OmNom

DM2E's Ingestion Platform

CONCEPT FOR THE RDFIZATION FRAMEWORK
DEVELOPED BY WORK PACKAGE 2 OF DM2E

Revision	Date	Authors
1	30.09.2012	Konstantin Baierer

Contents

1 Introduction

The next step to achieve the goals set forth by the DOW must be the streamlining of the various steps of data transformation into one platform. As of month 6 of DM2E, the process is completely manual, i.e. it consists of many steps with different tools, orchestrated by the user. This requires extensive knowledge about data formats and transformation tools from the user and forces her to keep track of the different states of data, so she is able to feed the correct dataset to the next tool in the manual processing pipeline. An automatable solution, as defined by Task 2.1 in the DM2E Description of Work would relieve her from this error-prone tedium.

This document is an internal report based on practical research and the ongoing implementation of a prototype. This document is *not* a scientific article ready for publishing in a Computer Science journal. This means that there are strong over-simplifications to be found throughout the text and the terminology might be idiosyncratic at times. While the author is willing to adjust the wording and correct half-truths and errors in later revisions, please keep in mind that the purpose of this report is to serve as the foundation for a specification and subsequent implementation and as a proposal as to how, when and by whom the framework should be realized.

Many aspects of this work, esp. on practical aspects of tools like MINT and R2R, can also be found in the DM2E Deliverable 2.1.

The report will first give a high-level overview of what this platform will have to be able to do. Afterwards, an architecture is defined in terms of backend, components, asynchronous functionality and user interface requirements. Actual instances of those components and libraries and tools to use them are presented in the following section. Finally, a prototype developed by the author over the course of three weeks in Perl of how this system can actually be coded is described.

The final section concerns itself with the project-related, hands-on aspects of how OmNom relates to the Description of Work, how the development is organized within WP2 and a list of dates when fixed milestones will have to be reached.

As for the name: OmNomNom is an onomatopoeism widely used in Net Culture that is meant to resemble chewing sounds. As the author sees it, this platform is for general data munging, so the name seems appropriate, plus it's short and catchy and easy to type. As with all other aspects of the system, the author is more than happy to discuss alternative names if this one seems inappropriate.

2 High Level overview of RDFization lifecycle

Before the concrete requirements for such a unified platform, One-Stop-Shop for transforming noisy legacy data to rich and deeply structured DM2E-flavored RDF - codename DM2E OmNom, can be described, a high level overview of the different stages of the processing pipeline is presented in this section.

There have to be at least six different stages:

- Preprocessing
- Schema Mapping
- Transformation
- Correction
- Linking
- Publishing

These will be explained in the following sections. One important distinction should be made right now: Mapping and Transformation are not the same thing. Mapping refers to the high level task of translating one data format ("input format") to another data format ("output format". This has to be done only once for every input/output format pair. The result of the mapping is also referred to as "mapping file" or mapping for short. The process of applying a mapping to input data in an input format produces output data in an output format - this process is called transformation and has to be executed for every set of input data.

2.1 Preprocessing

OmNom has to be able to handle at least three basic data formats: Hierarchical (serialized as XML), Relational (serialized as a SQL database) and Graph-Based (serialized in any standard RDF syntax). There is virtually no data in the domain DM2E is working on, that doesn't adhere to one of these structures. There are however cases, where the serialization differs, for example MAB-2 or MARC, which are flat key-value text formats or CSV, which is a tabular format. Those need to be transformed to an appropriate input format before the actual transformation can begin. Schema Mapping

OmNom needs information on the structure of the input data. This includes

- Set of global elements and properties
- Rules of containment and linking of elements
- Data Types of elements and properties
- Value Ranges of elements and properties (including class information for values representing resources rather than literals)

Based on these structural attributes of the input format, a mapping can be constructed that allows automatic transformation of any input data in this format to EDM+.

There are three levels of abstraction which could - in theory - be used for creating mappings:

- Serialization (one mapping each for XML, SQL and RDF)
- Data format (one mapping for MARC-XML, one for TEI, one for ExLibris Aleph's Oracle database schema and one for every relevant RDF schema/ontology)
- Data Provider (one mapping for BBAW's TEI flavor, one for UIB's TEI flavor . . .)

The serialization level is obviously to abstract - serialization is just the manifestation of a data structure and therefore enforces little restriction on structure and semantics of the data encoded in it.

Given that data formats are standardized for this very purpose - interoperability, it should be sufficient to create one mapping for every data format that should be applicable to every set of input data in this format. This has many advantages: First of all, mapping becomes work that has to be done only once and be re-used not only by DM2E's current data providers but future data providers which provide data in the same format. Secondly, the mapping can be done in a very formal way by strictly adhering to the specifications of the data format.

The reality begs to differ however: Every producer of data in a standardized data format runs into cases where the specifications are either to restrictive or too unspecific for the modeling puposes of the producer. The former case leads to "hacks", i.e. ways to circumvent this restriction by abusing elements for unintended purposes, the latter case leads to de-facto standards ("flavors") that might be consistent within the producer's organization or even larger groups, but requires "insider knowledge" not found in the specifications and yields unpredictable results for other flavors of this data format.

Prominent examples for data formats that are standardized in excruciating detail, but are used in practice in completely different ways are TEI and MARC. TEI allows both literals and elements for many elements and enforces little schematic limitations on properties (e.g. how to specify the <idno> element: Is it's @type property a string or a URL? Is the value a string or a URI?), while the line-based nature of MARC forces data producers to repeat the same elements with the same properties and opaquely store information in the order of those elements (e.g. the first 856a field is the URL of the thumbnail, the second 856a field is a link to Wikipedia) or in the value of the element itself (e.g. If 856a begins with "DNB:", the string after "DNB:" is the URL to this publication's landing page at the German National Library portal).

For these reasons there will have to be provider-specific adaptions of format-specific mappings. These can be made at three stages of processing:

- Preprocessing: The provider-specific adaptations are transformed to the flavor of the input format the format-specific mapping is aware of

- Transformation: The mapping is produced specifically for the provider
- Correction: Provider-specific format adaptations are ignored during the transformation process and only corrected afterwards (see section "Correction")

All three approaches have pros and cons. (1) is the best solution for a clean workflow, as after it is done, no more adaptions to the transformation platform have to be done. It will not work in many cases since there are no specifications for many of these adaptions in the data format specs, which led to the data format adaptions by the providers in the first place. (2) is the most cumbersome, because it has to be done for every provider, which hinders re-use and testing, but the mapping will be 100approach (3) means that the validity of data produced by the transformation cannot be guaranteed anymore and requires that provider's strictly adhere to their own flavor of the input format (i.e. every piece of input data must yield the same errors to be picked up in the correction step), but allows, again, to use one and only one mapping for every data file.

Leaving aside the difficulty of creating a correct mapping for the input data formats, the task of mapping data formats is not part of OmNom, since this is best left to tools that are created for this very purpose. OmNom will however integrate those tools in such a manner, that the user will notice as little disruption from her workflow as possible.

While tools like MINT, Culturegraph Metamorph, Cliopatria XMLRDF or Freemarker are often more than just a transformation engine with a thin API, but, as is the case with MINT, a fully-featured IDE for the complete mapping/transformation workflow. To make OmNom more versatile they will serve in two distinct functions: As an editor component for writing the mappings and as transformation engines. This has the advantage that one can use the editing features of a tool, while being free to choose a transformation engine of their liking.

2.2 Schema Mapping

Data transformation is an iterative process:

Create or improve mapping Transform data using this mapping Repeat until results are good enough

To make this development cycle easier, OmNom should offer capabilities to automate the transformation and result checking steps as much as possible and offer an integrated way to both edit and validate the mapping.

MINT includes all these features in a user-friendly interface – there is no User Interface for either D2R or R2R however.

2.3 Transformation

The central functionality of OmNom will be the actual transformation from whatever input format the data providers use to EDM+, an extension of the

Europeana Data Model (EDM). The choice of transformation engine coincides with the type of mapping. As there are three types of serializations OmNom will support, there will be at least three types of transformation engines (see table 1).

Table 1: Mapping tools and transformation engines for OmNom's formats Serialization Mapping Tool Transformation Engine Used by XML MINT XSLT Processor UIB, BBAW, NLI RDF Manual/Custom R2R MPIWG SQL database Manual D2R SBB

For XML-based data formats, the MINT tool will be integrated into OmNom. MINT is an integrated web-based platform for mapping XML formats to other XML formats, creating XSLT stylesheets in the process, which can be either used within the MINT platform or exported for use in other environments. For the actual transformation process, only this XSLT stylesheet is necessary and has to be made available to OmNom. A tight integration of the authentication and data management features of MINT and OmNom will be targeted.

For transformations from RDF or from RDB the input data is transformed using the tools R2R and D2R. Both are sophisticated tools that require a configuration file in RDF TURTLE syntax as the mapping. The transformation is then executed via a command line script or by using the Java API directly. An editor component that helps with writing the rules doesn't exist and it should arguably be created, with the minimum features being: well-formed-ness check and schema validation.

2.4 Correction

This step shouldn't be necessary in theory, but is in practice and as such a crutch to be dropped in either before, during or after the data transformation stage to "massage" invalid or insufficiently structured data into something that the next step within the processing pipeline can work with.

Correction as a task of it's own should be avoided whenever possible. Whenever recurring patterns of suboptimal data creation emerge, their causes should be tracked ups and fixed as early in the processing stage as possible, which means that the data provider's should ensure that their input data adheres to the specifications on which the mappings are based - and correct the data upstream before they submit it to OmNom.

In many cases, the data isn't really invalid but yields results that make the subsequent steps much harder. The results of the Linking heuristics can be much improved by ensuring that literal values follow a uniform syntax. For this purpose, the correction might actually remove data or store contextual objects as literal strings – which is contrary to what DM2E wants to achieve in general but might actually improve Linking results in certain cases.

A good example why correction is necessary is the syntax of person names, which varies widely across data providers and frequently even within a single provider's data. This might lead to first names and last names being swapped or additional information being interpreted as part of the name. Solving problems like these require either very sophisticated and specially trained pattern recognition algorithms or specialized heuristics for a small, but well-known set of patterns.

2.5 Linking

owl:sameAs and whatnot

Linking means identifying URLs for resources within the input data

2.6 Publishing

There are two levels of publishing that need to be distinguished: Publication of the more or less raw EDM RDF data (i.e. which is mostly metadata) and publication of the content in such a way that tools, like the Pundit annotation framework developed in WP3 can easily interact with the full texts of the manuscript.

2.6.1 Metadata Publishing

- Dump into DM2E triplestore
- Dump into user specified triplestore
- Publish via SPARQL/Update
- Publish via general HTTP mechanism (POST or PUT file somewhere)
- Upload as RDF dump to some FTP
- Download RDF dump

2.6.2 Full Text Publishing

DM2E aims at handling metadata and manuscript contents within the same framework. In accordance with the general shift from document-based data to networks of inter-related chunks of information prevalent in the Semantic Web community, the traditional dichotomy of content and metadata, i.e. distinct and self-contained, possibly non-digital units of information and metadata records describing them is being phased out in lieu of an information graph that doesn't distinguish between data and metadata anymore.

Consequently, OmNom should be able to handle not only the metadata of manuscripts, but the contents, structure and micro-structure of texts, including state-of-the-art NLP analyses of said contents, such as named entity recognition and resolvement of anaphora.

The transformation of full texts to EDM+ has to be bijective and reversible, meaning that the atoms that make up the text must be uniquely identifiable using a URI, and the hierarchy and sequence of the original text has to be reconstructible from the EDM+ graph structure. The specifics of how the chunks of information can be retrieved using this URI are subject to debate – either the providers set up a web service that uses a custom look-up database which renders the contents in a way accessible for the tools developed in WP3 or DM2E takes a RESTful approach and turns the URIs into HTTP-GET-resolvable URLs, to be handled directly using the Semantic Web stack deployed (i.e. the Linked Data frontend to the triplestore). This might however prove a performance burden since most triplestores aren't fit to handle millions of very long literals.

2.7 Round-tripping Data

Should the data transformation to EDM+ be bijective, i.e. should the the input data be reconstructable from the EDM+ version without information loss? DM2E has been taking a strong stance against round-tripping capabilities of the system, since the main goal of the project is to facilitate the dissemination of metadata and content to enable innovative ways of working with manuscript data, not to develop a data model that is capable of handling every last intricacy of the various storage solutions used by the data providers. While this is a wise and understandable constraint, the ingestion infrastructure should not structurally prohibit possible future development towards round-tripping. At the current project stage, the vast majority of input data to be ingested is serialized in plain XML, the scenario where a data provider wants to transform data in a legacy RDF format to EDM+ isn't far-fetched however. And from there, it is only a small conceptual step to round-tripping capabilities. If a user of the platform is willing to walk that extra mile and implement not only a mapping from their legacy data format to EDM+ but also a mapping to reverse the process the system should reward her by offering to handle this reverse mapping as well.

The baseline on the flexibility of the system is that input shouldn't be restricted to XML and output shouldn't be restricted to EDM+.

3 OmNom architecture

3.1 Backend

The backend can be implemented using any technology, the only restrictions are that the backend should be as performant as possible and support a developer-friendly API. As the backend is mostly "glue code" that combines various existing data loading and transformation tools and libraries, the backend components should be modelled with abstract interfaces, so that the implementations that connect to external tools can be exchanged without breaking the functionality (i.e. using a different job server or a different RDF stack) and the data flow should be flexible enough to accomodate data handling components beyond those listed in the specifications at this time.

3.1.1 Data Storage

Data ingestion is obviously a data-intensive process which requires the storage layer to be able to handle both very large binary blobs of data (e.g. data dumps ready to be split up and transformed) and many small chunks of character data (e.g. configuration files, temporary files during transformation, log files,). The "many small files"-aspect is probably best handled by a DVCS repository and a reasonable policy with the data providers on how changes are pushed/pulled into the repo. Using a DVCS for text file management solves many problems at once: Files are versioned, data integrity is guaranteed and data can be accessed and shared quite easily as there are exhaustive APIs and UIs to DVCS. For large files, DVCS aren't practical however. The easiest possibility is to prevent the users from storing actual data on the server, requiring them to give a remote location at ingestion time from where the backend pulls the data and optionally caches it locally for a limited period of time. The other extreme would be a fully featured data store backed by a distributed file system such as HDFS or GridFS. A good compromise would be to outsource the details of storage and backups to a Cloud Service like Amazon S3 and concentrate on Authentication/Authorization aspects and a solid API.

3.1.2 Data Model

A data type is a generic way of organizing information (Examples: XML, RDF, CSV, MAB2, MARC21), backed by an implementation in software that has functionality to access and store data in this structure (Examples: LibXML, Jena, RDF::Trine, Syck, LibXSLT, Saxon) and handle serialization and deserialization to/from textual representations of this format (Examples: RDF in TURTLE and RDF/XML or CSV with "comma" or "tab" as the separating character). Data formats are data types with at least shallow semantics, that is a schema against which incarnations of data in a data format can be validated. Data formats also can be defined by inheritance from other formats, allowing specialization. This way of distinguishing is fuzzy, as it makes mono-hierarchical classification of certain formats impossible (e.g. MARC-XML could be seen as a serialization of MARC21 and as an XML serialization; XSLT stylesheets are declared in XML), but it makes sense in OmNom's functionality as "glue code": The main reason for even trying to distinguish between different levels of abstraction - data type, data type implementation, data format - is to make it easier for the consumer of the platform to define the data flow for her needs as easily as possible.

If a user want's to inject snippets of XML into a text file, she might not even want to parse it, since there is no need to access individual nodes or run XPath queries on it - so make it easy for her to store the XML as text. Another user may need the complex features of XSLT 2.0, so she should be able to make it clear to the system that Saxon is the only acceptable XSLT implementation to be used for this specific step. Yet another user may want to handle TEI files, which is an XML-based format with many XSD schemas for different application profiles. This user, too, should be able to declare straightforward that the format is TEI and, no matter the implementation, should be treated and validated as such.

3.1.3 Data Flow Management

From a user perspective and from the perspective of a developer building on top of the DM2E ingestion platform, the process is supposed to be a black box for virtually all cases. The consumer sends the instructions for setting up the processing pipeline and the means to get the input data to the backend and the backend sets up the stage according to the instructions, runs the components - feeding output and additional data to components "downstream" until there is nothing left to do, then notifies the consumer of the results. There is a variety of functionality contained within this short paragraph and the complexity is augmented further by the multitude of data types, formats and flavors the backend has to support.

Configuration

The instructions, the configuration of the pipeline, must contain all the information that is necessary to set up the chain of events. Apart from metadata that is useful for fine-tuning the overall system, debugging and logging, there has to be a list of components to run in sequence[1]and their individual configuration. To make the system versatile, this configuration should be data structure universally representable in all major programming languages, ideally a JavaScript object that can be handled as a HashRef in Perl, a dictionnary in Python, a HashMap in Java etc. The benefit of keeping the configuration so simple is that it can easily serialized and deserialized and as such, already manifests an API for the data flow component.

Context, Buckets, BucketItems

Those instructions also define how the context on which the components are going to work is to be set up. In particular, this means that every component has to specify with what kind of data serialization and implemtentaion backends it is going to work. The context component then creates type- or implementation-specific buckets for the plugins to store data in. The *bucket and bucket item* metaphor offers several advantages over leaving the low-level data handling up to the individual components. For starters, a lot of behaviour can be stowed away in roles, interface or base classes and won't need to be re-implemented by components. Secondly, it offers a clean interface for different components to access shared data, *while keeping the data in memory*. This is especially important for large documents that would have to be serialized, stored, retrieved and de-serialized for data sharing otherwise. Since bucket items are named, the buckets can be accessed as a lookup table and as a stack. As such, cómponents can work like traditional UNIX pipeline where only the item *on top* is used as input and combine data items in arbitrarily complex patterns by accessing them by name. Another use-case is for caching of data retrieved from remote sites. Finally, encapsulating the actual data objects in a BucketItem object allows tracking the provenance of data during the pipeline lifetime: Which component created the data, which component manipulated it, and when, and who initiated the run? This makes evaluation of the platform usage easier and might later on even be stored as provenance information in the DM2E triplestore, as soon as the EDM+ versioning and provenance modelling is completed.

3.2 Data Handling Components

The data handling components are the core of the system and the least tightly integrated parts at the same time. While they are classified into a number of different classes, this distinction is largely arbitrary and serves more to make to offer specific functionality and helper methods for certain types of plugins by means of inheritance and role consumption, as well as for namespacing the components for the consumer of the API.

All components must implement a `work` method that takes the context object as an argument. They can specify any number of data types they are going to work on (signalling to the context that the respective buckets have to be set up). At construction time, attributes can bes set that define the behavior of this instance of the component:

`source` to load data from into one of the buckets

[1]and possibly in parallel, see 3.3.1.

`target` to publish data to
`pipeline` for nested contexts

Depending on the programming languae used for the actual implementation of the system, it might be worth considering if the interface of the component can be declared in such a manner as to make it possible to instantiate the components in the best suited programming language. The baseline should be that a component must be instantiable from a simple, easily serializable data structure such as JSON. However this still requires the framework to implement RPC interfaces for different programming languages and a protocol to mange those instances. It should however be considered if a more versatile multi-language service framework could be re-used. A good choice is Apache Thrift, a »software framework for scalable cross-language services development«[2], which has bindings to all major scripting languages (which are easy to use), as well as C++ (which is fast) and Java (which is the enterprise standard for most non-performance-critical applications and the langugage in which most of the tools listed in 4 are implemented).

3.2.1 Loader

Loader components take external data and parse them into an in-memory model that other plugins can work with. One thing to note here is that *in-memory* doesn't necessarily mean that all of the actual data is loaded into memory. Many backends work on iterators/streaming mode rather than raw data, which means that they fetch data from a filehandle, socket or similar on demand. Another way to prevent having to load all data in to memory at this stage are *lazy loading*, where the data is retrieved and parsed only when the respective bucket item is accessed by other components.

The `source` property of the Loader component could be either a HTTP-resolvable URL, a file path relative to the data backend (e.g. a path within a Git repository or an ID within a distributed file system) or in specific cases the raw data itself.

It is important that the loaders only store an access object in the context that allows the other components to access the data either by a simplified API for common actions (e.g. `toString`) or by the implementation's native methods (e.g. RDF::Trine's `serialize_model_to_file()`). The data object should be the native unit of storage for the data structure at hand, e.g.:

XML Root element of DOM tree
RDF Model (*Graph* of graphs of triples)
CSV Row iterator of cell iterators
MAB MAB2 record
MARC MARC record

[2]`http://thrift.apache.org/`.

Whereever possible the actual data shouldn't be loaded into memory until absolutely necessary and even then only in a streaming fashion.

3.2.2 Transformer

Transformers are components that work on input items and create or modify output items. In most cases they need detailled configuration, e.g. an XSLT stylesheet loaded earlier by a loader plugin. In other cases the transformation is straightforward, e.g. transforming a MARC21 field bases serialization to MARC-XML.

3.2.3 Publisher

Publisher components are responsible for storing data contained in the context to external data data storage resource. In the most simple case, say a HTTP `PUT` request, the plugin would take the `output` item from the `TEXT` bucket and send it to the URL defined in the `target` attribute.

The most important Publisher components to be implemented seem to be for Triplestores, SSH and HTTP requests.

A Triplestore publishing component would export the `output` model in the `RDF` bucket to the triplestore (or the SPARQL/Update serialized data in the `TEXT` item). A SSH publishing component can store data as files on a remote system, given that the keys have been set up accordingly - this is probably the easiest way to debug an ingestion run for human users. HTTP requests are versatile in that they can be used to access any RESTful API to store data to - which is probably the easiest way for external applications to interact with the results.

However, as the data target for DM2E's ingestion is still ill-defined at this stage of the project, the semantics of the Publisher plugin aren't clear yet either.

3.2.4 Serializer

Serializers inherit from Transformer which the constraint that the output bucket should be of type `TEXT`. This is a convenience class that makes it clear from the namespace already that the output produced by this transformer is plain text which can be printed to a filehandle, sent over HTTP or fed to a deserializer without having to worry about the underlying implementation of the `source` bucket.

3.2.5 Validator

Validator components are very similar to Transformaer components, though they don't necessarily have output. They use some validation information, e.g. an XSD schema or Schematron rule file and check the input item against this information. If the validation passes, the component is essentially a no-op, apart from maybe providing log messages. If the validation fails, the workflow should fail noisily and with log messages specifiying the exact validation errors to help the user create valid EDM+ (or whatever the backend validates against).

3.2.6 Splitter

Splitter components are a subclass of the Loader component, that has a `pipeline` attribute in addition to the `source` attribute. There purpose is to split up multi-record source data into the indidual records and run a pipeline on those items. The way this should be implemented is that the nested pipeline should start with a Loader plugin with unspecified `source` attribute, which will be sequentially set to the individual record.

There are various formats which can contain many records within a single data source, e.g. OAI-PMH responses, MAB2 batch files, MARC21 batch files. Some providers will want to provide their input data compressed as ZIP archives as well for batch processing.

To prevent performance degradation for large input data files, the underlying implementations should support a streaming interface, so that only those records currently processed have to be kept in memory.

Splitter plugins should support a Job interface so many records can be processed in parallel.

3.2.7 Linker

Linking is short for Link Discovery and means the (semi-)automatic discovery of relationships between a source dataset (controlled by he user) and one or more target datasets (somewhere in the *Linked Data cloud*

3.3 Job Server and Message Queue

The ingestion platform should be designed in such a way that functionality is contained in loosely coupled components that *do only one thing and one thing right*. This means that individual components shouldn't need to know anything about the context they're run in, apart from a thin API to the data storage layer (buckets and bucket items) and a message channel for logging and message exchange.

Another important design decision is that the system should *degrade gracefully*: If a component fails, this mustn't necessary mean that the whole ingestion run has failed - but if it does, it should do so as verbose and detailled as possible to make debugging easier.

Finally, OmNom is a *multi-user platform* with every user potentially wanting to ingest many records into the system at once. Were the system to be a single-process server, a single stalled ingestion would block the whole system for all users. If a new process is started for every user or – even worse – every ingestion, performance would degrade fast, possibly crashing the system when it runs out of ressources and opening a wide-open security hole.

To address all those issues, a job management component, as well as a messaging system are required.

3.3.1 Job Server

A job server is software that starts instances of the ingestion backend as necessary, limiting the number of processes globally, per-user or by some arbitrary convention (e.g. to prevent *hammering* external servers used by the Linking component). Every run of the backend is to be encapsulated into a job object that contains attributes for

- the state of the procss (`STARTED`, `WORKING`, `PAUSED`, `WAITING`, `STALLED`, `FAILED`, `COMPLETE`),
- the user who started the job,
- timestamps of state changes
- the initial configuration of the ingestion run
- logging messages

The job objects should be easy to persist in a database. The job server must have an API to let external applications and users introspect jobs and change their state.

Traditional job servers like Gearman can run jobs in three modes:[3]Blocking (i.e. jobs aren't parallelized), background (i.e all jobs are run in parallel) and taskset mode. The latter means that multiple jobs are run in parallel but the parent process waits until all of them are finished. This mechanism might be leveraged to not only parallelize complete ingestion runs but individual components. For example an ingestion configuration might want to load remote data both from an OAI-PMH endpoint, a SPARQL endpoint and a lookup XML document. All subsequent components require those data items, but the three Loader components are independent of each other so it might speed things up if they wouldn't need to wait for each other. This is a low priority feature however since it adds a level of asynchronicity that could possibly make the system hard to debug.

The job interface must however not be integrated directly into the context and component objects. In other words: The basic implementation of the workflow should be blocking but designed in such a way, that a job interface can re-implement the necessary parts for a non-blocking run.

3.3.2 Message Queue

Unless the components and the context they work on are implemented within the same environment – which would break the principle of loosely coupling of

[3]Remember: Not a scientific article :-)

components – there needs to be communication between components and context via some sort of protocol to signal state changes, log progress and possibly demand feedback (e.g. for user interaction). Again, such a protocol could be implemented from scratch on top of TCP or HTTP, which would not be a smart move. The messaging system has to be fast, i.e. with as little protocol overhead as possible, versatile enough to handle at least logging, component-context-communication and inter-component-communication and flexible enough to be applicable to future use-cases as well as changes of the underlying protocol.

A widely used possibility that should be evaluated is Apache RabbitMQ, a Message Queue that can be adopted to use several usage patterns, is highly performant and easy to intgrate.

It might even be worth considering to use the message queue as the protocol layer of the job server, i.e. for sending configurations of components over the network.

3.4 Frontend

Frontend development is either the easiest thing in the world (the hard work is done by the backend anyway, right?) or the toughest part of every software project (Usability, eh? Accessibility, huh? Responsiveness? Bling?) – depending on the perspective. The complete task 2.5 of DM2E's DOW is dedicated to the development of a clean user interface to the whole platform. While the details of how exactly it should be implemented cannot be clear until the backend is running smoothly, some views must be implemented and we can take an educated guess as to what technologies will be used for the UI and how the user will interact with it.

3.4.1 Technologies to use

The user interface is bound to be web-based. A solution that builds on OS-native applications just doesn't make sense for what is basically a configuration and job management tool.

Moreover, as the API of the backend is supposed to be HTTP-based, making use of JSON at several points, using JavaScript for building the interface, validating user input and communication with the backend seems the natural choice. Even better yet: If the processing pipeline is indeed set up from components that can transparently be created in several proramming languages, it might be possible to re-use the OmNom Data Model directly in the frontend, lifting the UI developers from the tedious and error-prone task of storing data in the DOM or mapping the backend data model to the frontend data model using a JS framework like Ext.JS or BackboneJS. While on the subject: It might even be worth to consider implementing the backend in Node.js, which is both performant, asynchronous by nature and wouldn't require any backend-frontend-translation at all.

Apart from the scripting, standard WWW technologies like CSS and HTML should be used.

3.4.2 Views to implement

User Preferences Let the user set preferences, like default settings for certain attributes of the components

Component List List the available components, ready to be sorted and filtered by arbitrary criteria

Configuration List List configurations that a user can re-use

File CRUD Let the user create/retrieve/update/delete data in the data storage

Pipeline Creator A view that combines a list of components, a list of configurations and an empty list. The user can then add components to the empty list which creates the configuration for that component. The same should happen when the user adds a configuration, except that multiple conifugrations are added at once. The pipeline should pre visualizable, so the user can be sure she has the workflow right before starting it. Starting it should serialize the list to a JSOn structure, ready to be sent to the backend and/or job API.

Pipeline Wizard A multi-form questionnarire for drilling down what a user wants and then providing her with a pre-computed pipeline. For example, knowing that a data provider uses TEI and wants to ingest into the DM2E triplestore, Loader, Transformer, Linker and Publisher can be set, leaving to the user the task of adding a Transformation that addresses his data's specifics.

Job list List of jobs, filterable by user, state, priority. . .

Job Details Detail one particular job, list log messages, status, priority, dates associated . . . Give user the possibility to change status

XML element chooser Display the hierarchy of elements of an XML document as a tree and let the user select one

4 Tools and Libraries to be integrated

In this section, the tools and libraries with direct impact on the RDFization process will be discussed and other projects that are not directly re-usable but might serve as an inspirational input, will be briefly described. They are broadly categorized by the component type they will be integrated as.

4.1 Loader

This section describes the data formats OmNom will have to be able to load. After a description of the format itself, an example for a library to actually use is given - the ones used in the prototype.

4.1.1 RDF - RDF::Trine

RDF (Resource Description Framework) is a way for structuring information in graphs of resources that can be linked to data-typable literals and other resources by URL-identifiable links. The goal of DM2E is to transform tabular, flat and hierarchical bibliographic (meta)data formats into this graph structure, a specific subset thereof defined by the EDM+ schema to be exact. A notworthy aspect is that provenance in DM2E (for tracking the origins of user-generated content) is going to be implemented using Named Graphs, that is an augmentation of the traditional Subject-Predicate-Object triple to a Subject-Predicate-Object-Context quad.

The RDF loader must be able to parse from and serialize to the various forms for storing triples/quads (RDF/XML, TURTLE, N-TRIPLES, N-QUADS). It must support different storage backends, at least in-memory and using a SPARQL endpoint using SPARQL/Update. It must support SPARQL SELECT and CONSTRUCT queries for searching and constructing. An easy to use component for simple (RDFS) reasoning would be a plus, though it isn't strictly necessary, since most simple reasoning (which is all that OmNom might need) can be achieved by SPARQL CONSTRUCT queries with property paths.

RDF::Trine[4] is a RDF framework for the Perl programming languages. It includes components for parsing and serializing RDF in various formats, interfaces to different triple stores and a SPARQL engine. Full Disclosure: The author is a contributor to RDF::Trine and knows it fairly well and is therefore inclined to use it. However, Jena[5] and Sesame[6] are the standard choice in the Java world and have the additional advantage of being the frameworks used in most RDF programming example code.

[4]https://metacpan.org/module/RDF::Trine.
[5]http://jena.apache.org/.
[6]http://www.openrdf.org/.

4.1.2 XML - LibXML

eXtensible Markup Language is a tree-based data structure and tree serialization format. With a degree of probability tantamount to certainty the reader of this report is familiar enough with XML to spare her or him the specifics of the format.

On the implementation side of things, it is important to differentiate between parsing, serialization and tree traversal. Parsing a large XML file at once into a tree structure has severe disadvantages in many cases. Therefore there are tree parsers and streaming parsers, which handle opening and closing of elements as events, a specific parser can hook into. Tree traversal performance can vary widely between implementations due to the underlying data structure used, how aggressively memory is reserved etc.

XML functionality is often provided by XML frameworks that combine various libraries for handling XML parsing, handling, XSL transformation, validation and more. One such framework is LibXML,[7] originally developed within the context of the Gnome Desktop Environment, but now widely used in other contexts as well. LibXML is written in C with bindings for a variety of languages. It encompasses serializers, bulk and streaming parsers and an XSLT 1.0-compliant XSL processor (LibXML, see 4.2.1).

4.1.3 CSV - Text::CSV

CSV (Character Separated Values) are a way of serializing tabular data by defining an ASCII character to serve as the field divider, the record being a single newline. By convention, the separating character is `,` (COMMA) and CSV the abbreviation for Comma Separated Values. Other common separators are `\t` (TAB) or `|` (BAR). The values of field can be optionally encased in quotation marks.

While seemingly trivial to parse, there is a variety of places parsing can break, most notably the handling of escaped characters. Therefore using a dedicated library for parsing CSV is highly recommended. One such library is Text::CSV[8], written in Perl/C, that has been in development for 10+ years, handles all the edge cases and is very fast.

4.1.4 MARC - MARC::Record and MARC::XML

MARC (MAchine Readable Cataloguing) is a bibliographic data exchange format widely used by libaries around the globe. It is a key-value based format, with limited suppot for hierarchies (subfields) and a wide variety of semantics attached to the fields. The most relevant *dialects* of MARC in the context of

[7] `http://www.xmlsoft.org/`.
[8] `https://metacpan.org/module/Text::CSV`.

DM2E are US-MARC and UNI-MARC. MARC can be serialized in a native format using lower ASCII for field, value and record separation, as well as in an XML serialization.

Because MARC's semantics relies on the position of characters in some fields and the default character encoding predates the inception of Unicode by twenty years, parsing MARC is non-trivial. MARC::Record[9] and MARC::XML[10] are Perl libraries for handling MARC21 records in both it's flat file orginial format and the XML-based MARC-XML serialization. They have been in development for around fifteen years, are used for libary automation and there is a mailing list for Library-related Perl modules,[11] where questions are answered in a timely and helpful manner.

4.1.5 MAB2 - MAB::Record

MAB2 (Maschinelles Austauschformat für Bibliotheken) is Germany's answer to a bad data format: A worse data format but in German and only for Germans. As such, MAB2 brings the same disadvantages and parsing difficulties as MARC but with German key names. As with MARC, there is a traditional serialization and an XML variant, character encoding issues are as abound as they are with MARC.

MAB::Record[12] is a recent effort by SBB's Johann Rolschewski to port the MARC::Record API to MAB data semantics. While not yet published in the CPAN, it is very usable and seems bug-free, apart from the usual text encoding nuisances that afflict all text handling tools. MAB::Record can convert between XML and traditional serialization.

4.2 Transformer

This section describes various data transformation tools relevant for DM2E. Transformation is used in the broad sense of data format and serialization conversion.

4.2.1 XSLT - Saxon and LibXSLT

XSL (eXtensible Stylesheet Language) is a functional programming language that can transform XML to XML or text. XSL stylesheets is written in XML as a set of templates that are applied to the input document or elements therein by defining XPaths. XSL transformers (XSLT) supports the evaluation of functions, which can be used to improve pattern matching, for string manipulations or as bridge to external data. However, the number of pre-defined functions in the XSLT '1.0 standard is very low. XSLT 2.0 is much more flexible

[9] `http://metacpan.org/module/MARC::Record`.
[10] `http://metacpan.org/module/MARC::XML`.
[11] `http://perl4lib.perl.org/`.
[12] `https://github.com/jorol/MAB-Record`.

in this regard but the standard is fully implemented by only one vendor - who coincidentally is also the main author of the specifications. Many XSL transformer allow the user to define her own functions and make them available to stylesheet authors.

The aforementioned only XSLT 2.0-capabable XSL transformer is Saxon.[13] Saxon is available for Java, has superior function support and allows meta-functions such as splitting a source tree into subtrees (useful for splitting up trees of individual records as e.g. in OAI-PMH into the records in a standard-compliant way). Unfortunately, Saxon is proprietary, commercial software. While it is also offered in a *Home Edition*, this version is functionally very restricted, e.g. without support for custom functions. Where XSLT 2.0 support is essential, Saxon must be usable nonetheless.

A very good v1.0-compatible XSL transformer is LibXSLT[14] which shares a development group with LibXML (see 4.1.2). It has support for custom functions. It is written in C (is as such very fast) and has bindings for many scripting languages.

4.2.2 MINT

Mint will be the main tool for XML-to-RDF transformation. As XML is such a widely used format and almost all traditional data structures can be serialized in XML without information loss, the processing of XML is the most important aspect of the platform. Mint is a Java-based application, consisting of a transformation engine based on XSLT, a rich user interface written in Javascript (Yahoo YUI specifically) and JSP and styled using CSS. Since MINT is used in many projects, it has been heavily customized. There is even a Mobile Interface that resembles an Android/iPhone App.[15]

OmNom considers MINT to be an advanced WYSIWYG editor for XSL transformations with additional Linked Data features. As for the backend, only the resulting XSLT stylesheet is relevant and ways to retrieve them will have to be implemnted. The integration of the MINT user interface will be purely for user convenience, i.e. to allow her to start a mapping editor session in MINT from within the OmNom user interface or to create a new mapping from scratch by using a loaded XML document as the canvas to work on.

4.2.3 D2RQ

D2RQ is a mapping language that allows accessing Relational Database Mangement systems as Linked Data by translating RDF-based queries to SQL and vice versa. The general procedure is to generate an initial set of mapping rules by letting the D2R software introspect the database and the iteratively refine

[13] `http://www.saxonica.com`.
[14] `http://xmlsoft.org/XSLT/`.
[15] `http://oreo.image.ece.ntua.gr:9990/partage/Login.action`.

this mapping to perfection. To grasp the changes of a new version of the mapping file, it has to be tested. This can be done programmatically, by running unit tests with e.g. SPARQL queries, or manually, by browsing the results. While the former approach is far superior, it requires a programmer and is rather tedious and time-consuming. Therefore, a manual feedback possibility is what OmNom currently aims at. For this purposes, D2R Server can be used which integrates both the mapping engine and a Linked Data frontend. Users can edit their mapping file within OmNom and preview the results of the current mapping using a customized D2R Server. If this proves unsuitable for performance reasons, it might be better to just use the D2RQ command line mapping tools and provide a native Linked Data frontend such as RDF::LinkedData or Pubby.

4.2.4 R2R

R2R[16] is a framework for mapping RDF vocabularies. It consists of a mapping language that is similar to SPARQL. Mappings can be defined in this SPARQL that declare rules on how to transform literals, resources or graph patterns in the source to the desired output structure.

R2R is a much more powerful of the SPARQL CONSTRUCT query, which allows to create new graphs by re-organizing the elements from a source pattern in a new way. However, R2R is made specifically for this purpose, whereas SPARQL CONSTRUCT is a very general query type. Advanced transformations such as the ubiqiuituos problems of differing phrasings of names (e.g. *John T. Smith; Smith, John T.; Smith, John <author> . . .*) can be solved by R2R in a very expressive and still terse form. This is very important for good results in the link discovery process (cf. 3.2.7).

R2R can also solve problems where record identity does not necessarily coincide with what EDM+ considers to be the ProvidedCHO. As long as there is no support for FRBRizing entities in EDM+, there is bound to be lots of propagation of predicate-object pairs throughout the graph (e.g. in the case where a provider differentiates between manifestation and work in it's interna;l data store, they are likely to use the URL for the work as the `edm:WebResource` for the work *and* all the manifestations. Using R2R for graph manipulations like this makes this step easier and less error-prone than to force the providers to flatten their data before ingestion.

4.2.5 Culturegraph Metamorph and Metaflow

Metamorph and Metaflow component of Linked-Open-Data-Service developed by the German National Libary. Metamorph is a software for transforming bibliographic formats, such as MAB2, MARC21 and PICA. Metaflow is a the implementation domain-specific scripting language for creating Metamorph data

[16]`http://www4.wiwiss.fu-berlin.de/bizer/r2r/`.

flows. The framework is written in Java and heavily in development but contacts between DM2E and Culturegraph have been established and a cooperation is planned.

4.2.6 ClioPatria XMLRDF and Amalgame

XMLRDF and Amalgame are tools that fit within the ClioPatria semantic web server developed by University of Amsterdam. XMLRDF is the component that handles transformation from XML to RDF using a sophisticated rule engine. Amalgame is a linking tool similar to Silk in functionality.

ClioPatria is written in SWI-Prolog which is unusual, but an understandable choice, considering how closely a rule-based language like Prolog and a graph-based structure like RDF are related. It is hard to interact with it on the language level, a command-line based approach or interaction via a HTTP API seems more feasible.

4.2.7 jMet2Ont

jMet2Ont[17] is a very recent effort by Poznan Supercomputing and Networking Center to create a XMl-to-RDF transformation engine. It is a unique approach in that it handles transformation neither rule-based nor on pure syntactics, but on ontology paths. JMet2Ont is written in Java and the transformation is controlled using a XML file.

Right now, jMet2Ont is still in it's infancy but it looks promising and DM2E is in contact with the developers. A simple interface to jMet2Ont would be easy to do and would increase the visibility of the tool which might in turn motivate the creators to create an easier to use user interface.

4.2.8 Javascript Engine - PhantomJS/V8

Every component is backed by software. To adapt the software to the users needs, configuration data needs to be given to the component who translates it to a correct call of the software. This means that a user mustn't only translate her data transformation needs to the framework of a specific tool (e.g. a MINT and then an XSL processor), but abstract one step further to the OmNom framework. While this is in many cases even desirable so the casual user is protected from the intricacies of the implementation, advanced users may find it easier to "just hack" their desired transformations in a scripting environment.

Javascript/ECMAscript has evolved from it's early days as a simple sandboxed language for adding basic client-side scripting to WWW browsers to span the breadth of areas that other scripting languages such as Perl, Python or Ruby occupy already. Sophisticated DOM handling libraries such as jQuery

[17]`http://fbc.pionier.net.pl/pro/jmet2ont/`.

and underscore are arguably the most frequent and easiest form of XML manipulation in use today. Therefore, it would be helpful to integrate a Javascript environment into the toolchain.

A conceivable setup would be PhantomJS (which is in essence a headless WWW browser) with some helper libraries for different data formats.

4.3 Linker

4.3.1 Silk

Silk is a Java-based framework for aligning data sets in the Semantic Web. The user defines a set of rules that define when two resources in different data domains should be considered identical or at least sufficiently similar to store a relation between the two (e.g. `owl:sameAs`). Another use-case is turning literals into resources by doing complex searches for strings and replacing/augmenting the string versions with a link to the resource, e.g. replacing the literal `"London"` with the URL `http://geonames.org/place/london`). Silk's strengths are it's high performance (the process can be distributed across multiple computers), the flexibility of the linking rule language and the configurable thresholds for linking results quality: Silk partitions the linking results by how certain it is the results are correct. Some results are certainly correct, some are probably correct, some might be correct, some could be, etc. It is up to the user how to react to those uncertainties.

What Silk is lacking, is an easy-to-use and easy-to-integrate User Interface. Silk is run by providing a configuration file that contains the linking rules, the location of source, lookup and target datasets. For novice users, an editor component that makes editing those configurations easier might be helpful (e.g. XML highlighting, snippets, help), experienced users could just provide a link to the configuration.

A thing to note about Silk is that it is best run in batch mode, while the general approach of the components in OmNom is to handle one record at a time. Therefore, Silk is probably best used as the last component in a pipeline that starts with a Splitter componennt, i.e. for running on batches of MAB2 or MARC records.

4.3.2 LIMES

LIMES (LInk discovery framework for MEtric Spaces)[18] is a relatively new link discovery framework similar to Silk in functionality, though with a different approach to similarity compution. It can reduce the number of comparisons drastically by determining a similarity threshold before running the actual alignment and filtering out too distant elements, which would be noise otherwise.

[18]`http://aksw.org/projects/limes.`

LIMES has a very user-friendly web interface and is well documented. While using Silk is the more appropriate choice, because it has been around for several years, is proven to be very performant and DM2E is proud to employ one of the Silk creators, keeping a look at and maybe implement an interface for LIMES seems reasonable, if for nothing else, then for it's really impressing web interface.[19], which might not only serve as inspiration for a Silk interface but for the workflow composer view described in section 3.4.2 and Figure 4.

4.4 Splitter

4.4.1 XML

If the case occurs that multiple records are stored in one XML document, the simplest way is to make the user provide an XPath and compute the subtrees using a simple tree traversal. Virtual all OAI-PMH harvesters – the most probably use case – have built-in support for splitting up the OAI-PMH responses into records.

The natural choice would be to use XSLT for this and it *is* possible to split up a document into subtrees and write them out to files in XSLT 2.0, but this is only supported by Saxon and could be highly inefficient due to the I/O overhead for XML documents with lots of records.

4.4.2 MAB2, MARC

Splitting batches of MAB2 or MARC records into individual records can either be delegated to a library (both libraries mentioned in 4.1 support that) or the record batches can be converted to XML and the resulting XML transformed to individual records using the method mentioned in the previous section.

4.4.3 OAI-PMH - OAI::Harvester

OAI-PMH is a widely used metadata distribution protocol used in the GLAM area and at least one of the data providers intends to ingest data via OAI-PMH into DM2E. It is a HTTP- and XML-based protocol and while format-agnostic in theory, it contains only flat Dublin Core in practice.

Net::OAI::Harvester[20] is an actively developed OAI-PMH harvester written in Perl.

4.5 Validator

4.5.1 XML Schema - LibXML Schema

XML Schema (XSD) is a language for defining how XML documents are structured, constraining element names, element order, element values, element

[19] `http://limes.aksw.org/colanut/.`

[20] `https://metacpan.org/module/Net::OAI::Harvester.`

containment and so on. XSD is used for datatyping in RDF as well. Many XML-based format define the specifics of their format in XSD, which makes it easy to validate instances of data in such an XML format to be automatically validated. MINT uses XSD for introspection for it's XML handling, both on the interface level (tree of elements) and the mapping level (required elements) and as such, an EDM+ XML Schema will be implemented by NTUA in late 2012. This schema could be re-used in OmNom for general validation if the outcome of a transformation to EDM+ is a valid graph according to the constraints.

LibXML is again (see sections 4.1.2 and 4.2.1) the most straightforward solution, offering basic XSD validaiton, which should be enough for OmNom's purposes.

4.5.2 Schematron - XSLT

Schematron is a rule-based language for validating the structural and syntactical correctness of an XML document. Other than XML Schema, described above, it is just a set of derived XSL stylesheets and is processed using an XSL transformator. This means that there is a Schematron *meta-stylesheet* which is compiled with a Scematron XML document that defines the constraints and assertions. The instance data is then transformed using this derived XSL stylesheet and the result is another XML document, listing all the broken rules. This is a much more usable approach for debugging XML document transformation, because other than XSD it doesn't break on the first error and it can break very verbosely. The assertions defined in Schematron are more flexible than those in XSD, too.

SBB has started a Schematron schema for EDM+, which could be used as a second step or an alternative to XSD validation.

Since Schematron is just XSLT with a twist, the technologies listed in 4.2.1 suffice to use it.

4.6 Publisher

4.6.1 File System - local and SSH

The easiest way to store data has been the file system. The user defines a path or a path pattern and the backend stores the data in files at the appropriate location. Using SSH (Secure SHell), it is also possible to treat remote filesystems like local filesystems, fully encrypted , provided an encryption details exchange scheme has been set up beforehand.

The easiest solution to access SSH shares from an application is to mount the remote host via SSHFS[21] from outside the application beforehand. This leaves the transportation and encryption layer to the operating system, making file access transparent to the application.

[21]`http://fuse.sourceforge.net/sshfs.html`.

4.6.2 HTTP

HTTP (Hyptertex transfer protocol) isn't a storage engine, but a network protocol. However there are so many storage engines that expose an API via HTTP, that a generic component implementing HTTP based data transfer is sensible.

The easiest form of transfer of large chunks of data via HTTP is a HTTP POST with Content-Type `text/www-form-urlencoded`. This is analogous to submitting an HTML form with a field of type texttfile. Other, more programming-friendly APIs to store data based on HTTP are easy to implement or re-use (most Content Management Systems offer such interfaces). This might be the simplest solutions for data provider to review the results of their ingestion workflows.

Virtually all programming languages have libraries for implementing HTTP client and server functionality.

4.6.3 E-Mail

4.6.4 Distributed Version Control System - Git

A VCS is a repository for tracking the evolution of files, used in virtual every software development project, but usable in different contexts. Examples for classical VCS are CVS or Subversion. DVCS are the next-generation of VCS, which don't require a central repository server to be set up. Instead, every developer can fork the repository, cloning all the files and the complete history. Changes to the Repository can then be pushed, pulled and merged from/to any of the forks. The most widely used DVCS are Git, Mercurial and Bazaar.

OmNom strives to use a DVCS for the storage of component configurations because DVCS are proven to work for features which would otherwise have to be error-pronely implemented by hand. Wheels not to re-invent include of course versioning, authentication, file storage and data exchange.

Arguably, even manuscript raw and intermediate data could be stored in a DVCS, though VCS in generally don't handle very large repositories (>10GB) and binary files (such as images) very well. Which DVCS to use is open to debate, the author prefers Mercurial for being more light-weight and straightforward, others might prefer Git for it's ubiqiuituousness and the multitude of tools and libraries that exist for interacting with it.

There are libraries for programmatically interacting with Git repositories in many languages. One especially straightforward to use is Git::Wrapper[22] for the Perl programming language, it's API is just a thin wrapper around the command line usage of the `git(1)` command.

[22]`https://metacpan.org/module/Git::Wrapper.`

4.6.5 Distributed File System - GridFS and HDFS

Distributed file systems are data storage engines that distribute the storage across multiple computers. Similar to RAID, there are multiple reasons for distributing file storage: to increase capacity, to improve reliablity by multiplexing content, to improve retrieval time by distributing the I/O overhead etc.

One area where distributed file systems could be used in OmNom are the storage of large quantities of (temporary) data that is too big or too unstructured to be sensibly stored in a Version Control System. For instance, data providers might prefer to send in their records in bulk as zipped archives or in the original form of distribution, as an archive including thumbnail and hi-res pictures of the manuscript pages. Another example are large *lookup* XML files to be cached for later re-use, such as localization/internationalization data or the record attachments used in TEI.

Examples of distributed file systems are GridFS,[23] which is part of MongoDB or Hadoop's HDFS.[24] Both can be used from a variety of scripting languages and have been known to scale very well.

5 An OmNom Prototype

This section describes the prototype of the platform that has been developed while writing this report. The prototype is, as the designation implies, not the finished product. It is in fact, not even an alpha version of the product but a set of tools that implement interfaces and interact in a manner similar to that of the product to be produced later on. This has been rapidly prototyped between 2012-09-04 and 2012-09-23 and as such is bound to contain not-loosely coupled components, inconsistent naming, hard-coded data and tight requirements on the environment. Though it has been successfully deployed on Linux systems, the process is requires extensive knowledge of the inner workings and dependencies and - as a final self-deprecating comment - documentation beyond this document, a few text files and some inline comments sprinkled throughout the $\approx$ 3000 SLOC.

The following subsections outline it's technological choices, define what's there and what isn't and show the functionality using screenshots from the prototype frontend.

5.1 Technology Stack

The backend is completely written in Perl, using the Moose metaobject framework as the object system. Moose is a remarkably practical choice for very flexible interfaces, because it has Roles, which can be used like a Java Interface, but with added behavior. Roles can even be added at object instantiation time,

[23] `http://www.mongodb.org/display/DOCS/GridFS+Specification.`
[24] `http://hadoop.apache.org/docs/hdfs/current.`

which makes this very flexible for different implementations of functionality (e.g. adding asynchronicity to the context, replacing one XSLT backend with another) without having to change a single line of code. Another great feature of Moose is the type constraint and coercion system that allows one to use the wonderful *Throw any data structure at new and be given an instance* pattern without polymorphism and duplicated if-then-else-cascades all around.

The HTTP API is implemented on top of the Dancer Microframework, a minimalist but complete and easy-to-use Web framework. The API is an application, based on the Plack server middleware, that can be deployed on a number of performant servers. This is mentioned here just to make it clear that this is a persistent application that is started but once and not for every request as with CGI.

A frontend is implemented as a separate Dancer application, that accesses the API and serves some content where that is sensible. The main user interface logic is implemented in Javascript using the BackboneJS MVC framework. The relevant models of the backend are implemented as BackboneJS models, which are rendered using views. This makes it very simple to synchronize the state of elements in the HTML DOM to the model they represent. Communication between the backend API and the frontend happens using JSON which is very easy to do using jQuery's AJAX functionality.

The data handling components are implemented in Perl as well. They virtually contain no functionality of their own but delegate the loading, transformation, splitting and so on to external modules from the CPAN. Since libraries, archives and libray automation software vendors have a long tradition of using Perl for administrative tasks and as a scripting language. Because of that, there is at least one CPAN modules for pretty much any metadata format used within the framework, if not on the data format level, then on the serialization level.

Jobs are handled by the Gearman Job Server, which starts ingestion runs as necessary. Job data and session information are stored in a MongoDB database - which is both fast and straightforward to use for JSON-structured data.

5.2 Implemented functionality

In compliance with the overall structure, the prototype has been developed in the areas backend, components, API and UI.

5.2.1 Backend

The backend is working mostly as it is supposed to: Contexts can be created, with buckets for specific data types defined by the components. Per default, calling the contexts's `work` method will start the process, call the components' `work` method with the context as parameter in sequence and finish when there is nothing left to do. Alternatively, at instantiation time of the context, a `HasJob` trait can be added to the context which turns the run of the `work` method into a job persistable in a database.

Since the prototype consists of four servers (MongoDB for session storage, Gearman as the job server, a worker-spawning dummy script that actually starts the jobs and a web server for the API and UI), there is a Bash script `omnomd` that can start/stop/restart these servers individually or at once, e.g.

`omnomd START` to start the whole application
`omnomd RESTART MONGODB` to only restart MongoDB

5.2.2 Components

The loader components implemented can handle loading data from a string, from a file path relative to a Git repository base or from a URI. Loader plugins for the following data types exist (if differing, serializations are listed in brackets):

TEXT
XML
RDF (RDF/XML, Turtle, NTriples, NQuads . . .)
MAB2 (MAB2, MAB2DIS, MAB-XML)
MARC21 (MARC21, MARC-XML)
XSLT
XSLT::FromMint This is special, because it loads the XSLT directly from a Mint instance: The user provides Mint URI, login details and mappingID/uploadID for the mapping and the plugin simulates a login and then a preview on the selected mapping/upload, retrieves the XSLT and loads it.

The following splitters have been implemented:

MAB2 Splits up a batch of MAB2 records into individual records and runs ingestion with every record as input
OAI-PMH Harvests an OAI-PMH repository and runs an ingestion with every record as input

The following data munging components have been implemented:

Nop Does nothing for configurable amount of time (debugging)
XSLT XSLT transformer using LibXSLT (i.e. XSLT 1.0)
XSLT2 XSLT transformer using Saxon (i.e. XSLT 2.0)
MAB2XML transforms a MAB2 record to MAB-XML
XML2TEXT serializes an XML document
RDF2TEXT serializes an RDF model
Schematron Validates an XML document according to some rules defined in a schematron document

JSXML run Javascript code on the DOM of an XML tree (proof-of-concept)

The following publishing components have been implemented:

LocalFile Store bucket item to a file on the (server-) local file system (debugging)

HttpRequest Set URI, HTTP verb and headers and send the contents of the bucket item as the HTTP body

HttpUpload Shortcut for sending the bucket item as form input with the bucket item as a file field's content

Git Store the contents of a bucket item as a file in a Git repository, then add that change and commit it to the repo - making it available to others to pull and clone.

5.2.3 Application Programming Interface

The API is a Dancer application that responds to HTTP requests to resources beyond the `/api` path. The following HTTP actions are possible at this time:

`GET /api/visualize?pipeline=...` Visualize an ingestion configuration in terms of data flow, data buckets and plugin sequence (c.f. Figure 6 and Figure 7)

`GET /api/doc?module=...` Get the documentation for a component, by retrieving the POD documentation of the corresponding perl module

`GET /api/git/my/path/` List the files in the directory `GITBASE/my/path` where `GITBASE` is the base directory of the server-side copy of a Git repository for configurations, mappings etc.

`GET /api/git/my/path/foo.xml` Return the contents of the file `GITBASE/my/path/foo.xml` where `GITBASE` is the base directory of the server-side copy of a Git repository for configurations, mappings etc.

`GET /api/plugin` Get a list of plugin names, their type (by means of the roles they consume, such as `Loader`, `Transformer` and their attributes (by means of metaobject introspection)

`GET /api/job/list?status=...` List all jobs, optionally filtered to list only those with status `status`

`GET /api/job/abcdefg01234` Return the details of the job with ID `abcdefg01234`

`POST /api/job` Post a new job by sending an array of component configurations. Returns the ID of the job posted

`POST /api/file` Upload a file to the GridFS filesystem provided by MongoDB. Useful for storing input data to a HTTP-resolvable URL without hassle. Returns the URL of the file

`GET /api/file/` List all the filenames in the GridFS filesystem.

`GET /api/file/abcdefg01234` Retrieve the content of the file with ID `abcdefg01234` from the GridFS filesystem.

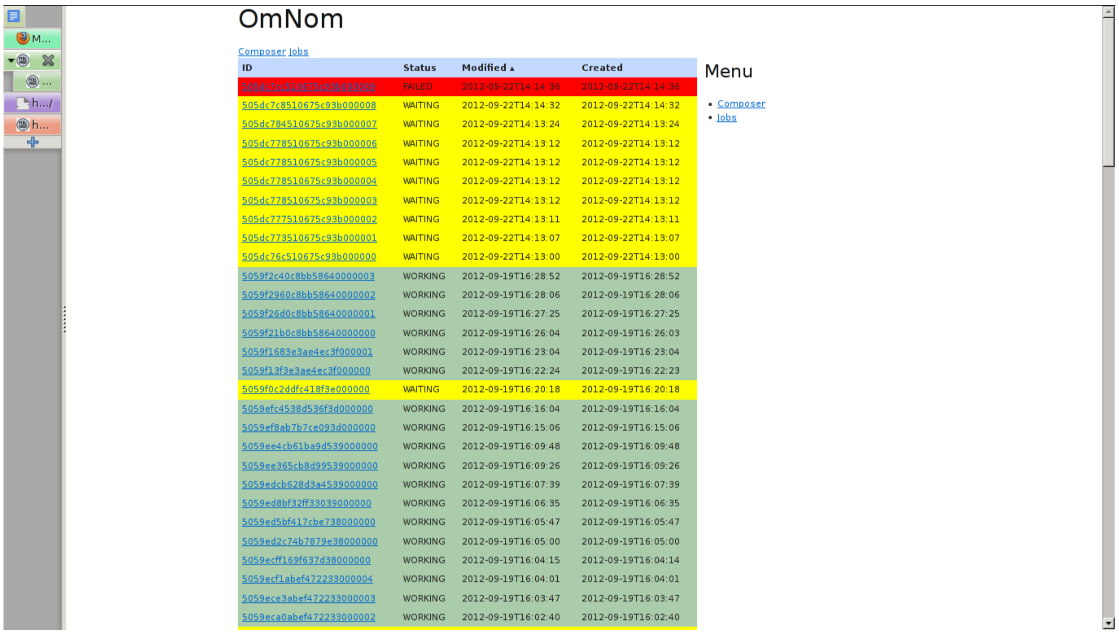
There are actually some other API functions that aren't listed here because they are most likely going away in subsequent versions. The `/api/file` calls aren't necessary and probably just a maintainance burden. Providers should be required to store the data that components need to retrieve by URL at a HTTP-resolvable path or by some other means (e.g. pushing those files to their copy of the Git repository and sending a pull request to the server-side Git repo.

5.2.4 User Interface

The user interface is browser-based, consisting of a web server that serves pre-generated content for some simple cases (such as the job listing and job details page), as well as Javascript code, CSS and structural HTML.

The main functionality, i.e. the creation of ingestion workflows, is realized using a Javascript application which communicates with the API using AJAX/JSON. The application is is based on the BackboneJS Javascript MVC framework, which has a somewhat steep learning curve for the occasional jQuery programmer, but leveraging it's semantics really pays off in the long run.

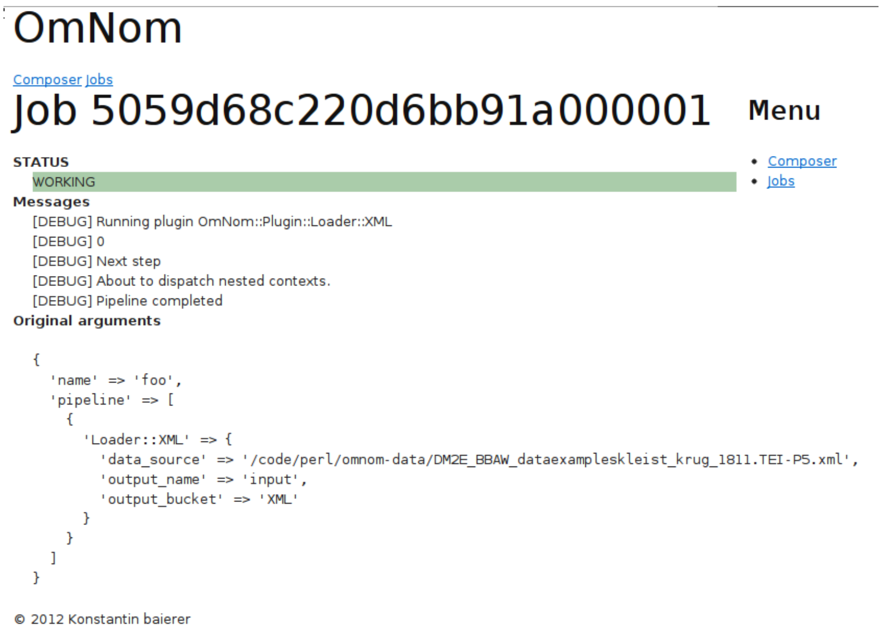
The state of the user front-end is probably best explained using some screenshots and descriptions.



ID	Status	Modified ▴	Created
505dc7cc510675c93b000009	FAILED	2012-09-22T14:14:36	2012-09-22T14:14:36
505dc7c8510675c93b000008	WAITING	2012-09-22T14:14:32	2012-09-22T14:14:32
505dc784510675c93b000007	WAITING	2012-09-22T14:13:24	2012-09-22T14:13:24
505dc778510675c93b000006	WAITING	2012-09-22T14:13:12	2012-09-22T14:13:12
505dc778510675c93b000005	WAITING	2012-09-22T14:13:12	2012-09-22T14:13:12
505dc778510675c93b000004	WAITING	2012-09-22T14:13:12	2012-09-22T14:13:12
505dc778510675c93b000003	WAITING	2012-09-22T14:13:12	2012-09-22T14:13:12
505dc777510675c93b000002	WAITING	2012-09-22T14:13:11	2012-09-22T14:13:11
505dc773510675c93b000001	WAITING	2012-09-22T14:13:07	2012-09-22T14:13:07
505dc76c510675c93b000000	WAITING	2012-09-22T14:13:00	2012-09-22T14:13:00
5059f2c40c8bb58640000003	WORKING	2012-09-19T16:28:52	2012-09-19T16:28:52
5059f2960c8bb58640000002	WORKING	2012-09-19T16:28:06	2012-09-19T16:28:06
5059f26d0c8bb58640000001	WORKING	2012-09-19T16:27:25	2012-09-19T16:27:25
5059f21b0c8bb58640000000	WORKING	2012-09-19T16:26:04	2012-09-19T16:26:03
5059f1683e3ae4ec3f000001	WORKING	2012-09-19T16:23:04	2012-09-19T16:23:04
5059f13f3e3ae4ec3f000000	WORKING	2012-09-19T16:22:24	2012-09-19T16:22:23
5059f0c2ddfc418f3e000000	WAITING	2012-09-19T16:20:18	2012-09-19T16:20:18
5059efc4538d536f3d000000	WORKING	2012-09-19T16:16:04	2012-09-19T16:16:04
5059ef8ab7b7ce093d000000	WORKING	2012-09-19T16:15:06	2012-09-19T16:15:06
5059ee4cb61ba9d539000000	WORKING	2012-09-19T16:09:48	2012-09-19T16:09:48
5059ee365cb8d99539000000	WORKING	2012-09-19T16:09:26	2012-09-19T16:09:26
5059edcb628d3a4539000000	WORKING	2012-09-19T16:07:39	2012-09-19T16:07:39
5059ed8bf32ff33039000000	WORKING	2012-09-19T16:06:35	2012-09-19T16:06:35
5059ed5bf417cbe738000000	WORKING	2012-09-19T16:05:47	2012-09-19T16:05:47
5059ed2c74b7879e38000000	WORKING	2012-09-19T16:05:00	2012-09-19T16:05:00
5059ecff169f637d38000000	WORKING	2012-09-19T16:04:15	2012-09-19T16:04:14
5059ecf1abef472233000004	WORKING	2012-09-19T16:04:01	2012-09-19T16:04:01
5059ece3abef472233000003	WORKING	2012-09-19T16:03:47	2012-09-19T16:03:47
5059eca0abef472233000002	WORKING	2012-09-19T16:02:40	2012-09-19T16:02:40

Figure 1: List of jobs currently in queue

Figure 1 shows the list of jobs as a sortable table. Jobs are color-coded according to their status. Green jobs have completed successfully, while red jobs have failed. Yellow jobs are currently running.



OmNom

Composer Jobs

Job 5059d68c220d6bb91a000001

Menu

- Composer
- Jobs

STATUS

WORKING

Messages

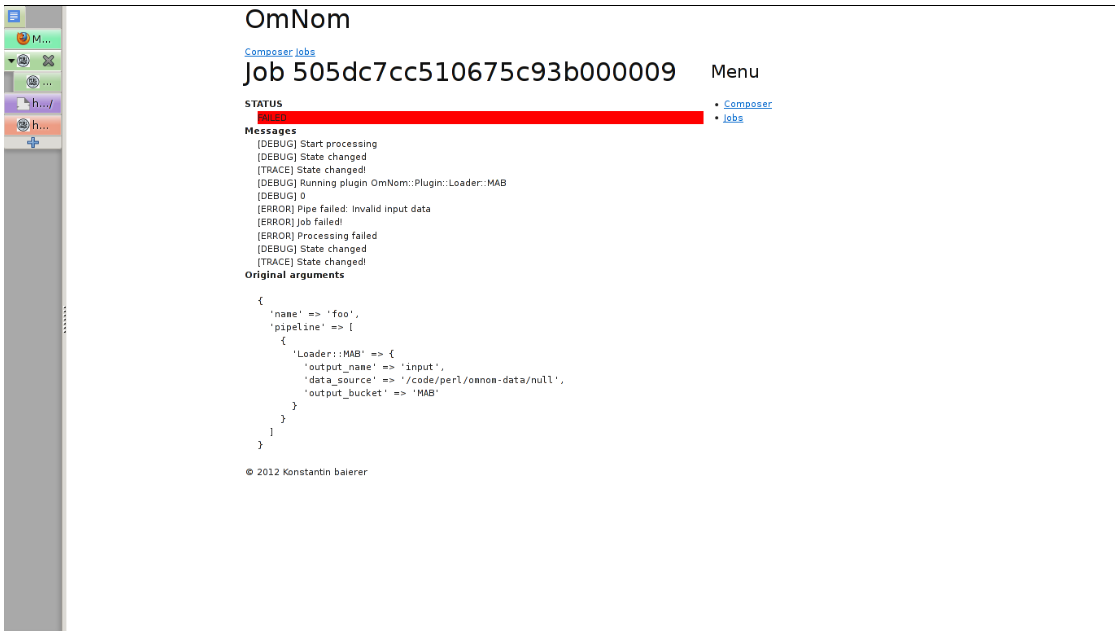
```
[DEBUG] Running plugin OmNom::Plugin::Loader::XML
[DEBUG] 0
[DEBUG] Next step
[DEBUG] About to dispatch nested contexts.
[DEBUG] Pipeline completed
```

Original arguments

```
{
  'name' => 'foo',
  'pipeline' => [
    {
      'Loader::XML' => {
        'data_source' => '/code/perl/omnom-data/DM2E_BBAW_dataexampleskleist_krug_1811.TEI-P5.xml',
        'output_name' => 'input',
        'output_bucket' => 'XML'
      }
    }
  ]
}
```

© 2012 Konstantin baierer

Figure 2: Simple completed Job



OmNom

Composer Jobs

Job 505dc7cc510675c93b000009

Menu

- Composer
- Jobs

STATUS

FAILED

Messages

```
[DEBUG] Start processing
[DEBUG] State changed
[TRACE] State changed!
[DEBUG] Running plugin OmNom::Plugin::Loader::MAB
[DEBUG] 0
[ERROR] Pipe failed: Invalid input data
[ERROR] Job failed!
[ERROR] Processing failed
[DEBUG] State changed
[TRACE] State changed!
```

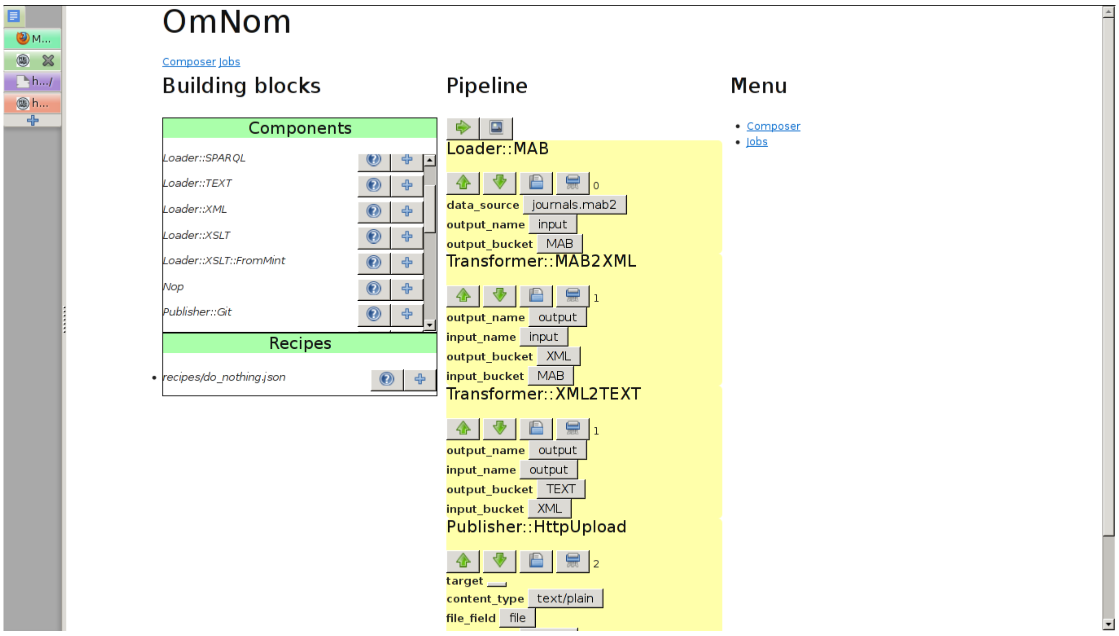
Original arguments

```
{
  'name' => 'foo',
  'pipeline' => [
    {
      'Loader::MAB' => {
        'output_name' => 'input',
        'data_source' => '/code/perl/omnom-data/null',
        'output_bucket' => 'MAB'
      }
    }
  ]
}
```

© 2012 Konstantin baierer

Figure 3: Simple failed Job

Figure 2 and Figure 3 show the detailed job descriptions for a completed, resp. failed job. Note that every job stores the arguments for setting up the workflow and the logging messages that components emitted during the run. This should make it easier for the experienced user to quickly spot at which point the ingestion failed and why.



OmNom

Composer Jobs

Building blocks

Components

Loader::SPARQL
Loader::TEXT
Loader::XML
Loader::XSLT
Loader::XSLT::FromMint
Nop
Publisher::Git

Recipes

- recipes/do_nothing.json

Pipeline

Loader::MAB
0
data_source journals.mab2
output_name input
output_bucket MAB

Transformer::MAB2XML
1
output_name output
input_name input
output_bucket XML
input_bucket MAB

Transformer::XML2TEXT
1
output_name output
input_name output
output_bucket TEXT
input_bucket XML

Publisher::HttpUpload
2
target
content_type text/plain
file_field file

Menu

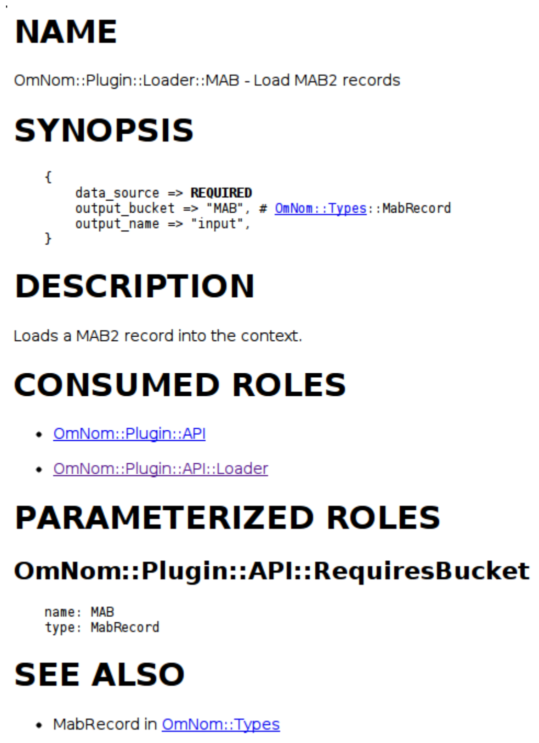
- Composer
- Jobs

Figure 4: OmNom composer

Figure 4 is the heart of the User Interface. The canvas has three parts: A section for a list of components to add to the pipeline (left-top), a list of *recipes*, i.e. re-usable configurations to insert into the pipeline (left-bottom), and the pipeline itself (middle column).

Every component and every recipe has both a help button and an add button. Clicking the help button should present the user with a concise summary of how this component is to be used, which attributes it supports, which attributes are required and other usage notes. The easiest solution is to just extract the documentation of the source code of the component, which yields for example the page in Figure 5. Clicking the add button on a component deeply copies the default configuration of a component and adds it to the pipeline. Clicking the add button on a recipe does the same for every component within the recipe.

Every component configuration in the pipeline view has four buttons: One for moving the component one step ahead in the component sequence, one for moving it one step behind, one for editing the attributes and one for removing the configuration from the pipeline altogether. Editing means that the view for the configuration being edited changes from a simple `<div>` to a `<form>`. For



NAME

OmNom::Plugin::Loader::MAB - Load MAB2 records

SYNOPSIS

```
{
    data_source => REQUIRED
    output_bucket => "MAB", # OmNom::Types::MabRecord
    output_name => "input",
}
```

DESCRIPTION

Loads a MAB2 record into the context.

CONSUMED ROLES

- OmNom::Plugin::API
- OmNom::Plugin::API::Loader

PARAMETERIZED ROLES

OmNom::Plugin::API::RequiresBucket

```
name: MAB
type: MabRecord
```

SEE ALSO

- MabRecord in OmNom::Types

Figure 5: Documentation of a component

fields that expect a file path, a list of files from the Git repository is provided using the API.

The pipeline view itself has two buttons: One for running the pipeline as a job and one for visualizing it. Clicking the former serializes the pipeline view to a JSON structure and creates a job from that using the API and opens the job details page for the newly started job in a new window. The visualization button is useful for ensuring that one does really understand the data flow and there are no accidental *dead ends*. Since access to buckets for input/output is name-based, this can happen only too easily but is quickly spotted using visualization.

For example in Figure 6, the following sequence is to be run:

1. Load MAB2 record, store in `MAB/input`
2. Transform MAB2 in `MAB/input` to MAB-XML in `XML/output`.
3. Serialize XML in `XML/output` to `TEXT/output`
4. Publish `TEXT/output` using a HttpRequest

Obviously, there are no wholes and typos in there. If there *was* a typo, say `XML/outptu` instead of `XML/output`, this would show because there would be two bucket items which are only connected to one component each, which is unrealistic (Why would a component want to read from an empty bucket item? Why would a component write to a bucket item no other component is going to retrieve?).

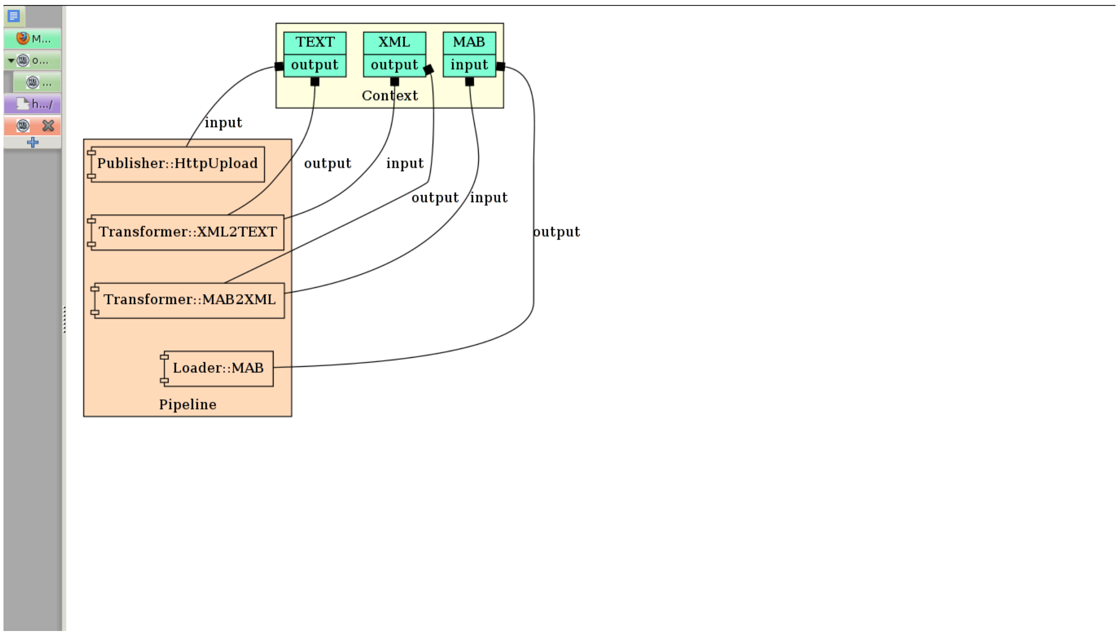


Figure 6: Visualization of a simple pipeline

An example for a visualization of a more complex workflow is in Figure 7. While tedious to explain in detail and visually overwhelming, it shows that the visualization can handle arbitrarily complex data flows.

5.3 Missing functionality?

5.3.1 Auth and Sync

There is virtually no authentication/authorization mechanisms built into the system yet, beyond the data storage layer, where Git handles it. Seeing as DM2E is a Linked Data project, it would be a nice-to-have feature to support Semantic Web technologies in this area as well, such as FOAF+SSL/WebID and a RDF-based ACL system. But while this is probably wishful thinking, a tight integration into the auth mechanisms of the supporting tools like MINT, D2RQ and Silk are high priority. Ideally, the system should be single-sign-on: The user logs into OmNom and can access his data space in MINT. This opens up another open issue: Complex systems like MINT have their own data storage system, which should be kept in sync with OmNom's, again in the spirit of a single-sign-on: If the user changes his mappings in MINT, the changes should be reflected in a timely manner in OmNom and vice versa, without requiring action on the user's part.

User Authentication: Would be cool to do that using FOAF+SSL, RDF ACL and so on, but yarr, well, it's complicated.

5.3.2 Editor components

In the current form, the only way to interact with components is by providing them with a configuration at the time the pipeline is created. Some of the items of this configuration (i.e. some of the attributes of the component classes) can easily be mapped to User Interface controls and widgeta. For instance, a field representing a file within the Git Repository can be a dropdown list of file paths and other constraints such as value ranges can be encoded in the component class as annotation data and exported and visualized to the User Interface (e.g. as value sliders). For other cases, more complex widgets might be worth implementing, e.g. a widget for choosing an element from an XML tree – this is necessary for proper interfacing with MINT as well as for splitting up XML files into subtrees.

There are cases where the user has to use an external tool to sensibly use a component. MINT is again such an example: A user might want to set up an ingestion workflow that includes a component that references XSLT from a MINT instance. Now, if this user wants to edit the XSLT, she has to leave OmNom and switch to the MINT tool, which has it's own data storage, transformation and publishing components. After the user is done adjusting her MINT mapings, the system has to sync the XSLT with MINT or – worse and more unintiuitive yet – the user has to re-load the MINT data into OmNom.

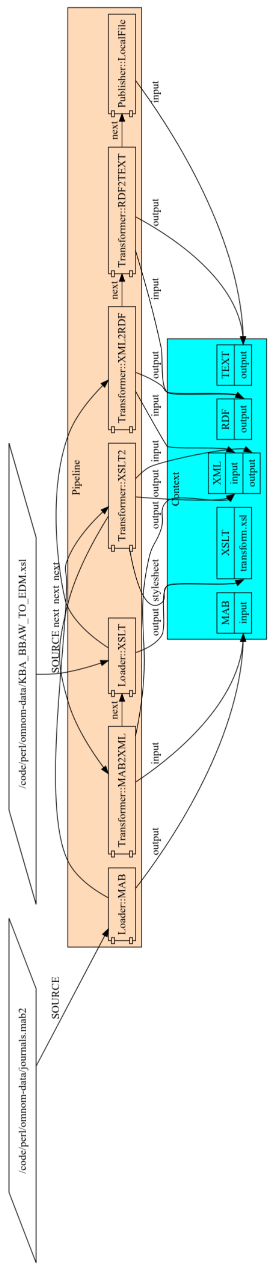


Figure 7: Visualization of a more complex pipeline

To clarify: This is a problem of the architecture only insofar as the authentication and data storage has to be exposed in such a way that it is easy to synchronize or unify both with external tools. The main work is at the frontend layer where users should be able to use external tools to create the configuration of a backend component with as little hassle as possible. If a user wants to select a MINT mapping within a transformation component, she should be able to select one from a list and click a button to edit it in a new window.

Thankfully, NTUA as the creators of the MINT tool are part of the WP2 Architecture Task Force and are actively developing ways to expose as much of MINTs functionality via APIs. This will be subject of further discussions (see 6.2).

5.3.3 Data Model

Model-View-Controller

OmNom follows the Model-View-Controller (MVC) paradigm. This means that there is a clear separation between the data domain (Model), the User and Application interface (View) and the business logic (Controller).

Model User A user is a human agent interacting with OmNom. Every Group File Correction Plugin Transformation Profile View Transformation Selection Wizard The Transformation Selection Wizard (TSW) is a multi-form questionnaire users may take to drill down their transformation profile needs in a number of steps. This view should be the first step Wrapping Functionality Authentication and Authorization Plack::Middleware::Auth::WebID File Management - Schema files - Mapping files - Should support - archives (ZIP, TARGZ) - HTTP - FTP

Functionality by processing stage

Preprocessing - Maybe scripting support (Javascript e.g. which is relatively safe) - Regular expressions?

Mapping

6 Relation to DOW, implementation timeline and testing framework

This part of the report tries to turn the findings listen earlier into actionable tasks. First, OmNom and WP2's tasks are inter-related, then the organization and preliminary results of the task force working on the issues OmNom tries to solve is discussed. Finally, a set of milestones is produced that can be used for testing and controlling.

6.1 Tasks in in WP2 of the DOW related to OmNom

This subsection is dedicated to contextualising the research into DM2E's project flow. OmNom is more than just a RDFization framework, but touches all areas of Work Package 2 of DM2E. The first subseciton therefore walks through the tasks listen in the for WP2 in the Description Of Work of DM2E and relates them to sections

Task 2.1 - Development of the RDFization Infrastructure for legacy and existing content and metadata

Sections:	2, 3, 4, 3.2, 5
Notes:	RDFization is actually pretty easy, since virtually any data structure can be expressed as a graph *somehow*. Therefore the actual XY-to-RDF transformation isn't the main focus of this report but the fine-grained definition of *somehow*.

Task 2.2 - Mapping into the Europeana Data Model

Sections:	4, 4.2.2, 3.2.2
Notes:	The actual mapping (in the sense of the distinction in 2 is not OmNom's concern, but the mapping results are and the integration of mapping tools like MINT is.

Task 2.3 - Contextualization and Interlinking

Sections:	3.2.7, 4.2.4
Notes:	Contextualization should be part of the ingestion workflow, because there seems to be little reason from the user's viewpoint to *not* have that functionality as a pluggable component in the general workflow.

Task 2.4 - Development of Workflow Management Component

Sections:	3.3.1, 3.3, 3.1, 5.2.1

Notes:	This task is touched by both the backend that manages components, context, buckets etc. and by the job interface augmenting the backend.

Task 2.5 - User Interface for Creating Mapping, Interlinking Heuristics and for Configuring the Workflow

Sections:	3.4, 5, 5.2.4
Notes:	A set of requirements for the user interface and the practical findings from developing the prototype frontend are listed in the report.

6.2 RDFization architecture Task Force

As of September 2012, Work Package 2 is focussing on two areas: The finalization of the EDM+ data model and the development of the ingestion platform. Task forces for the two areas have been organized. The Architecture Task Force has the goal to unify the various tools in the platform described in this report.

First meeting of Architecture Task Force

Date	2012-09-24
Description	A task force for creating the RDFization infrastructure is set up. MAN will lead the development of the actual implementation. MPWIG will provide coding Know-How from past projects. NTUA will be kept up-to-date at all times and will provide interfaces to MINT. EXL will lead the task force until actual implementation. A
Responsible	EXL (Konstantin Baierer), MAN (Kai Eckert), MPIWG (Jorge Urzua), NTUA (Nasos Drosopoulos, Kostas Pardalis, Fotis Xenikoudakis)

Second meeting of Architecture Task Force

Date	2012-10-08
Description	A preliminary set of interface specifications for the individual data handling components is ready, extending, clearifying and formalising the elements in this report, esp. 3.2.
Responsible	EXL, MAN, NTUA, MPIGW, SBB (Kilian Schmidtner)

Third meeting of Architecture Task Force

Date	2012-10-26
Description	Last call before the Project meeting
Responsible	EXL, MAN, NTUA, MPIGW, SBB

Fourth meeting of Architecture Task Force

Date	2012-10-30
Description	At the Vienna All WP meeting
Responsible	EXL, MAN, NTUA, MPIGW, SBB

6.3 Milestones

This section lists the most important milestones that the platform will reach in the next 36 months, from the perspective of the author of this report. The milestones are aligned with the work plan set forth by the Description of Works as well as the contract ExLibris-Baierer.

MS-01: Publication of initial report

Date	2012-09-30
Description	The first revision of this document is ready and sent out to the DM2E coordinator, WP2 leader and ExLibris.
Responsible	EXL (Konstantin Baierer)

MS-02: Prototype stable enough for demo

Date	2012-10-17
Description	The prototype of OmNom discussed in 5 is stable enough for a simple demonstation. It will be presented at ExLibris Germany HQ, ideally including a live demo, using screenshots if not feasible.
Responsible	EXL (Konstantin Baierer)

MS-03: Publication of first revised report

Date	2012-10-31
Description	All parties have been able to review the first revision of this document. Feedback from all parties is integrated to create a more coherent set of specifications
Responsible	EXL (Konstantin Baierer)

MS-04: Prototype is able to ingest UIB data

Date	2012-11-15
Description	The protype of OmNom has reached a level of stability that it is able to process the data from UIB (TEI XML) from their base format using the XSLT stylesheets pulled from MINT to EDM+.
Responsible	EXL (Konstantin Baierer)

MS-05: Phasing out of prototype, implementation of The Real Thing

Date	2012-12-01
Description	The prototype of OmNom is either replaced by a re-implementation in Java or developed further as the actual platform. Development will be distributed from now on, with MAN, NTUA, MPIWG, EXL and SBB in the team.

Responsible	EXL (Konstantin Baierer), MAN (Kai Eckert), MPIWG (Jorge Urzua), NTUA (Nasos Drosopoulos, Kostas Pardalis, Fotis Xenikoudakis), SBB (Kilian Schmidtner)

MS-06: OmNom fullfills indicator 3.4.5(2)

Date	2013-03-01
Description	OmNom is able to handle the complete workflow for 3 of the data providers
Responsible	EXL (Konstantin Baierer), MAN (Kai Eckert), MPIWG (Jorge Urzua), NTUA (Nasos Drosopoulos, Kostas Pardalis, Fotis Xenikoudakis), SBB (Kilian Schmidtner)

MS-07: Evaluation by ExLibris and WP2 leaders

Date	2013-12-31
Description	ExLibris and WP2 leaders will evaluate if the progress of the development of the infrastructure is according to the plan as defined by the contract ExLibris-Baierer.
Responsible	Konstantin Baierer, Axel Kaschte, Stefan Gradmann, Chris Bizer

MS-08: First Beta Release of the platform

Date	2013-09-01
Description	–
Responsible	MAN, EXL, NTUA, MPIGW, SBB

MS-09: OmNom fullfills indicater 3.4.5(3)

Date	2013-12-01
Description	OmNom is able to handle the complete workflow for 7 of the data providers
Responsible	MAN, EXL, NTUA, MPIGW, SBB

MS-10: Release of OmNom v1.0

Date	2014-02-01
Description	–
Responsible	MAN, EXL, NTUA, MPIGW, SBB

MS-11: Evaluation by ExLibris and WP2 leaders

Date	2014-03-31

Description	ExLibris and WP2 leaders will evaluate if the progress of the development of the infrastructure is according to the plan as defined by the contract ExLibris-Baierer. End of contract ExLibris-Baierer.
Responsible	Konstantin Baierer, Axel Kaschte, Stefan Gradmann, Chris Bizer

MS-12: Release of OmNom v1.1

Date	2015-02-01
Description	–
Responsible	MAN, NTUA, MPIGW, SBB

7 Conclusion

This report tried to conceptualize how manuscript (meta-)data from the various data providers to Europeana via DM2E is possible. From a general problematization of the different steps and the requirements for the architecture of a software allowing the automatization of these steps, we went to practical examples of tools and libraries that are more or less ready-made for doin these steps. The section about the OmNom prototype showed that it is feasible to implement an approximation of the final system in less than three weeks and that the preliminary specificatin of the architecture, while still in flux, is evolving in the right direction.

The final chapter turned the various bits and pieces of information, requirements and tools to use into tasks that are going to be done by people until fixed dates.

However, this is far from a definitve plan, leave alone the directly implementable specification for a software system. OmNom might just be glue code but it must handle such a mutlitude of cases and interact with such a large number of systems and backends that no single person can create such specifications from scratch without making mistakes.

Therefore, the author hopes that this concept is the foundation for discussion and that it can be improved by further discussions with ExLibris, the DM2E project coordinator, the WP2 leader and the Architecture Task Force to create a set of specifications and thereupon a framework that is robust, versatile and user-friendly.