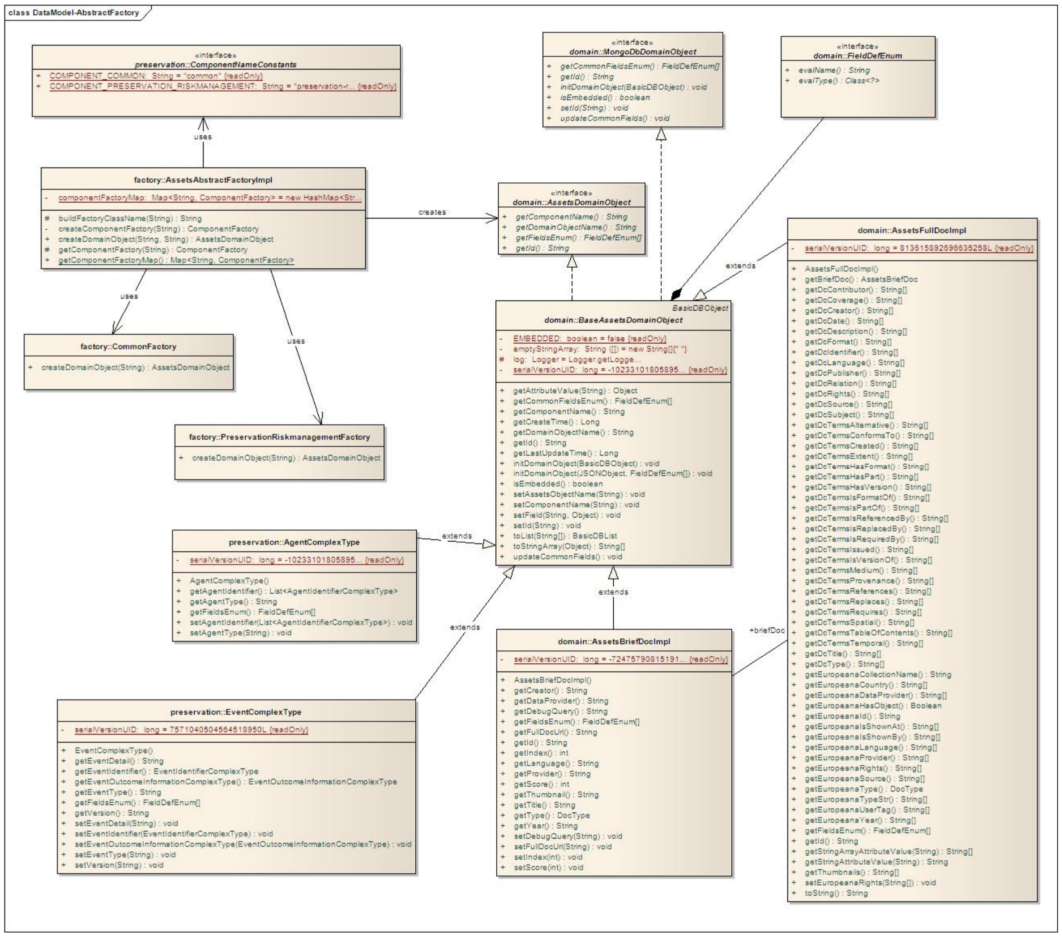

class DataModel-AbstractFactory
«interface»
preservation::ComponentNameConstants
+ COMPONENT_COMMON: String = "common" {readOnly}
+ COMPONENT_PRESERVATION_RISKMANAGEMENT: String = "preservation-r... {readOnly}
uses
factory::AssetsAbstractFactoryImpl
- componentFactoryMap: Map<String, ComponentFactory> = new HashMap<Str...
# buildFactoryClassName(String) : String
- createComponentFactory(String) : ComponentFactory
+ createDomainObject(String, String) : AssetsDomainObject
# getComponentFactory(String) : ComponentFactory
+ getComponentFactoryMap() : Map<String, ComponentFactory>
creates
uses
factory::CommonFactory
+ createDomainObject(String) : AssetsDomainObject
uses
factory::PreservationRiskmanagementFactory
+ createDomainObject(String) : AssetsDomainObject
«interface»
domain::MongoDbDomainObject
+ getCommonFieldsEnum() : FieldDefEnum[]
+ getId() : String
+ initDomainObject(BasicDBObject) : void
+ isEmbedded() : boolean
+ setId(String) : void
+ updateCommonFields() : void
«interface»
domain::FieldDefEnum
+ evalName() : String
+ evalType() : Class<?>
«interface»
domain::AssetsDomainObject
+ getComponentName() : String
+ getDomainObjectName() : String
+ getFieldsEnum() : FieldDefEnum[]
+ getId() : String
extends
BasicDBObject
domain::BaseAssetsDomainObject
- EMBEDDED: boolean = false {readOnly}
- emptyStringArray: String ([]) = new String[]{" "}
# log: Logger = Logger.getLogge...
- serialVersionUID: long = -10233101805895... {readOnly}
+ getAttributeValue(String) : Object
+ getCommonFieldsEnum() : FieldDefEnum[]
+ getComponentName() : String
+ getCreateTime() : Long
+ getDomainObjectName() : String
+ getId() : String
+ getLastUpdateTime() : Long
+ initDomainObject(BasicDBObject) : void
+ initDomainObject(JSONObject, FieldDefEnum[]) : void
+ isEmbedded() : boolean
+ setAssetsObjectName(String) : void
+ setComponentName(String) : void
+ setField(String, Object) : void
+ setId(String) : void
+ toList(String[]) : BasicDBList
+ toStringArray(Object) : String[]
+ updateCommonFields() : void
domain::AssetsFullDocImpl
- serialVersionUID: long = 8136158926966352 58L {readOnly}
+ AssetsFullDocImpl()
+ getBriefDoc() : AssetsBriefDoc
+ getDcContributor() : String[]
+ getDcCoverage() : String[]
+ getDcCreator() : String[]
+ getDcDate() : String[]
+ getDcDescription() : String[]
+ getDcFormat() : String[]
+ getDcIdentifier() : String[]
+ getDcLanguage() : String[]
+ getDcPublisher() : String[]
+ getDcRelation() : String[]
+ getDcRights() : String[]
+ getDcSource() : String[]
+ getDcSubject() : String[]
+ getDcTermsAlternative() : String[]
+ getDcTermsConformsTo() : String[]
+ getDcTermsCreated() : String[]
+ getDcTermsExtent() : String[]
+ getDcTermsHasFormat() : String[]
+ getDcTermsHasPart() : String[]
+ getDcTermsHasVersion() : String[]
+ getDcTermsIsFormatOf() : String[]
+ getDcTermsIsPartOf() : String[]
+ getDcTermsIsReferencedBy() : String[]
+ getDcTermsIsReplacedBy() : String[]
+ getDcTermsIsRequiredBy() : String[]
+ getDcTermsIssued() : String[]
+ getDcTermsIsVersionOf() : String[]
+ getDcTermsMedium() : String[]
+ getDcTermsProvenance() : String[]
+ getDcTermsReferences() : String[]
+ getDcTermsReplaces() : String[]
+ getDcTermsRequires() : String[]
+ getDcTermsSpatial() : String[]
+ getDcTermsTableOfContents() : String[]
+ getDcTermsTemporal() : String[]
+ getDcTitle() : String[]
+ getDcType() : String[]
+ getEuropeanaCollectionName() : String
+ getEuropeanaCountry() : String[]
+ getEuropeanaDataProvider() : String[]
+ getEuropeanaHasObject() : Boolean
+ getEuropeanaId() : String
+ getEuropeanaIsShownAt() : String[]
+ getEuropeanaIsShownBy() : String[]
+ getEuropeanaLanguage() : String[]
+ getEuropeanaProvider() : String[]
+ getEuropeanaRights() : String[]
+ getEuropeanaSource() : String[]
+ getEuropeanaType() : DocType
+ getEuropeanaTypeStr() : String[]
+ getEuropeanaUserTag() : String[]
+ getEuropeanaYear() : String[]
+ getFieldsEnum() : FieldDefEnum[]
+ getId() : String
+ getStringArrayAttributeValue(String) : String[]
+ getStringAttributeValue(String) : String
+ getThumbnails() : String[]
+ setEuropeanaRights(String[]) : void
+ toString() : String
+briefDoc
extends
preservation::AgentComplexType
- serialVersionUID: long = -10233101805895... {readOnly}
+ AgentComplexType()
+ getAgentIdentifier() : List<AgentIdentifierComplexType>
+ getAgentType() : String
+ getFieldsEnum() : FieldDefEnum[]
+ setAgentIdentifier(List<AgentIdentifierComplexType>) : void
+ setAgentType(String) : void
extends
extends
preservation::EventComplexType
- serialVersionUID: long = 7571040504564518950L {readOnly}
+ EventComplexType()
+ getEventDetail() : String
+ getEventIdentifier() : EventIdentifierComplexType
+ getEventOutcomeInformationComplexType() : EventOutcomeInformationComplexType
+ getEventType() : String
+ getFieldsEnum() : FieldDefEnum[]
+ getVersion() : String
+ setEventDetail(String) : void
+ setEventIdentifier(EventIdentifierComplexType) : void
+ setEventOutcomeInformationComplexType(EventOutcomeInformationComplexType) : void
+ setEventType(String) : void
+ setVersion(String) : void
extends
domain::AssetsBriefDocImpl
- serialVersionUID: long = -72475790815191... {readOnly}
+ AssetsBriefDocImpl()
+ getCreator() : String
+ getDataProvider() : String
+ getDebugQuery() : String
+ getFieldsEnum() : FieldDefEnum[]
+ getFullDocUrl() : String
+ getId() : String
+ getIndex() : int
+ getLanguage() : String
+ getProvider() : String
+ getScore() : int
+ getThumbnail() : String
+ getTitle() : String
+ getType() : DocType
+ getYear() : String
+ setDebugQuery(String) : void
+ setFullDocUrl(String) : void
+ setIndex(int) : void
+ setScore(int) : void

Figure 7 - Abstract Factory Implementation

4.2.1 ASSETS Data-Model Component

The main goal of the common data-model is to offer a common representation for the information exchanged between Assets services. The component implements an AbstractFactory pattern for the instantiation of the domain objects.

4.2.2 Core Data Model

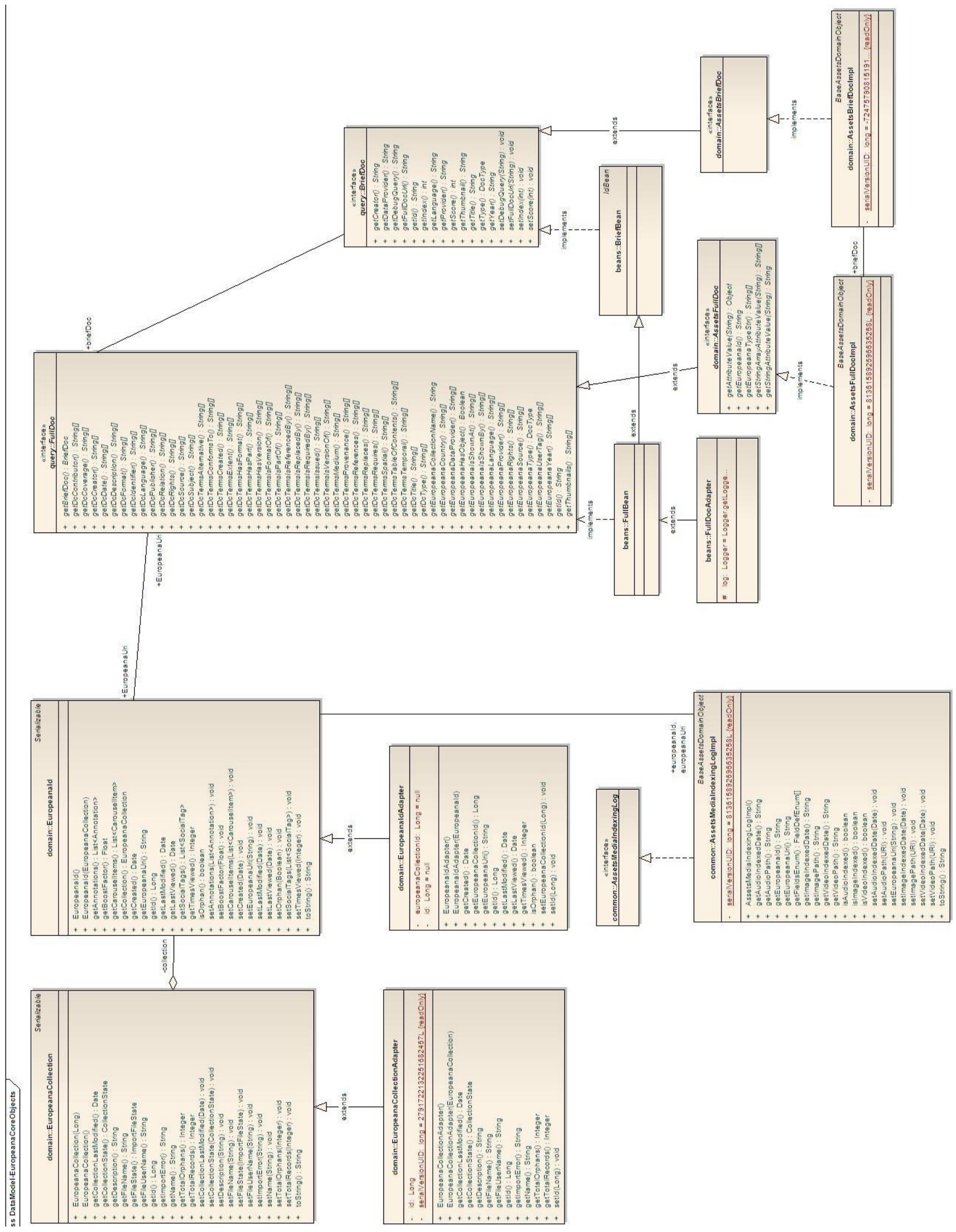
Assets common application layer offers access to the information managed by the Europeana application. This information is organized in collections provided by individual content providers (EuropeanaCollection objects), and the metadata containing the descriptions of the pasterpieces (FullDoc objects). Each object in the collection is identified by a set of properties which are grouped in EuropeanaId objects. In order to be able to expose these objects over the Rest interface, the Assets adapter classes enhance the core application objects by adding JAXB serialization annotations (EuropeanaCollectionAdapter, EuropeanaIdAdapter, FullDocAdapter). The metadata descriptions are stored into the Solr index for providing fast search and access in the Europeana Portal. Assets needs to process and persist these objects in a database, therefore the AssetsFullDoc representation of the objects was created. All Assets domain objects need to implement a common interface: AssetsDomainObject. The other objects of the Assets domain model are presented together with the components which are responsible for their management.

Interface Name	AssetsAbstractFactory
Key Concepts	Domain object, component factory
Operations	• **public AssetsDomainObject createDomainObject(String componentName, String domainObjectName)** - This method is used for the instantiation of the given domain objects from the given component

Interface Name	ComponentFactory
Key Concepts	Instantiation of current component domain objects
Operations	• **public AssetsDomainObject createDomainObject(String domainObjectName)** - This method creates an instance of the domain object identified by the given domain object name.

Interface Name	AssetsDomainObject
Key Concepts	Field enumeration
Operations	• **public String getId()** - Retrieve the identifier of the object stored in database • **public String getDomainObjectName()** - This method returns the logical

	name of the current domain object. By default, the simple classname will be used to as object name. • **public String getComponentName()** - This method returns the name of the component to which the current domain object belongs. • **public FieldDefEnum getFieldsEnum()** - This method returns the list with the name of the attributes which hold the information related to the current domain object.



ss DataModel-EuropeanaCoreObjects
«interface» query::BriefDoc
«interface» query::FullDoc
IdBean
beans::BriefBean
beans::FullBean
beans::FullDocAdapter
«interface» domain::AssetsBriefDoc
«interface» domain::AssetsFullDoc
BaseAssetsDomainObject
domain::AssetsBriefDocImpl
domain::AssetsFullDocImpl
Serializable
domain::EuropeanaId
domain::EuropeanaIdAdapter
«interface» common::AssetsMediaIndexingLog
common::AssetsMediaIndexingLogImpl
domain::EuropeanaCollection
domain::EuropeanaCollectionAdapter
extends
implements
+briefDoc
+EuropeanaUri
+europeanaId, europeanaUri
+collection

Figure 8 - Core data models

4.2.3 ASSETS Common API and Common Server API Components

The API components implement functionality that is common and should be reused by all Assets Services. There is functionality which is independent from the location at which it is used. For example, the reading of the configuration files, conversion between different textual representations of date information can be used without any restrictions in server-side and client-side components. This functionality is implemented in the "common-api" component.

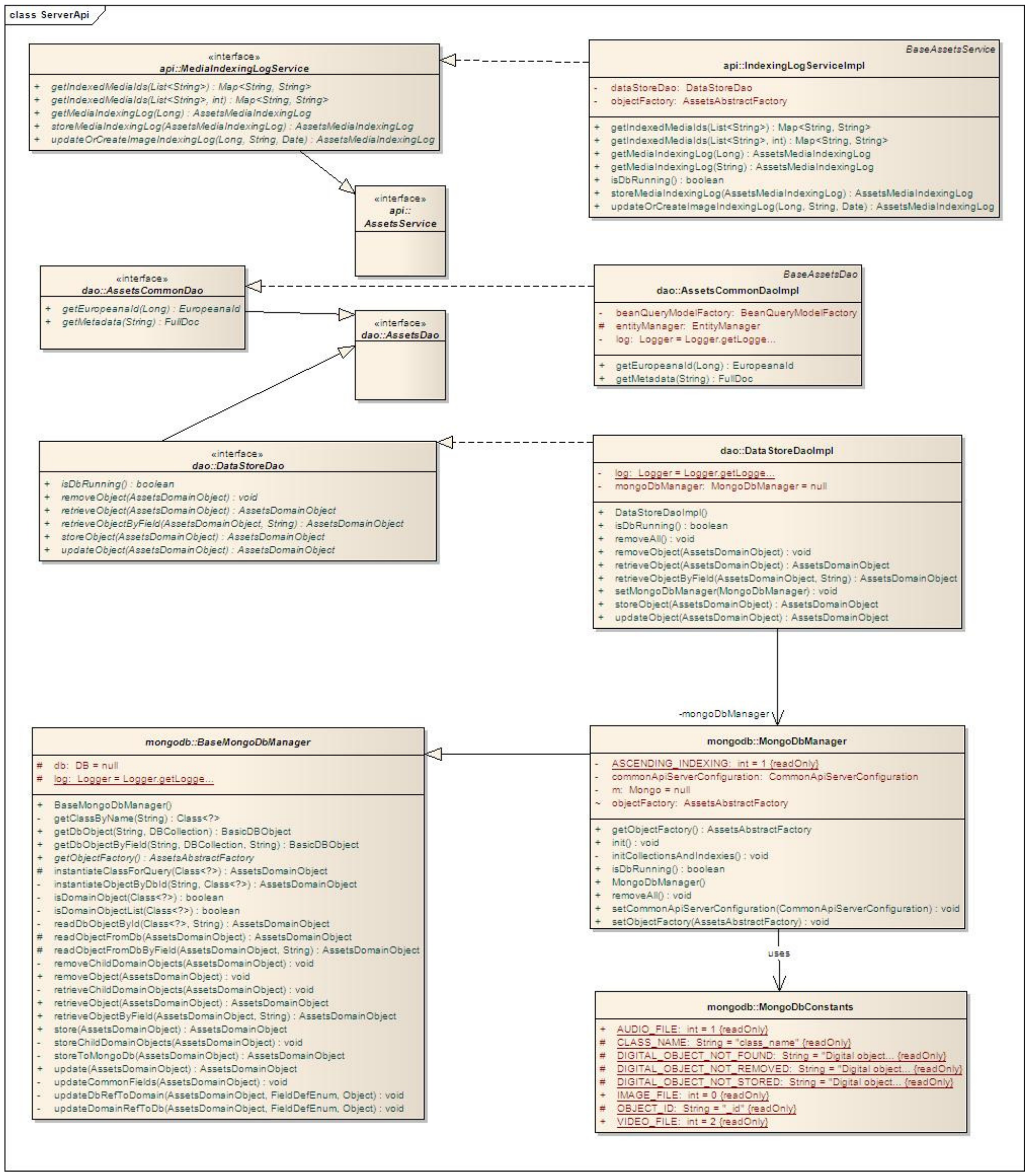
Further more, there is functionality which needs to access restricted or protected resources, like the persistence system. This functionality is made available only for being accessed on the server; therefore, it resides in the "common-api-server" component. In the current version of the system, this component implements a generic implementation of the MongoDb based DataStore and the logging for the media indexing functionality. Further common functionality will be identified during the implementation of the Assets services.

Service Name	Common Server API
Responsibility	1. Generic data Store 2. Logging support for media indexing
Provided Interfaces	1. DataStoreDao 2. MediaIndexingLogService
Dependencies	Common data model

Interface Name	DataStoreDao
Key Concepts	AssetsDomainObject
Operations	• **public AssetsDomainObject storeObject(AssetsDomainObject object)** - Stores the given domain object into the database • **public AssetsDomainObject retrieveObject(AssetsDomainObject object)** - Reads the object identified by the given object id from the database • **public AssetsDomainObject retrieveObjectByField(AssetsDomainObject object, String fieldName)** - Reads the object identified by the passed field from the database • **public AssetsDomainObject updateObject(AssetsDomainObject object)** - Updates the object identified by the given object id from the database • **public void removeObject(AssetsDomainObject object)** - This method removes the given object from the database • **public boolean isDbRunning() -** This utility method checks if the

	database connection can be established

Interface Name	MediaIndexingLogService
Key Concepts	AssetsDomainObject
Operations	• **public Map<String, String> getIndexedMediaIds(List<String> europeanaUris)** - This method evaluates the indexing log for media objects and returns a map of EuropeanaId.ids which are already available in the media index. • **public Map<String, String> getIndexedMediaIds(List<String> europeanaUris, int type)** - This method evaluates the indexing log for media objects and returns a map of EuropeanaId.ids which are already available in the media index. • **public AssetsMediaIndexingLog getMediaIndexingLog(Long europeanaId)** - This method returns the AssetsMediaIndexingLog for the given EuropeanaID.id • **public AssetsMediaIndexingLog storeMediaIndexingLog(AssetsMediaIndexingLog mediaIndexingLog)** - This method stores the AssetsMediaIndexingLog data representation in database. If the object already exists in the database it will be overriden with the current database. • **public AssetsMediaIndexingLog updateOrCreateImageIndexingLog(Long europeanaId, String europeanaUri, Date imageIndexingDate)** - This method updates the image indexing date for the object identified by the given europeanaId.



class ServerApi
«interface»
api::MediaIndexingLogService
+ getIndexedMediaIds(List<String>) : Map<String, String>
+ getIndexedMediaIds(List<String>, int) : Map<String, String>
+ getMediaIndexingLog(Long) : AssetsMediaIndexingLog
+ storeMediaIndexingLog(AssetsMediaIndexingLog) : AssetsMediaIndexingLog
+ updateOrCreateImageIndexingLog(Long, String, Date) : AssetsMediaIndexingLog
BaseAssetsService
api::IndexingLogServiceImpl
- dataStoreDao: DataStoreDao
- objectFactory: AssetsAbstractFactory
+ getIndexedMediaIds(List<String>) : Map<String, String>
+ getIndexedMediaIds(List<String>, int) : Map<String, String>
+ getMediaIndexingLog(Long) : AssetsMediaIndexingLog
+ getMediaIndexingLog(String) : AssetsMediaIndexingLog
+ isDbRunning() : boolean
+ storeMediaIndexingLog(AssetsMediaIndexingLog) : AssetsMediaIndexingLog
+ updateOrCreateImageIndexingLog(Long, String, Date) : AssetsMediaIndexingLog
«interface»
api::
AssetsService
«interface»
dao::AssetsCommonDao
+ getEuropeanaId(Long) : EuropeanaId
+ getMetadata(String) : FullDoc
«interface»
dao::AssetsDao
BaseAssetsDao
dao::AssetsCommonDaoImpl
- beanQueryModelFactory: BeanQueryModelFactory
# entityManager: EntityManager
- log: Logger = Logger.getLogge...
+ getEuropeanaId(Long) : EuropeanaId
+ getMetadata(String) : FullDoc
«interface»
dao::DataStoreDao
+ isDbRunning() : boolean
+ removeObject(AssetsDomainObject) : void
+ retrieveObject(AssetsDomainObject) : AssetsDomainObject
+ retrieveObjectByField(AssetsDomainObject, String) : AssetsDomainObject
+ storeObject(AssetsDomainObject) : AssetsDomainObject
+ updateObject(AssetsDomainObject) : AssetsDomainObject
dao::DataStoreDaoImpl
- log: Logger = Logger.getLogge...
- mongoDbManager: MongoDbManager = null
+ DataStoreDaoImpl()
+ isDbRunning() : boolean
+ removeAll() : void
+ removeObject(AssetsDomainObject) : void
+ retrieveObject(AssetsDomainObject) : AssetsDomainObject
+ retrieveObjectByField(AssetsDomainObject, String) : AssetsDomainObject
+ setMongoDbManager(MongoDbManager) : void
+ storeObject(AssetsDomainObject) : AssetsDomainObject
+ updateObject(AssetsDomainObject) : AssetsDomainObject
-mongoDbManager
mongodb::BaseMongoDbManager
# db: DB = null
# log: Logger = Logger.getLogge...
+ BaseMongoDbManager()
- getClassByName(String) : Class<?>
+ getDbObject(String, DBCollection) : BasicDBObject
+ getDbObjectByField(String, DBCollection, String) : BasicDBObject
+ getObjectFactory() : AssetsAbstractFactory
# instantiateClassForQuery(Class<?>) : AssetsDomainObject
- instantiateObjectByDbId(String, Class<?>) : AssetsDomainObject
- isDomainObject(Class<?>) : boolean
- isDomainObjectList(Class<?>) : boolean
- readDbObjectById(Class<?>, String) : AssetsDomainObject
# readObjectFromDb(AssetsDomainObject) : AssetsDomainObject
# readObjectFromDbByField(AssetsDomainObject, String) : AssetsDomainObject
- removeChildDomainObjects(AssetsDomainObject) : void
+ removeObject(AssetsDomainObject) : void
- retrieveChildDomainObjects(AssetsDomainObject) : void
+ retrieveObject(AssetsDomainObject) : AssetsDomainObject
+ retrieveObjectByField(AssetsDomainObject, String) : AssetsDomainObject
+ store(AssetsDomainObject) : AssetsDomainObject
- storeChildDomainObjects(AssetsDomainObject) : void
- storeToMongoDb(AssetsDomainObject) : AssetsDomainObject
+ update(AssetsDomainObject) : AssetsDomainObject
- updateCommonFields(AssetsDomainObject) : void
- updateDbRefToDomain(AssetsDomainObject, FieldDefEnum, Object) : void
- updateDomainRefToDb(AssetsDomainObject, FieldDefEnum, Object) : void
mongodb::MongoDbManager
- ASCENDING_INDEXING: int = 1 {readOnly}
- commonApiServerConfiguration: CommonApiServerConfiguration
- m: Mongo = null
~ objectFactory: AssetsAbstractFactory
+ getObjectFactory() : AssetsAbstractFactory
+ init() : void
- initCollectionsAndIndexies() : void
+ isDbRunning() : boolean
+ MongoDbManager()
+ removeAll() : void
+ setCommonApiServerConfiguration(CommonApiServerConfiguration) : void
+ setObjectFactory(AssetsAbstractFactory) : void
uses
mongodb::MongoDbConstants
+ AUDIO_FILE: int = 1 {readOnly}
# CLASS_NAME: String = "class_name" {readOnly}
# DIGITAL_OBJECT_NOT_FOUND: String = "Digital object... {readOnly}
# DIGITAL_OBJECT_NOT_REMOVED: String = "Digital object... {readOnly}
# DIGITAL_OBJECT_NOT_STORED: String = "Digital object... {readOnly}
+ IMAGE_FILE: int = 0 {readOnly}
# OBJECT_ID: String = "_id" {readOnly}
+ VIDEO_FILE: int = 2 {readOnly}

Figure 9 – Common Server API

4.2.4 Common Server and Common Client

The common layer of Assets architecture follows the same structure as the regular components. This layer is also provided with a REST and a client API, which makes available the core information of the assets system to the other components. The business functionality provided in the Server, Rest and Client interfaces is the same; therefore we will describe in the followings only the Server interface which is the most important one.

Service Name	Metadata Management Service
Responsibility	1. Define a unified representation for the assets domain model 2. Instantiate domain objects
Provided Interfaces	1. MetadataManagementService, 2. CommonRest, 3. DataManagement
Dependencies	Europeana Core Data – Model

Interface Name	MetadataManagementService
Key Concepts	Collection, CollectionObject, Metadata
Operations	• **public Integer getCollectionCount() -** This method returns the number of collections available into the database • **public EuropeanaCollection getCollection(Long id)** - This method returns the collection identified by the given id • **public List<EuropeanaCollection> getCollections() –** Fetch all collections. • **public List<EuropeanaId> getCollectionObjects(EuropeanaCollection collection)** - The list of Europeana ids available in the collection • **public EuropeanaId getCollectionObject(Long id)** - This method returns the EuropeanaId object identified by the given database id • **public FullDoc getMetadataFromSolr(EuropeanaId euId)** - This method retrieves the metadata of the collection object from the SolrIndex • **public AssetsFullDoc getMetadata(Long euId)** - This method retrieves the metadata of the collection object from the database • **public AssetsFullDoc storeMetadata(AssetsFullDoc afd)** - This method stores the FullDoc metadata representation in database. If the object already exists in the database it will be overriden with the current database. • **public Long getCollectionObjectId(String europeanaUri)** - This method retrieves the ID of the EuropeanaId object identified by the given URI

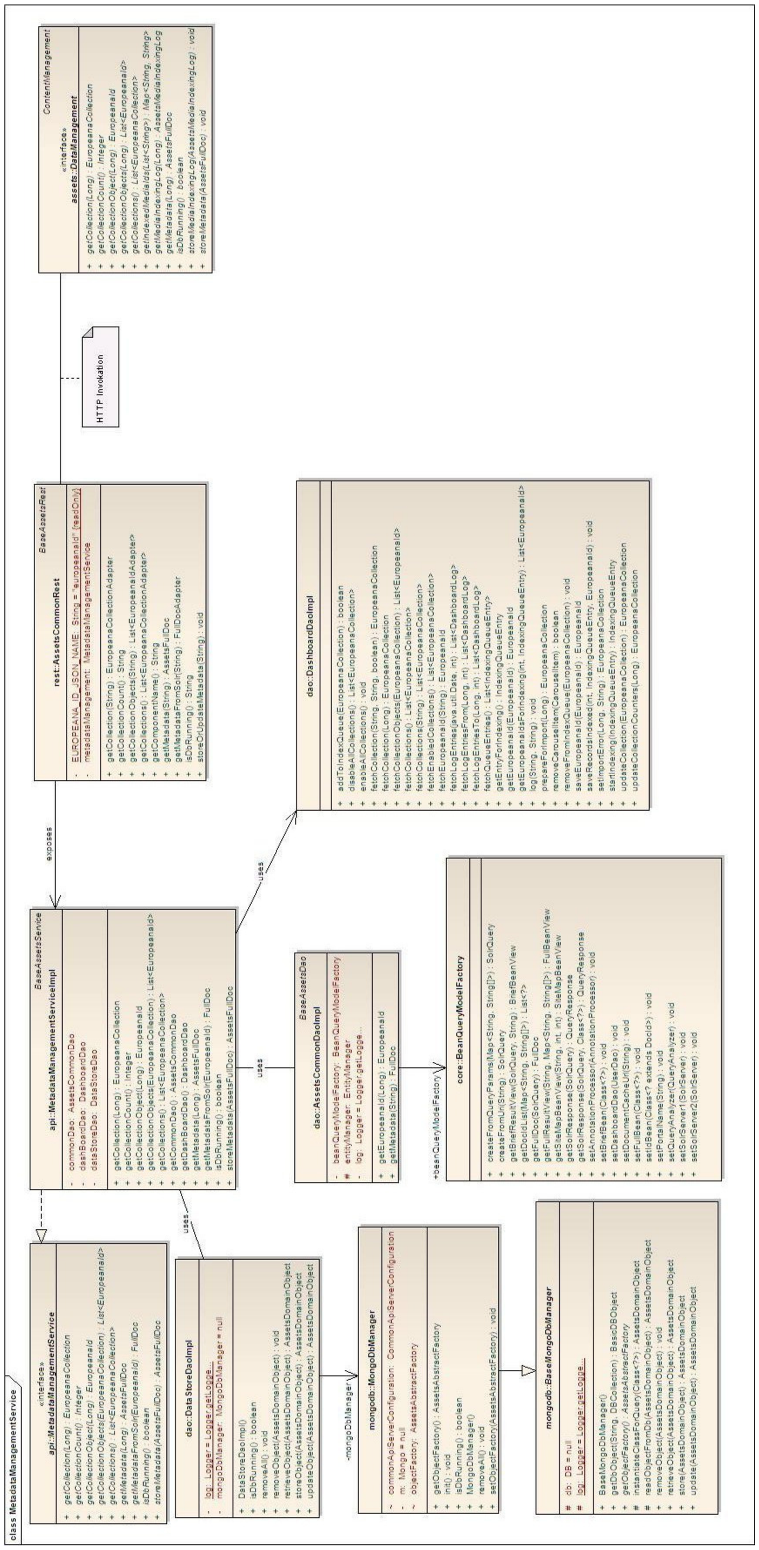


Figure 10 - Metadata Management service

4.3 The Ingestion Models and Interfaces

4.3.1 The Metadata Cleaning Service Models and Interfaces

The metadata cleaning service has the responsibility of managing and performing basic error correction, normalization and cleaning tasks.

Service Name	Metadata Cleaning Service
Responsibility	1. Basic error correction 2. Value normalization 3. Basic enrichment.
Provided Interfaces	1. MetadataCleaningManager 2. MetadataErrorCorrection 3. MetadataValueNormalization 4. MetadataFieldEnrichment
Dependencies	ASSETS common, other modules and services used during the ingestion stage

The manager interfaces allows to query the service for the available correction/normalization models and resources (e.g., authority files and controlled vocabularies), and to train new models by providing training examples.

Interface Name	MetadataCleaningManager
Key Concepts	MetadataErrorCorrection, MetadataValueNormalization, MetadataFieldEnrichment
Operations	• TrainMetadataErrorCorrector – trains an error correction model from a training set of example composed of pair of metadata records describing the metadata record before and after the correction. • TrainMetadataValueNormalizer – trains a value normalization model from a training set of example composed of pair of metadata records describing the metadata record before and after the value normalization. • GetStatus – polls the service to obtain the status of a training process. • ListMetadataErrorCorrectors - returns a list of the available models trained to perform error correction. • DeleteMetadataErrorCorrector – deletes an error correction model • ListMetadataValueNormalizers - returns a list of the available

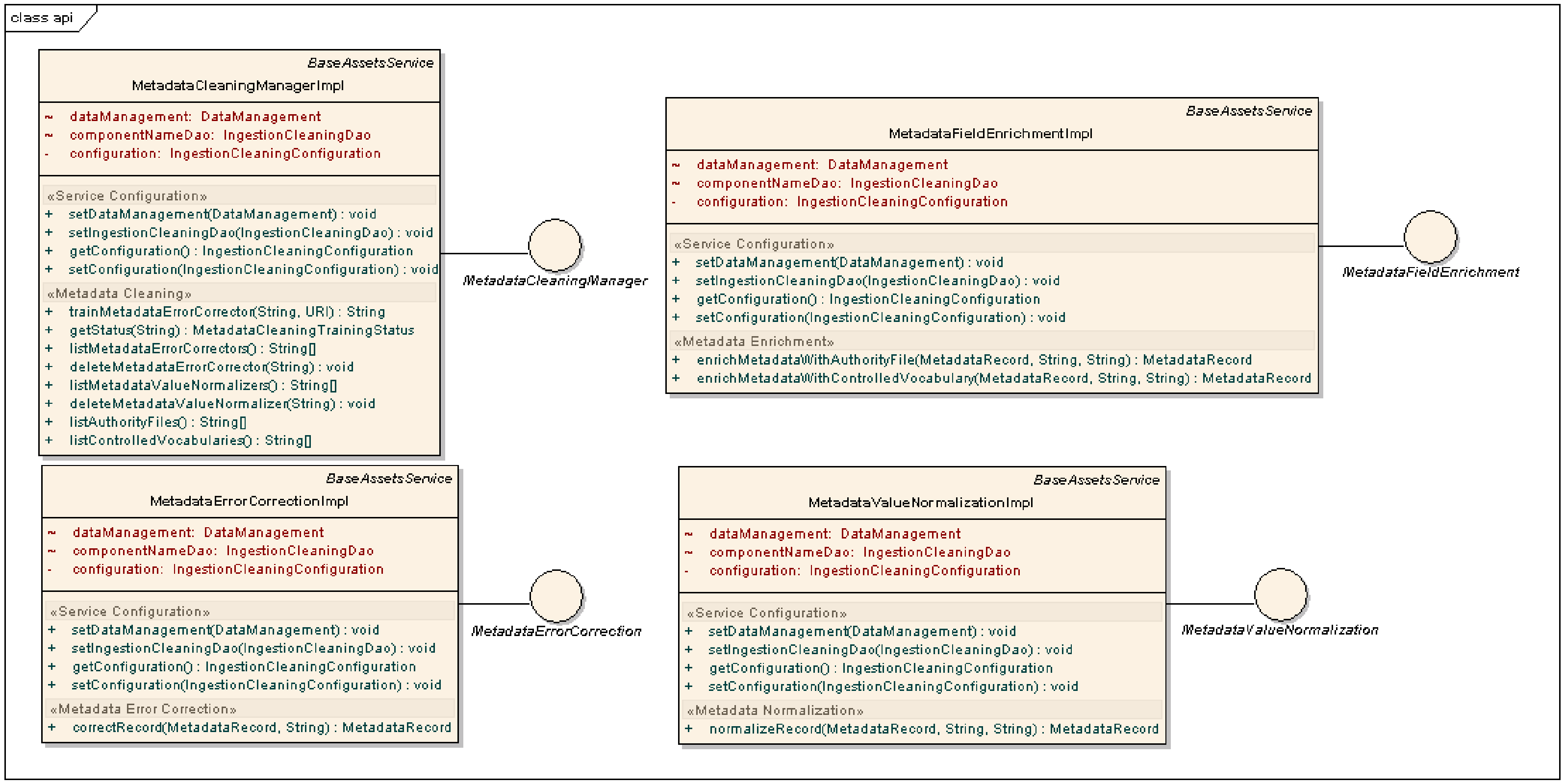
	models trained to perform value normalization • DeleteMetadataValueNormalizer – deletes an error value normalization model • ListAuthorityFiles - lists the available authority files • ListControlledVocabularies - lists the available controlled vocabularies

The following three interfaces provide the actual methods to process the metadata records during the ingestion process.

Interface Name	MetadataErrorCorrection
Key Concepts	MetadataErrorCorrectionDescriptor
Operations	• CorrectRecord – takes in input a record and the name of an error correction model, returns an automatically corrected metadata record according to the correction model

Interface Name	MetadataValueNormalization
Key Concepts	MetadataValueNormalizerDescriptor
Operations	• NormalizeRecord - takes in input a record and the name of a value normalization model, returns an automatically normalized metadata record according to the normalization model

Interface Name	MetadataFieldEnrichment
Key Concepts	MetadataFieldEnricherDescriptor
Operations	• EnrichMetadataWithAuthorityFile - takes in input a record and the name of an authority file, returns an automatically enriched record according to the authorithy file • enrichMetadataWithControlledVocabulary - takes in input a record and the name of a controlled vocabulary, returns an automatically enriched record according to the controlled vocabulary



```
BaseAssetsService
MetadataCleaningManagerImpl
~ dataManagement: DataManagement
~ componentNameDao: IngestionCleaningDao
- configuration: IngestionCleaningConfiguration
«Service Configuration»
+ setDataManagement(DataManagement) : void
+ setIngestionCleaningDao(IngestionCleaningDao) : void
+ getConfiguration() : IngestionCleaningConfiguration
+ setConfiguration(IngestionCleaningConfiguration) : void
«Metadata Cleaning»
+ trainMetadataErrorCorrector(String, URI) : String
+ getStatus(String) : MetadataCleaningTrainingStatus
+ listMetadataErrorCorrectors() : String[]
+ deleteMetadataErrorCorrector(String) : void
+ listMetadataValueNormalizers() : String[]
+ deleteMetadataValueNormalizer(String) : void
+ listAuthorityFiles() : String[]
+ listControlledVocabularies() : String[]
```

```
BaseAssetsService
MetadataFieldEnrichmentImpl
~ dataManagement: DataManagement
~ componentNameDao: IngestionCleaningDao
- configuration: IngestionCleaningConfiguration
«Service Configuration»
+ setDataManagement(DataManagement) : void
+ setIngestionCleaningDao(IngestionCleaningDao) : void
+ getConfiguration() : IngestionCleaningConfiguration
+ setConfiguration(IngestionCleaningConfiguration) : void
«Metadata Enrichment»
+ enrichMetadataWithAuthorityFile(MetadataRecord, String, String) : MetadataRecord
+ enrichMetadataWithControlledVocabulary(MetadataRecord, String, String) : MetadataRecord
```

```
BaseAssetsService
MetadataErrorCorrectionImpl
~ dataManagement: DataManagement
~ componentNameDao: IngestionCleaningDao
- configuration: IngestionCleaningConfiguration
«Service Configuration»
+ setDataManagement(DataManagement) : void
+ setIngestionCleaningDao(IngestionCleaningDao) : void
+ getConfiguration() : IngestionCleaningConfiguration
+ setConfiguration(IngestionCleaningConfiguration) : void
«Metadata Error Correction»
+ correctRecord(MetadataRecord, String) : MetadataRecord
```

```
BaseAssetsService
MetadataValueNormalizationImpl
~ dataManagement: DataManagement
~ componentNameDao: IngestionCleaningDao
- configuration: IngestionCleaningConfiguration
«Service Configuration»
+ setDataManagement(DataManagement) : void
+ setIngestionCleaningDao(IngestionCleaningDao) : void
+ getConfiguration() : IngestionCleaningConfiguration
+ setConfiguration(IngestionCleaningConfiguration) : void
«Metadata Normalization»
+ normalizeRecord(MetadataRecord, String, String) : MetadataRecord
```

Figure 11 - Ingestion Cleaning API: Overview

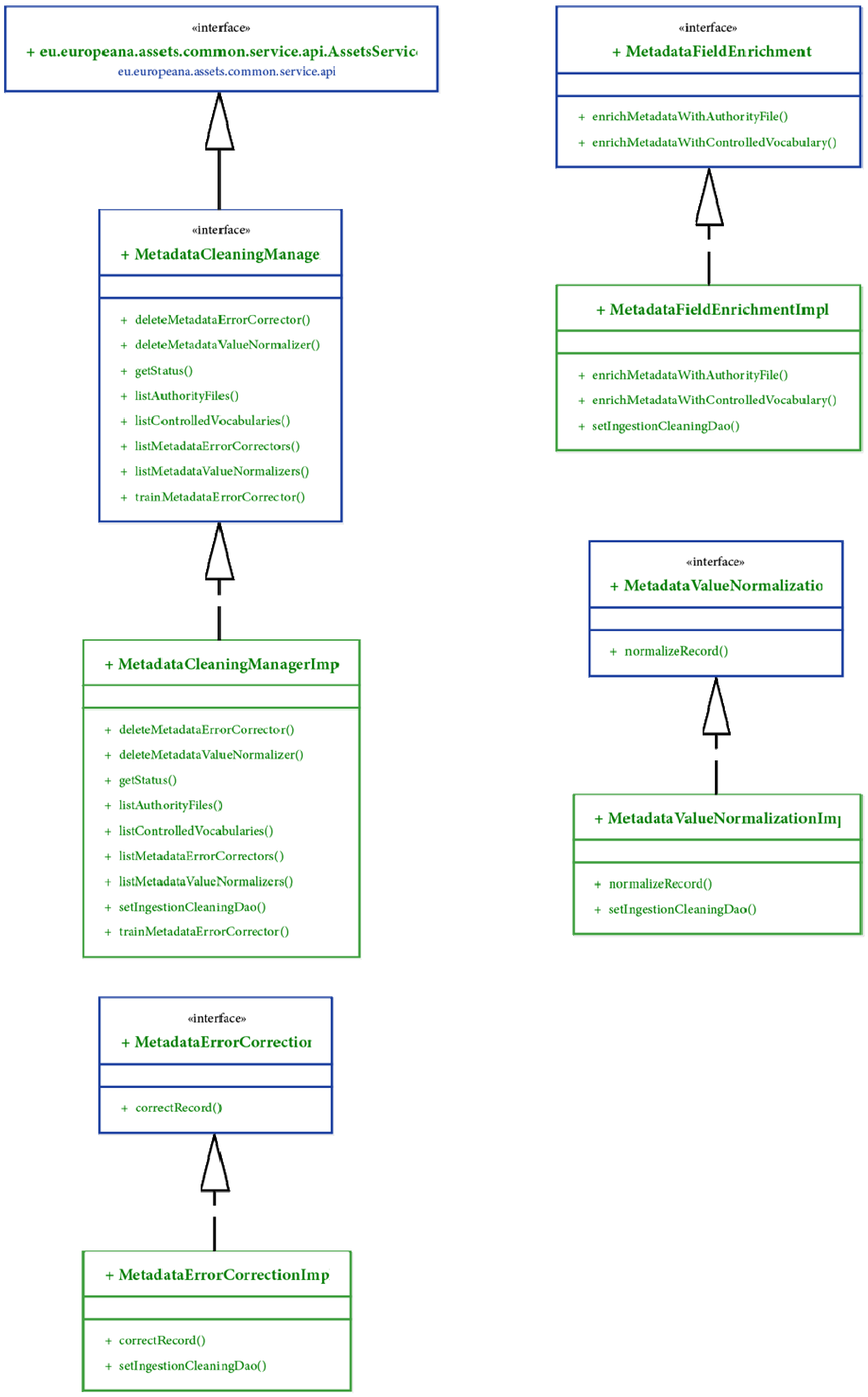

«interface»
+ eu.europeana.assets.common.service.api.AssetsServic
eu.europeana.assets.common.service.api
«interface»
+ MetadataCleaningManage
+ deleteMetadataErrorCorrector()
+ deleteMetadataValueNormalizer()
+ getStatus()
+ listAuthorityFiles()
+ listControlledVocabularies()
+ listMetadataErrorCorrectors()
+ listMetadataValueNormalizers()
+ trainMetadataErrorCorrector()
+ MetadataCleaningManagerImp
+ deleteMetadataErrorCorrector()
+ deleteMetadataValueNormalizer()
+ getStatus()
+ listAuthorityFiles()
+ listControlledVocabularies()
+ listMetadataErrorCorrectors()
+ listMetadataValueNormalizers()
+ setIngestionCleaningDao()
+ trainMetadataErrorCorrector()
«interface»
+ MetadataErrorCorrectior
+ correctRecord()
+ MetadataErrorCorrectionImp
+ correctRecord()
+ setIngestionCleaningDao()
«interface»
+ MetadataFieldEnrichment
+ enrichMetadataWithAuthorityFile()
+ enrichMetadataWithControlledVocabulary()
+ MetadataFieldEnrichmentImpl
+ enrichMetadataWithAuthorityFile()
+ enrichMetadataWithControlledVocabulary()
+ setIngestionCleaningDao()
«interface»
+ MetadataValueNormalizatio
+ normalizeRecord()
+ MetadataValueNormalizationIm
+ normalizeRecord()
+ setIngestionCleaningDao()

Figure 12 – Ingestion Cleaning API: details

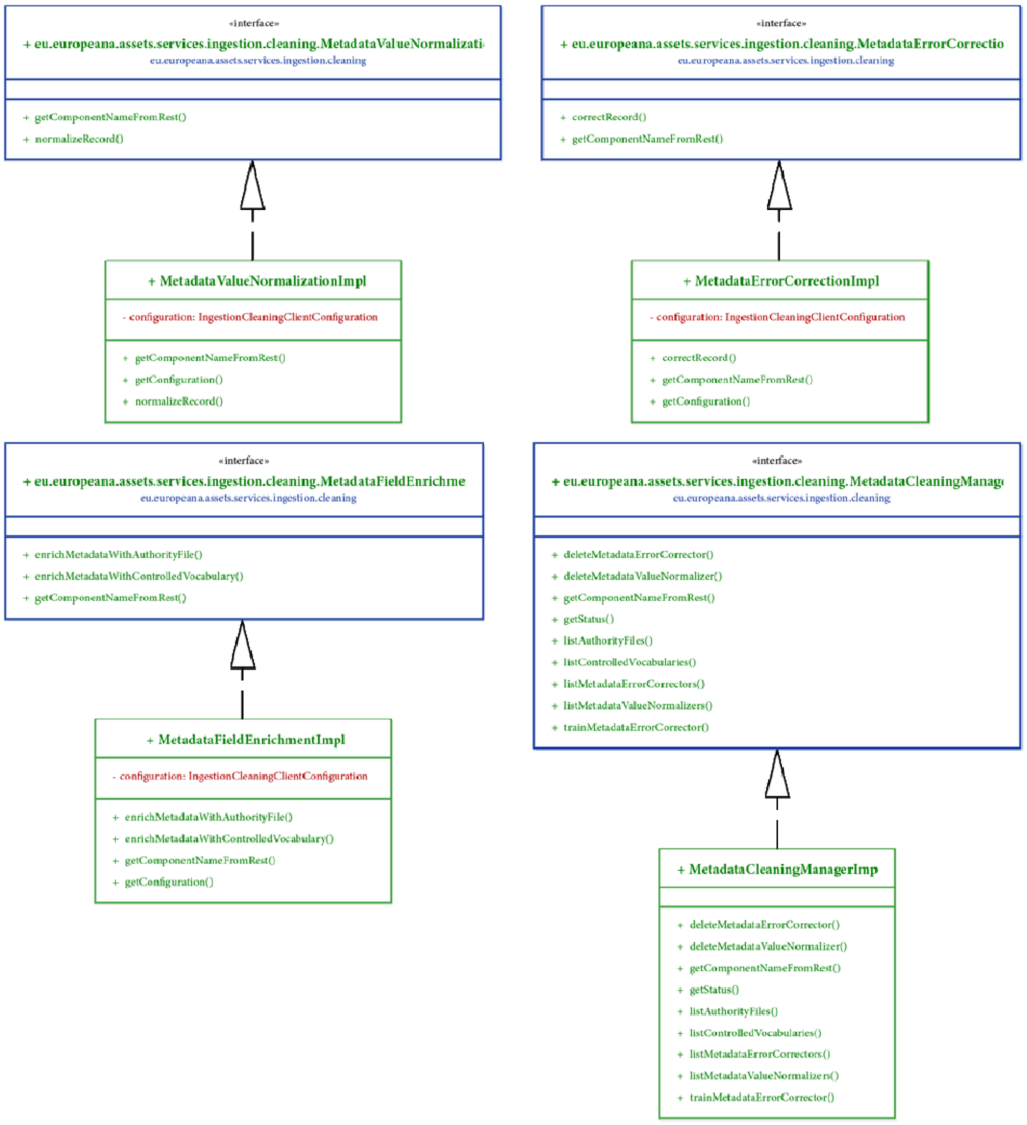

«interface»
+ eu.europeana.assets.services.ingestion.cleaning.MetadataValueNormalizati
eu.europeana.assets.services.ingestion.cleaning
+ getComponentNameFromRest()
+ normalizeRecord()
«interface»
+ eu.europeana.assets.services.ingestion.cleaning.MetadataErrorCorrectio
eu.europeana.assets.services.ingestion.cleaning
+ correctRecord()
+ getComponentNameFromRest()
+ MetadataValueNormalizationImpl
- configuration: IngestionCleaningClientConfiguration
+ getComponentNameFromRest()
+ getConfiguration()
+ normalizeRecord()
+ MetadataErrorCorrectionImpl
- configuration: IngestionCleaningClientConfiguration
+ correctRecord()
+ getComponentNameFromRest()
+ getConfiguration()
«interface»
+ eu.europeana.assets.services.ingestion.cleaning.MetadataFieldEnrichme
eu.europeana.assets.services.ingestion.cleaning
+ enrichMetadataWithAuthorityFile()
+ enrichMetadataWithControlledVocabulary()
+ getComponentNameFromRest()
«interface»
+ eu.europeana.assets.services.ingestion.cleaning.MetadataCleaningManag
eu.europeana.assets.services.ingestion.cleaning
+ deleteMetadataErrorCorrector()
+ deleteMetadataValueNormalizer()
+ getComponentNameFromRest()
+ getStatus()
+ listAuthorityFiles()
+ listControlledVocabularies()
+ listMetadataErrorCorrectors()
+ listMetadataValueNormalizers()
+ trainMetadataErrorCorrector()
+ MetadataFieldEnrichmentImpl
- configuration: IngestionCleaningClientConfiguration
+ enrichMetadataWithAuthorityFile()
+ enrichMetadataWithControlledVocabulary()
+ getComponentNameFromRest()
+ getConfiguration()
+ MetadataCleaningManagerImp
+ deleteMetadataErrorCorrector()
+ deleteMetadataValueNormalizer()
+ getComponentNameFromRest()
+ getStatus()
+ listAuthorityFiles()
+ listControlledVocabularies()
+ listMetadataErrorCorrectors()
+ listMetadataValueNormalizers()
+ trainMetadataErrorCorrector()

Figure 13 – Ingestion Cleaning : Client Side Models

4.3.2 *Knowledge Extraction Models and Interfaces*

The Knowledge Extraction service has the responsibility of managing and performing extraction of structured information from pieces of unstructured text contained in metadata records.

Service Name	Knowledge Extraction
Responsibility	1. Extraction of structured information from unstructured textual metadata fields of metadata records
Provided Interfaces	1. KnowedgeExtractionTrainer, 2. KnowledgeExtractionManager, 3. KnowledgeExtractor
Dependencies	ASSETS common, other modules and services used during the ingestion stage

The KnowledgeExtractionTrainer interface provides the backend with methods for the creation of knowledge extraction models, while the KnowedgeExtractionManager interface provides the ingestion workflow with methods to access them.

Interface Name	KnowledgeExtractionTrainer
Key Concepts	MetadataKnowedgeExtractionTrainingSet, MetadataKnowledgeExtractionModel
Operations	• TrainMetadataKnowledgeExtractor - trains a knowledge extraction model from a training set of examples consisting of a list of metadata records in which the relevant information to be extracted has been manually annotated. • GetTrainingStatus - polls the service to obtain the status of a training process.

Interface Name	KnowedgeExtractionManager
Key Concepts	MetadataKnowledgeExtractionModel, KnowledgeExtractorDescriptor, KnowledgeExtractor
Operations	• ListMetadataKnowledgeExtractor – returns a list of the available knowledge extraction models. • DeleteMetadataKnowledgeExtractor – deletes a knowledge extraction model. • GetKnowledgeExtractorDescriptor – load and returns a knowledge extraction model.
Interface Name	KnowledgeExtractor
Key Concepts	MetadataDataset, MetadataKnoledgeExtractionModel

Operations	• ExtractKnowledgeFromMetadata – applies a knowledge extraction model to a metadata record, returns a new version of it with additional information generated by the automatic knowledge extraction process.

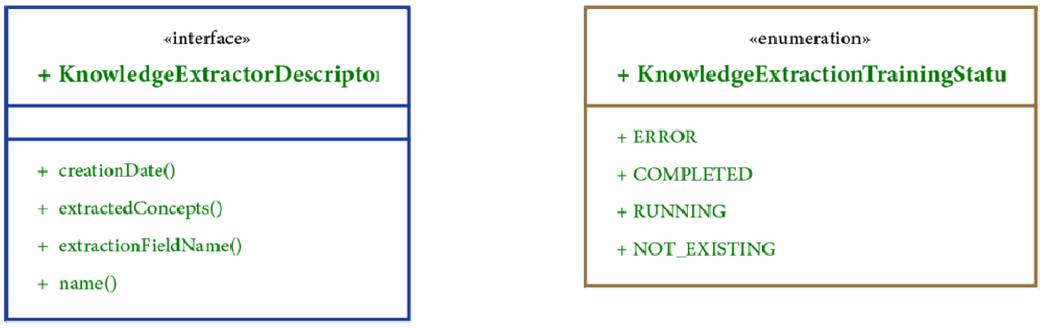


Figure 14 – Ingestion Knowledge Extraction Data Model

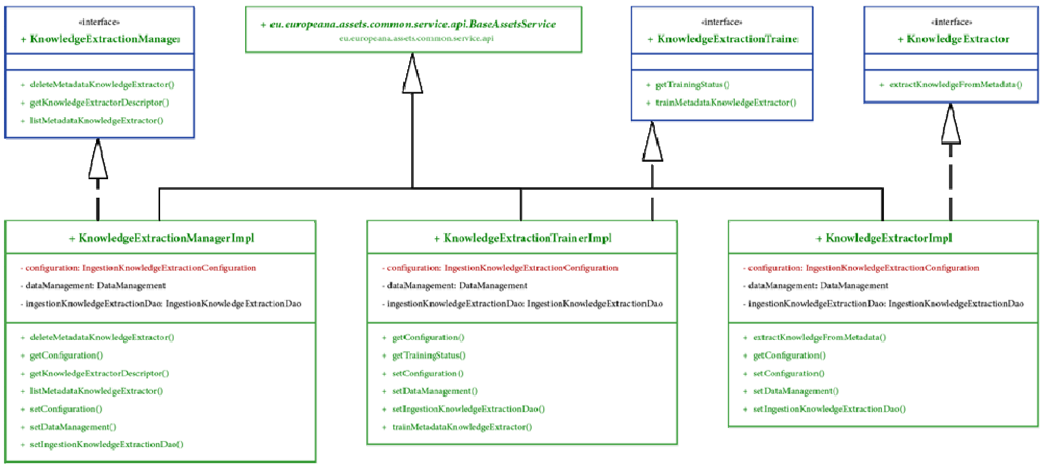

«interface»
+ KnowledgeExtractionManager
+ deleteMetadataKnowledgeExtractor()
+ getKnowledgeExtractorDescriptor()
+ listMetadataKnowledgeExtractor()
+ eu.europeana.assets.common.service.api.BaseAssetsService
eu.europeana.assets.common.service.api
«interface»
+ KnowledgeExtractionTrainer
+ getTrainingStatus()
+ trainMetadataKnowledgeExtractor()
«interface»
+ KnowledgeExtractor
+ extractKnowledgeFromMetadata()
+ KnowledgeExtractionManagerImpl
- configuration: IngestionKnowledgeExtractionConfiguration
- dataManagement: DataManagement
- ingestionKnowledgeExtractionDao: IngestionKnowledgeExtractionDao
+ deleteMetadataKnowledgeExtractor()
+ getConfiguration()
+ getKnowledgeExtractorDescriptor()
+ listMetadataKnowledgeExtractor()
+ setConfiguration()
+ setDataManagement()
+ setIngestionKnowledgeExtractionDao()
+ KnowledgeExtractionTrainerImpl
- configuration: IngestionKnowledgeExtractionConfiguration
- dataManagement: DataManagement
- ingestionKnowledgeExtractionDao: IngestionKnowledgeExtractionDao
+ getConfiguration()
+ getTrainingStatus()
+ setConfiguration()
+ setDataManagement()
+ setIngestionKnowledgeExtractionDao()
+ trainMetadataKnowledgeExtractor()
+ KnowledgeExtractorImpl
- configuration: IngestionKnowledgeExtractionConfiguration
- dataManagement: DataManagement
- ingestionKnowledgeExtractionDao: IngestionKnowledgeExtractionDao
+ extractKnowledgeFromMetadata()
+ getConfiguration()
+ setConfiguration()
+ setDataManagement()
+ setIngestionKnowledgeExtractionDao()

Figure 15 - Ingestion Knowledge Extraction API

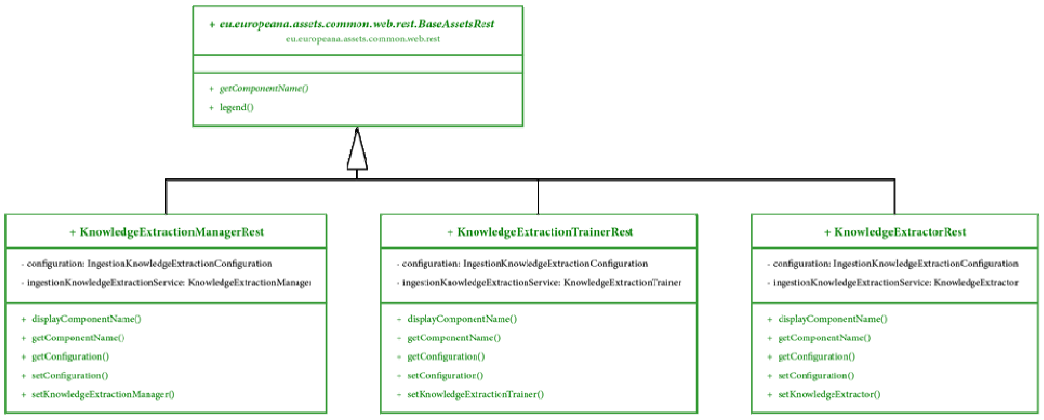

+ eu.europeana.assets.common.web.rest.BaseAssetsRest
eu.europeana.assets.common.web.rest
+ getComponentName()
+ legend()
+ KnowledgeExtractionManagerRest
- configuration: IngestionKnowledgeExtractionConfiguration
- ingestionKnowledgeExtractionService: KnowledgeExtractionManager
+ displayComponentName()
+ getComponentName()
+ getConfiguration()
+ setConfiguration()
+ setKnowledgeExtractionManager()
+ KnowledgeExtractionTrainerRest
- configuration: IngestionKnowledgeExtractionConfiguration
- ingestionKnowledgeExtractionService: KnowledgeExtractionTrainer
+ displayComponentName()
+ getComponentName()
+ getConfiguration()
+ setConfiguration()
+ setKnowledgeExtractionTrainer()
+ KnowledgeExtractorRest
- configuration: IngestionKnowledgeExtractionConfiguration
- ingestionKnowledgeExtractionService: KnowledgeExtractor
+ displayComponentName()
+ getComponentName()
+ getConfiguration()
+ setConfiguration()
+ setKnowledgeExtractor()

Figure 16 - Ingestion Knowledge Extraction REST API

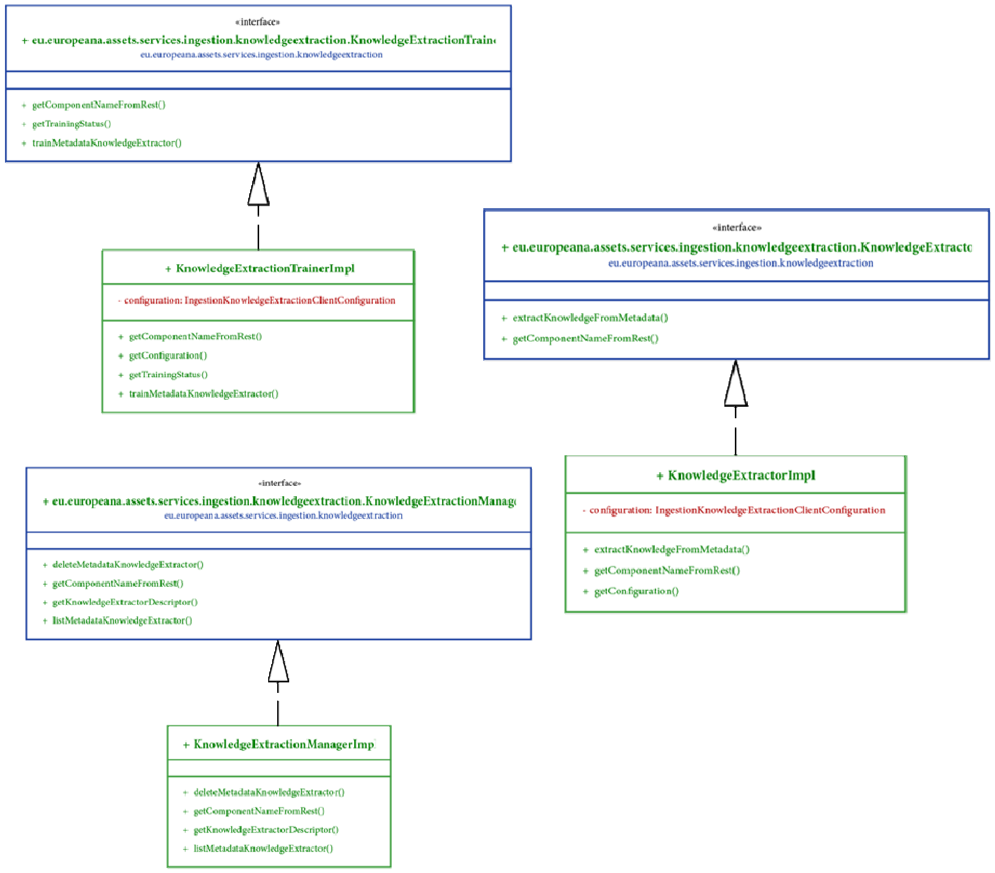

«interface»
+ eu.europeana.assets.services.ingestion.knowledgeextraction.KnowledgeExtractionTrain
eu.europeana.assets.services.ingestion.knowledgeextraction
+ getComponentNameFromRest()
+ getTrainingStatus()
+ trainMetadataKnowledgeExtractor()
+ KnowledgeExtractionTrainerImpl
- configuration: IngestionKnowledgeExtractionClientConfiguration
+ getComponentNameFromRest()
+ getConfiguration()
+ getTrainingStatus()
+ trainMetadataKnowledgeExtractor()
«interface»
+ eu.europeana.assets.services.ingestion.knowledgeextraction.KnowledgeExtracto
eu.europeana.assets.services.ingestion.knowledgeextraction
+ extractKnowledgeFromMetadata()
+ getComponentNameFromRest()
+ KnowledgeExtractorImpl
- configuration: IngestionKnowledgeExtractionClientConfiguration
+ extractKnowledgeFromMetadata()
+ getComponentNameFromRest()
+ getConfiguration()
«interface»
+ eu.europeana.assets.services.ingestion.knowledgeextraction.KnowledgeExtractionManag
eu.europeana.assets.services.ingestion.knowledgeextraction
+ deleteMetadataKnowledgeExtractor()
+ getComponentNameFromRest()
+ getKnowledgeExtractorDescriptor()
+ listMetadataKnowledgeExtractor()
+ KnowledgeExtractionManagerImp
+ deleteMetadataKnowledgeExtractor()
+ getComponentNameFromRest()
+ getKnowledgeExtractorDescriptor()
+ listMetadataKnowledgeExtractor()

Figure 17 - Ingestion Knowledge Extraction Client

4.3.3 *Metadata Classification Models and Interfaces*

The Metadata Classification service has the responsibility of managing and performing classification metadata records with respect to a relevant taxonomy.

Service Name	Metadata Classification
Responsibility	1. Classification of europeana metadata records on relevant taxonomies
Provided Interfaces	1. ClassificationTrainer, 2. ClassificationManager, 3. ClassificationService
Dependencies	ASSETS common, other modules and services used during the ingestion stage

The ClassificationTrainer interface provides the backend with methods for the creation of classification models, while the ClassificationManager interface provides the ingestion workflow with methods to access them.

Interface Name	ClassificationTrainer
Key Concepts	MetadataClassificationTrainingSet, MetadataClassificationModel
Operations	• TrainMetadataClassifier - trains a classification model from a training set of examples consisting of a specification of a taxonomy and a list of metadata records each one manually classified with respect to such taxonomy. • GetTrainingStatus - polls the service to obtain the status of a training process.

Interface Name	ClassificationManager, ClassificationService
Key Concepts	MetadataClassificationModel
Operations	• ListMetadataClassifier – returns a list of the available classification models. • DeleteMetadataClassifier - deletes a classification model. • GetClassificationService – load and returns a classification model.

Interface Name	ClassificationService
Key Concepts	MetadataDataset, MetadataClassificationModel
Operations	• ClassifyMetadata – applies a classification model to a metadata record, returns a new version of it with additional information specifying the taxonomy labels assigned by the automatic

	classification process.

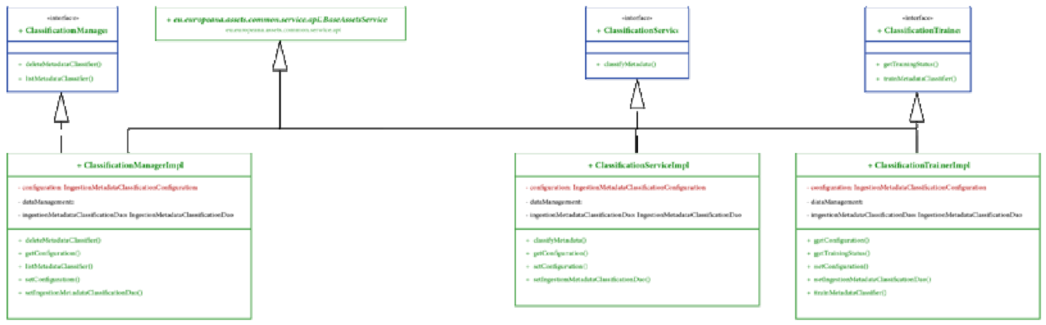

«interface»
+ ClassificationManager
+ deleteMetadataClassifier()
+ listMetadataClassifier()
+ eu.europeana.assets.common.service.api.BaseAssetsService
eu.europeana.assets.common.service.api
«interface»
+ ClassificationService
+ classifyMetadata()
«interface»
+ ClassificationTrainer
+ getTrainingStatus()
+ trainMetadataClassifier()
+ ClassificationManagerImpl
- configuration: IngestionMetadataClassificationConfiguration
- dataManagement:
- ingestionMetadataClassificationDao: IngestionMetadataClassificationDao
+ deleteMetadataClassifier()
+ getConfiguration()
+ listMetadataClassifier()
+ setConfiguration()
+ setIngestionMetadataClassificationDao()
+ ClassificationServiceImpl
- configuration: IngestionMetadataClassificationConfiguration
- dataManagement:
- ingestionMetadataClassificationDao: IngestionMetadataClassificationDao
+ classifyMetadata()
+ getConfiguration()
+ setConfiguration()
+ setIngestionMetadataClassificationDao()
+ ClassificationTrainerImpl
- configuration: IngestionMetadataClassificationConfiguration
- dataManagement:
- ingestionMetadataClassificationDao: IngestionMetadataClassificationDao
+ getConfiguration()
+ getTrainingStatus()
+ setConfiguration()
+ setIngestionMetadataClassificationDao()
+ trainMetadataClassifier()

Figure 18 – Ingestion Metadata Classification Service API

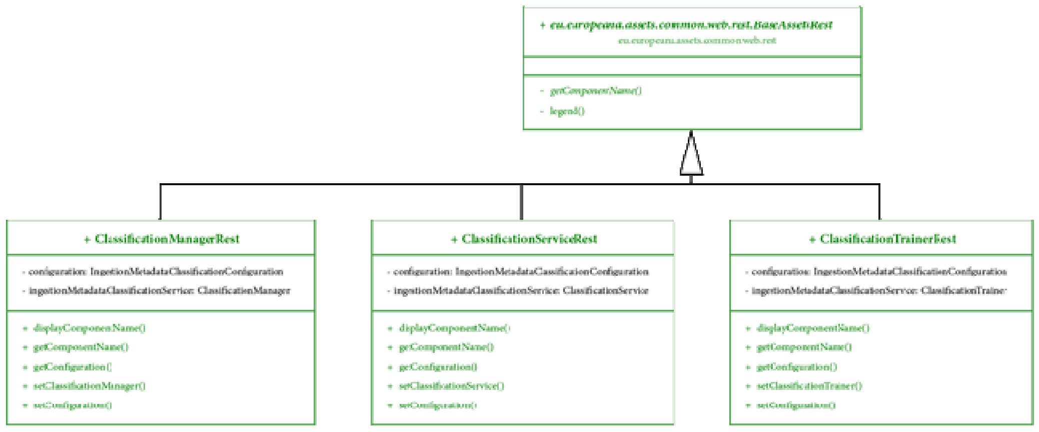

+ eu.europeana.assets.common.web.rest.BaseAssetsRest
eu.europeana.assets.common.web.rest
- getComponentName()
- legend()
+ ClassificationManagerRest
- configuration: IngestionMetadataClassificationConfiguration
- ingestionMetadataClassificationService: ClassificationManager
+ displayComponentName()
+ getComponentName()
+ getConfiguration()
+ setClassificationManager()
+ setConfiguration()
+ ClassificationServiceRest
- configuration: IngestionMetadataClassificationConfiguration
- ingestionMetadataClassificationService: ClassificationService
+ displayComponentName()
+ getComponentName()
+ getConfiguration()
+ setClassificationService()
+ setConfiguration()
+ ClassificationTrainerRest
- configuration: IngestionMetadataClassificationConfiguration
- ingestionMetadataClassificationService: ClassificationTrainer
+ displayComponentName()
+ getComponentName()
+ getConfiguration()
+ setClassificationTrainer()
+ setConfiguration()

Figure 19 - Ingestion Metadata Classification REST API

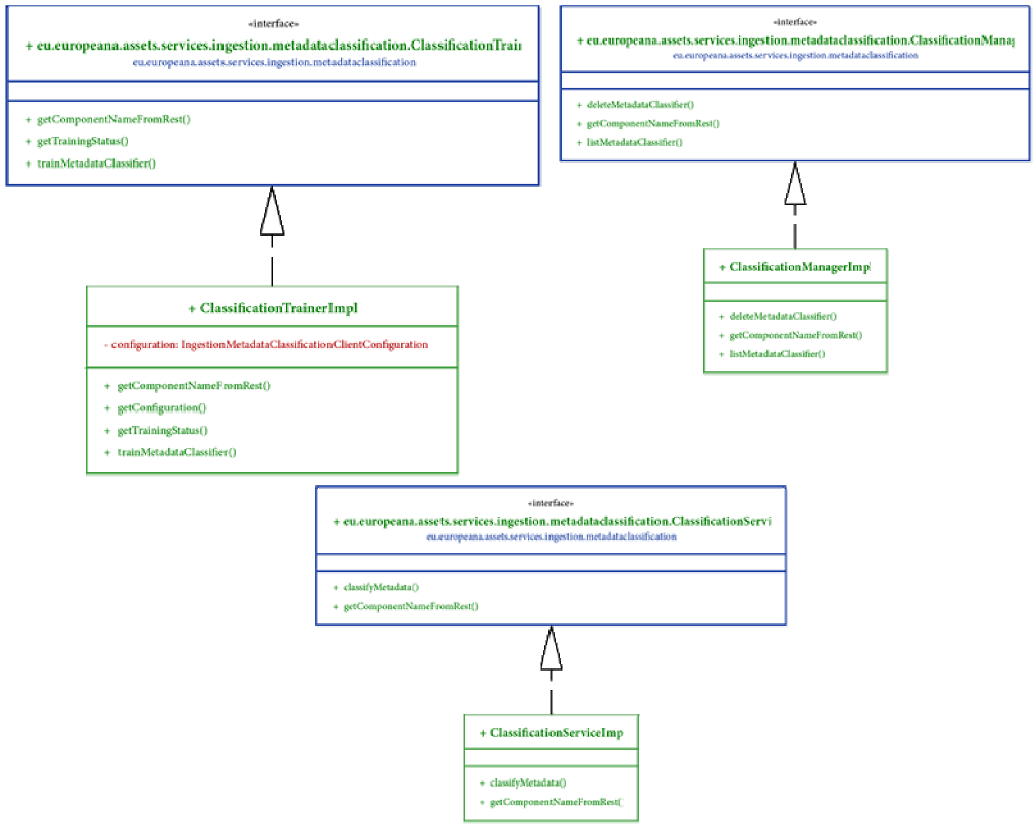

«interface»
+ eu.europeana.assets.services.ingestion.metadataclassification.ClassificationTrai
eu.europeana.assets.services.ingestion.metadataclassification
+ getComponentNameFromRest()
+ getTrainingStatus()
+ trainMetadataClassifier()
«interface»
+ eu.europeana.assets.services.ingestion.metadataclassification.ClassificationMana
eu.europeana.assets.services.ingestion.metadataclassification
+ deleteMetadataClassifier()
+ getComponentNameFromRest()
+ listMetadataClassifier()
+ ClassificationTrainerImpl
- configuration: IngestionMetadataClassificationClientConfiguration
+ getComponentNameFromRest()
+ getConfiguration()
+ getTrainingStatus()
+ trainMetadataClassifier()
+ ClassificationManagerImp
+ deleteMetadataClassifier()
+ getComponentNameFromRest()
+ listMetadataClassifier()
«interface»
+ eu.europeana.assets.services.ingestion.metadataclassification.ClassificationServi
eu.europeana.assets.services.ingestion.metadataclassification
+ classifyMetadata()
+ getComponentNameFromRest()
+ ClassificationServiceImp
+ classifyMetadata()
+ getComponentNameFromRest(

Figure 20 - Ingestion Metadata Classification Client Model

4.3.4 *Ingestion Workflow Models and Interfaces*

Concept Definitions

- **configuration time**: execution phase during which the plugins are configured. In OSGi this corresponds to the registration phase, during which validity checks are performed by the system.
- **processing time**: execution phase during which the MetaDataRecords are being processed

Workflows

A number of predefined workflows will be provided which cover the standard steps for processing and ingesting collections in Europeana platform. These workflows can be adapted and tweaked by technical staff for certain collections. The ingestion team needs to assign a workflow to the collection before it is going to be processed.

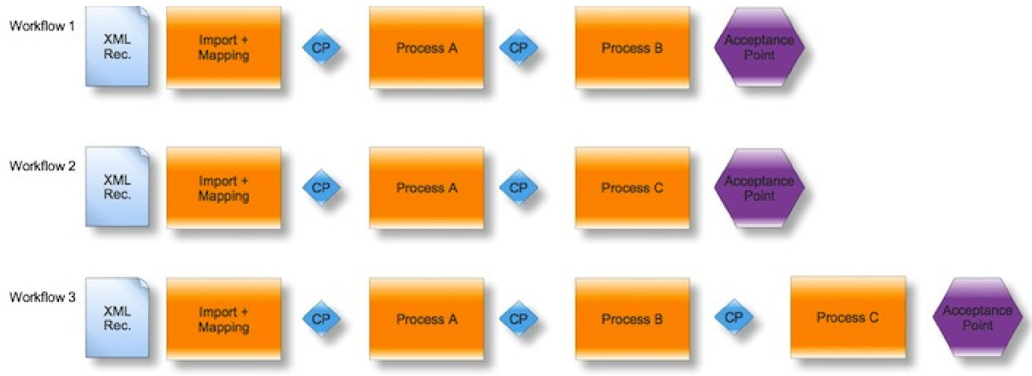


Figure 21 – Ingestion Workflows

Processing Model

Due to the necessity of optimal resource usage, each process part is asynchronous executed within a thread pool. A plugin must make explicit if it is not thread safe - in which case the framework ensures that no more than one thread at a time uses the plugin. To uncouple each processing block from each other a FIFO queue will be provided. The input queue will thereby be filled by the framework controller to ensure, that the framework is in control of all load-balancing issues

The ingestion workflow management is developed as a joint effort with Europeana and The European Library. It provides a framework for a scalable and robust execution of the ingestion of large quantities of meta-data records and allows specialized processing by using a plugin based mechanism.

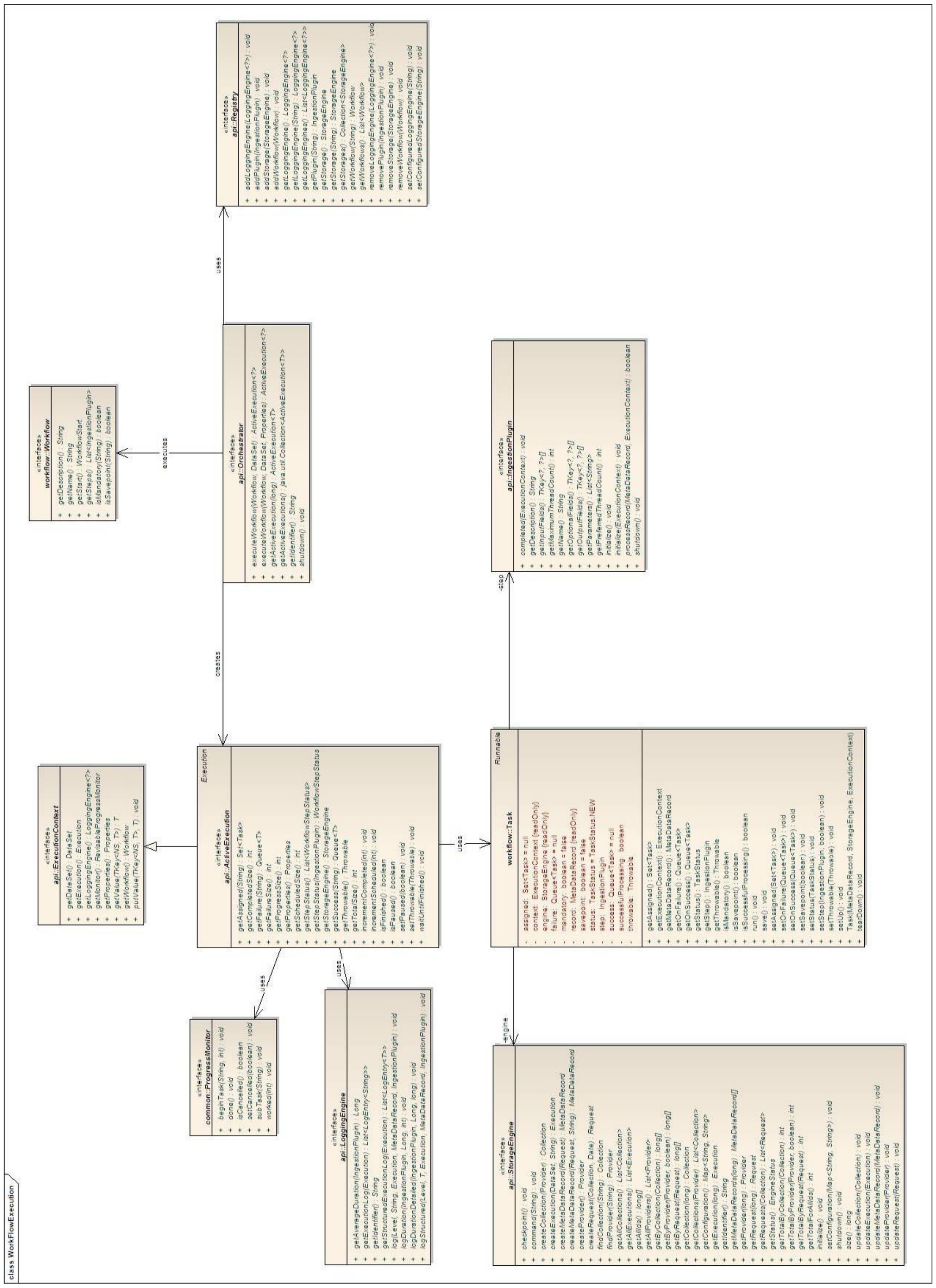

class WorkFlowExecution
«interface» workflow::Workflow
+ getDescription() : String
+ getName() : String
+ getStart() : WorkflowStart
+ getSteps() : List<IngestionPlugin>
+ isMandatory(String) : boolean
+ isSavepoint(String) : boolean
executes
«interface» api::Orchestrator
+ executeWorkflow(Workflow, DataSet) : ActiveExecution<?>
+ executeWorkflow(Workflow, DataSet, Properties) : ActiveExecution<?>
+ getActiveExecution(long) : ActiveExecution<T>
+ getActiveExecutions() : java.util.Collection<ActiveExecution<T>>
+ getIdentifier() : String
+ shutdown() : void
uses
«interface» api::Registry
+ addLoggingEngine(LoggingEngine<?>) : void
+ addPlugin(IngestionPlugin) : void
+ addStorage(StorageEngine) : void
+ addWorkflow(Workflow) : void
+ getLoggingEngine() : LoggingEngine<?>
+ getLoggingEngine(String) : LoggingEngine<?>
+ getLoggingEngines() : List<LoggingEngine<?>>
+ getPlugin(String) : IngestionPlugin
+ getStorage() : StorageEngine
+ getStorage(String) : StorageEngine
+ getStorages() : Collection<StorageEngine>
+ getWorkflow(String) : Workflow
+ getWorkflows() : List<Workflow>
+ removeLoggingEngine(LoggingEngine<?>) : void
+ removePlugin(IngestionPlugin) : void
+ removeStorage(StorageEngine) : void
+ removeWorkflow(Workflow) : void
+ setConfiguredLoggingEngine(String) : void
+ setConfiguredStorageEngine(String) : void
creates
«interface» api::ExecutionContext
+ getDataSet() : DataSet
+ getExecution() : Execution
+ getLoggingEngine() : LoggingEngine<?>
+ getMonitor() : RevisableProgressMonitor
+ getProperties() : Properties
+ getValue(TKey<NS, T>) : T
+ getWorkflow() : Workflow
+ putValue(TKey<NS, T>, T) : void
Execution
«interface» api::ActiveExecution
+ getAssigned(String) : Set<Task>
+ getCompletedSize() : int
+ getFailure(String) : Queue<T>
+ getFailureSize() : int
+ getProgressSize() : int
+ getProperties() : Properties
+ getScheduledSize() : int
+ getStepStatus() : List<WorkflowStepStatus>
+ getStepStatus(IngestionPlugin) : WorkflowStepStatus
+ getStorageEngine() : StorageEngine
+ getSuccess(String) : Queue<T>
+ getThrowable() : Throwable
+ getTotalSize() : int
+ incrementCompleted(int) : void
+ incrementScheduled(int) : void
+ isFinished() : boolean
+ isPaused() : boolean
+ setPaused(boolean) : void
+ setThrowable(Throwable) : void
+ waitUntilFinished() : void
uses
«interface» common::ProgressMonitor
+ beginTask(String, int) : void
+ done() : void
+ isCancelled() : boolean
+ setCancelled(boolean) : void
+ subTask(String) : void
+ worked(int) : void
uses
«interface» api::LoggingEngine
+ getAverageDuration(IngestionPlugin) : Long
+ getExecutionLog(Execution) : List<LogEntry<String>>
+ getIdentifier() : String
+ getStructuredExecutionLog(Execution) : List<LogEntry<T>>
+ log(Level, String, Execution, MetaDataRecord, IngestionPlugin) : void
+ logDuration(IngestionPlugin, Long, int) : void
+ logDurationDetailed(IngestionPlugin, Long, long) : void
+ logStructured(Level, T, Execution, MetaDataRecord, IngestionPlugin) : void
uses
Runnable
workflow::Task
- assigned: Set<Task> = null
- context: ExecutionContext {readOnly}
- engine: StorageEngine {readOnly}
- failure: Queue<Task> = null
- mandatory: boolean = false
- record: MetaDataRecord {readOnly}
- savepoint: boolean = false
- status: TaskStatus = TaskStatus.NEW
- step: IngestionPlugin
- success: Queue<Task> = null
- successfulProcessing: boolean
- throwable: Throwable
+ getAssigned() : Set<Task>
+ getExecutionContext() : ExecutionContext
+ getMetaDataRecord() : MetaDataRecord
+ getOnFailure() : Queue<Task>
+ getOnSuccess() : Queue<Task>
+ getStatus() : TaskStatus
+ getStep() : IngestionPlugin
+ getThrowable() : Throwable
+ isMandatory() : boolean
+ isSavepoint() : boolean
+ isSuccessfulProcessing() : boolean
+ run() : void
+ save() : void
+ setAssigned(Set<Task>) : void
+ setOnFailure(Queue<Task>) : void
+ setOnSuccess(Queue<Task>) : void
+ setSavepoint(boolean) : void
+ setStatus(TaskStatus) : void
+ setStep(IngestionPlugin, boolean) : void
+ setThrowable(Throwable) : void
+ setUp() : void
+ Task(MetaDataRecord, StorageEngine, ExecutionContext)
+ tearDown() : void
-step
«interface» api::IngestionPlugin
+ completed(ExecutionContext) : void
+ getDescription() : String
+ getInputFields() : TKey<?, ?>[]
+ getMaximumThreadCount() : int
+ getName() : String
+ getOptionalFields() : TKey<?, ?>[]
+ getOutputFields() : TKey<?, ?>[]
+ getParameters() : List<String>
+ getPreferredThreadCount() : int
+ initialize() : void
+ initialize(ExecutionContext) : void
+ processRecord(MetaDataRecord, ExecutionContext) : boolean
+ shutdown() : void
-engine
«interface» api::StorageEngine
+ checkpoint() : void
+ command(String) : void
+ createCollection(Provider) : Collection
+ createExecution(DataSet, String) : Execution
+ createMetaDataRecord(Request) : MetaDataRecord
+ createMetaDataRecord(Request, String) : MetaDataRecord
+ createProvider() : Provider
+ createRequest(Collection, Date) : Request
+ findCollection(String) : Collection
+ findProvider(String) : Provider
+ getAllCollections() : List<Collection>
+ getAllExecutions() : List<Execution>
+ getAllIds() : long[]
+ getAllProviders() : List<Provider>
+ getByCollection(Collection) : long[]
+ getByProvider(Provider, boolean) : long[]
+ getByRequest(Request) : long[]
+ getCollection(long) : Collection
+ getCollections(Provider) : List<Collection>
+ getConfiguration() : Map<String, String>
+ getExecution(long) : Execution
+ getIdentifier() : String
+ getMetaDataRecords(long) : MetaDataRecord[]
+ getProvider(long) : Provider
+ getRequest(long) : Request
+ getRequests(Collection) : List<Request>
+ getStatus() : EngineStatus
+ getTotalByCollection(Collection) : int
+ getTotalByProvider(Provider, boolean) : int
+ getTotalByRequest(Request) : int
+ getTotalForAllIds() : int
+ initialize() : void
+ setConfiguration(Map<String, String>) : void
+ shutdown() : void
+ size() : long
+ updateCollection(Collection) : void
+ updateExecution(Execution) : void
+ updateMetaDataRecord(MetaDataRecord) : void
+ updateProvider(Provider) : void
+ updateRequest(Request) : void

Figure 22- Workflow Execution Model

Service Name	Unified Ingestion Manager
Responsibility	1. definition of ingestion workflows; 2. workflow execution orchestration; 3. reporting
Provided Interfaces	1. Workflow, 2. MetaDataRecord, 3. IngestionPlugin, 4. SavePoint, 5. Execution, 6. Orchestrator
Dependencies	Apache Karaf OSGi implementation

Interface Name	Workflow
Key Concepts	Representation of a workflow, composed of multiple WorkflowSteps.
Operations	• String getName() - name of the workflow, should be reasonable meaningful • String getDescription() - description of this specific workflow (what does it perform, what should be the outcome, etc.) • WorkflowStart getStart() - defined start point of work flow • List<IngestionPlugin> getSteps() - plugins as steps in this workflow • boolean isSavepoint(String pluginName) - Is this a save point plugin? • boolean isMandatory(String pluginName) - Is this a mandatory plugin, so unsuccesful processing is a failure?

Interface Name	IngestionPlugin
Key Concepts	Definition of a plugin that processes meta-data records. Services that provide ingestion-time capabilities need to implement this interface. An ingestion plugin is a single processing step within a workflow
Operations	• String getName() - Get the class name of the plugin which is used to register the plugin with the registry. • String getDescription() -Get the description of the plugin which is provided to the operators when starting analyzing workflows. • TKey<?, ?> getInputFields() - Get the list of fields this plugin wants to operate on. This is used for information purposes, so that it can be validated if the records hold these data. • TKey<?, ?> getOptionalFields() - Get the list of fields this plugin would

	like to operate on or can get additional information for the working process. This is used for information purposes, so that it can be validated if the records hold these data. • TKey<?, ?> getOutputFields() - Get the list of output fields. @return a list of fields this plugin creates • void initialize() - Initialize the plugin when it is loaded in the OSGI container and attached to the uim registry. • void shutdown() - Shutdown the plugin when it is removed from the uim registry (due to OSGI shutdown or reinstallation etc. • List<String> getParameters() - List of configuration parameters this plugin can take from the execution context to be configured for a specific execution. • int getPreferredThreadCount() - A plugin is always executed within a thread pool, this parameter defines the preferred size of the pool. Plugins should know best, what's a good level of parallelism. • int getMaximumThreadCount() - Number of maximum threads. The plugin might specify here one (1) if it is not thread safe. • void initialize(ExecutionContext context) throws IngestionPluginFailedException - Initialization method for an execution context. The context holds the properties specific for this execution. • void completed(ExecutionContext context) throws IngestionPluginFailedException - Finalization method (tear down) for an execution. At the end of each execution this method is called to allow the plugin to clean up memory or external resources. • boolean processRecord(MetaDataRecord mdr, ExecutionContext context) throws IngestionPluginFailedException, CorruptedMetadataRecordException - Process a single meta data record within a given execution context. It returns true, if processing went well and false, if something failed.

Interface Name	ActiveExecution
Key Concepts	Type-safe representation of a meta-data record
Operations	• getId • addField • addQField • setField • setQField

Interface Name	Execution

Key Concepts	An Execution in a running state. It keeps track of the overall progress.
Operations	• StorageEngine getStorageEngine() • public void setPaused(boolean paused); • boolean isPaused(); • boolean isFinished() - test the execution if all tasks are done eather completly finished or failed. so if true: scheduled == finished + failed • void setThrowable(Throwable throwable); • Throwable getThrowable(); • Queue<T> getSuccess(String name); • Queue<T> getFailure(String name); • Set<Task> getAssigned(String name); • void incrementCompleted(int count); int getProgressSize(); - gives an estimate of tasks/records which are currently in the pipeline. Note that failed tasks are not counted. The system can not guarantee the number of records, due to the problem that some of the tasks might change their status during the time of counting. • int getCompletedSize(); - gives the number of tasks/records which are completly finished successful by all steps. • int getFailureSize() - gives the number of tasks/records which have failed on the way through the workflow no matter where. • int getScheduledSize() - gives the number of tasks/records which have been scheduled to be processed in the first place. So scheduled = progress + finished + failure. • int getTotalSize() - gives the number of records which this execution will need to deal with. If not possible to estimate Integer.MAX_VALUE is given. • List<WorkflowStepStatus> getStepStatus(); • WorkflowStepStatus getStepStatus(IngestionPlugin step); • public Properties getProperties(); • void waitUntilFinished(); • void incrementScheduled(int work);

Interface Name	Orchestrator
Key Concepts	Workflow execution orchestration
Operations	• public String getIdentifier(); • ActiveExecution<?> executeWorkflow(Workflow w, DataSet dataset); • ActiveExecution<?> executeWorkflow(Workflow w, DataSet dataset,

	Properties properties); • <T> ActiveExecution<T> getActiveExecution(long id); • <T> java.util.Collection<ActiveExecution<T>> getActiveExecutions(); • void shutdown();

4.3.5 Post-Ingestion Processing

Service Name	Post-Ingestion Processing
Responsibility	1. multimedia content harvesting, 2. multimedia content indexing
Provided Interfaces	1. MultimediaContentHarvesting, 2. MultimediaContentIndexing
Dependencies	Assets Common, Europeana Core, Text Indexing, Image Indexing, Audio Indexing, 3D Indexing

Interface Name	MultimediaContentHarvesting
Key Concepts	MultimediaContent MultimediaContentUrl ObjectMetadata (FullDoc/ESE/EDM)
Operations	• downloadMultimediaContent

Interface Name	MultimediaIndexing
Key Concepts	Multimedia Index, MultimediaContent
Operations	• startIndexing, • saveIndexedRecord, • getIndexQueueSize
Extends	• Core-Indexing

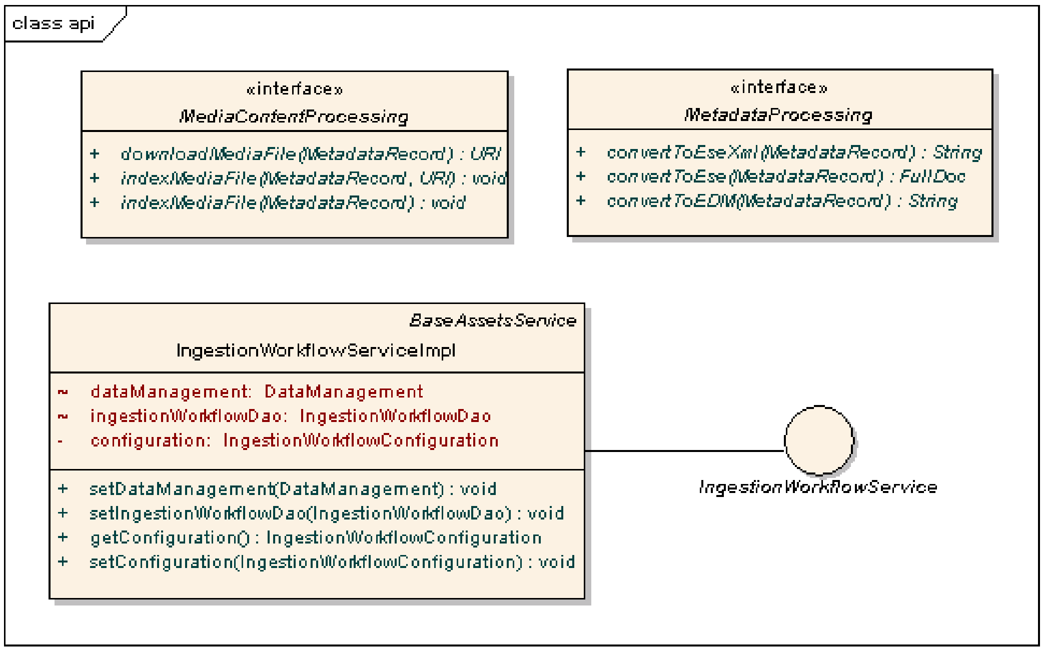

class api
«interface»
MediaContentProcessing
+ downloadMediaFile(MetadataRecord) : URI
+ indexMediaFile(MetadataRecord, URI) : void
+ indexMediaFile(MetadataRecord) : void
«interface»
MetadataProcessing
+ convertToEseXml(MetadataRecord) : String
+ convertToEse(MetadataRecord) : FullDoc
+ convertToEDM(MetadataRecord) : String
BaseAssetsService
IngestionWorkflowServiceImpl
~ dataManagement: DataManagement
~ ingestionWorkflowDao: IngestionWorkflowDao
- configuration: IngestionWorkflowConfiguration
+ setDataManagement(DataManagement) : void
+ setIngestionWorkflowDao(IngestionWorkflowDao) : void
+ getConfiguration() : IngestionWorkflowConfiguration
+ setConfiguration(IngestionWorkflowConfiguration) : void
IngestionWorkflowService

Figure 23 – Ingestion Workflow API model

References

1. ASSETS D2.0.1 "Requirements Specification" – Internal Document
2. ASSETS MS12 "System Architecture" – Internal Document
3. ASSETS MS27 "Digital Preservation Service Design"– Internal Document
4. Roy Thomas Fielding "Architectural Styles and the Design of Network-based Software Architectures", 2000 – available at http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
5. H. Belhaj-Frej, P. Rigaux, N. Spyratos "User notification in taxonomy based digital libraries", SIGDOC '06 Proceedings of the 24th annual ACM international conference on Design of communication. Available at http://portal.acm.org/citation.cfm?id=1166366
6. Europeana Aggregator's Handbook - available at http://www.version1.europeana.eu/c/document_library/get_file?uuid=94bcddbf-3625-4e6d-8135-c7375d6bbc62&groupId=10602
7. ASSETS D2.0.2 "Interface Specifications and System Design" – Internal Document
8. ASSETS D2.0.4 "The ASSETS APIs"

Appendix 1 - Enrichment Services Training Data Format

This page2 provides information on how CPs have to format their data in order to submit training data to the enrichment services of WP2.1:

- Metadata cleaning
- Knowledge extraction
- Metatada classification

The training data file format is XML. For each task an XML schema file, i.e,. an XSD file, is provided. CPs should use the schema for each task to produce their training data files (one file for each task).

In order to simplify the process we also provide an XML example file for each task and a document with guidelines and comments. CPs could follow the guidelines and use the examples as a starting point to produce their training data files.

The ASSETS wiki folder "Enrichment Services Training Data Guidelines" contains both the guidelines for the content providers and the xsd/xml files that are to be used in preparing the training sets. More in details, that folder contains the following files:

- cleaningSchema.xsd.txt: XML Schema Definition for providing training sets to task T2.1.1 "Metadata Cleaning". The file extension is txt because the wiki does not allow the upload of xsd files. The file should be renamed by removing the extension .txt.
- cleaningExample.xml: example of a well-formed training set for task T2.1.1. The content providers may modify this file for providing their training data.
- extractionSchema.xsd.txt: XML Schema Definition for providing training sets to task T2.1.2 "Knowledge Extraction". The file extension is txt because the wiki does not allow the upload of xsd files. The file should be renamed by removing the extension .txt.
- extractionExample.xml: example of a well-formed training set for task T2.1.2. The content providers may modify this file for providing their training data.
- classificationSchema.xsd.txt: XML Schema Definition for providing training sets to task T2.1.3 "Metadata Classification". The file extension is txt because the wiki does not allow the upload of xsd files. The file should be renamed by removing the extension .txt.
- classificationExample.xml: example of a well-formed training set for task T2.1.3. The content providers may modify this file for providing their training data.
- T2.1_TrainingGuidelines.pdf: pdf document presenting and discussing the xml/xsd files above.
- T2.1_Training.tgz": tgz pack containing all of the previous files (with the correct file extension).

All of the previous files are available on the ASSETS wiki at the following URL: http://www.assets4europeana.eu/web/portal/documents?p_p_id=20&folderId=35495

2 http://www.assets4europeana.eu/web/portal/wiki/-/wiki/Main/Enrichment%20Services%20Training%20Data%20Format