ICT-PSP Project no. 270905

LINKED HERITAGE

Coordination of standard and technologies

for the enrichment of Europeana

Starting date: 1st April 2011

Ending date: 30th September 2013

Deliverable Number:	D 5.2
Title of the Deliverable:	Documented APIs
Dissemination Level:	Public

Contractual Date of Delivery to EC:	Month 18
Actual Date of Delivery to EC:	October 2012

Project Co-ordinator

Company name : Istituto Centrale per il Catalogo Unico (ICCU)
Name of representative : Rosa Caffo
Address : Viale Castro Pretorio 105, I-00185 Roma
Phone number : +39.06.49210427
Fax number : +39.06. 06 4959302
E-mail : rcaffo@beniculturali.it
Project WEB site address : http://www.linkedheritage.org

Context

WP 5	Technical Integration
WP Leader	NTUA
Task 5.*	
Task Leader	NTUA
Dependencies	

Author(s)	Nasos Drosopoulos, Kostas Pardalis, Kostas Sismanis
Contributor(s)	
Reviewers	Andrea Tempera (ICCU)
Approved by:	

History

Version	Date	Author	Comments
0.1	21/09/2012	Kostas Pardalis	Draft
0.2	11/10/2012	Nasos Drosopoulos	For internal review
0.3	18/10/2012	Nasos Drosopoulos	Final

Table of Contents

EXECUTIVE SUMMARY

This report documents a set of application programming interfaces offering a simple and structured way to access the aggregation functionalities and repository capabilities of the Linked Heritage technology platform. A set of read access interfaces allows external systems to programmatically search and retrieve user and organization profiles, mapping files, reports and other data from the system. Write capabilities enable external systems to dynamically add new content as well as perform operations on existing datasets. The architecture design is based on predominant models such as REST and, standard wire-level protocols such as the AMPQ, in an effort to adopt existing tools and technologies that facilitate interoperability with external systems. Data exchange is following open standards such as XML, JSON, RSS and the Atom Publishing Protocol to allow developers to get up to speed quickly and reuse existing solutions. The APIs have been used in Task 5.2 to setup the authentication of the Terminology Management Platform (TMP) developed in WP3, allowing its integration with the technology platform, to establish the publication through the current Europeana harvesting mechanism and, in an experimental integration of the whole MINT platform with the newly introduced Europeana United Ingestion Manager (UIM).

BACKGROUND

The Linked Heritage Technology Platform provides a user friendly web environment that allows for the ingestion and presentation of all relevant and statistical information concerning content provider's metadata together with an intuitive mapping service that illustrates the LIDO metadata model. It offers all the functionality and documentation required for the providers to define their crosswalks, in formal transformations that are editable, reusable and can be applied incrementally to user input. Throughout all steps, examples, previews and visual indications illustrate and guide user actions. One of the key capabilities lies in the ability to semantically enhance user metadata through conditional mapping of input elements using transformation functions that allows for the addition and enrichment of semantics even when those are not specifically stated in the input data.

The tool also determines the operational work flow processes which bring the amalgamated content of the partner institutions into Europeana and defines, manages and executes, with the European Digital Library Office, the implementation plan (WP6 - T6.1) to ensure that the content is visible in Europeana. The platform supports the export of the aggregated metadata to several established standards concerning presentation and archive management. The primary effort is directed to the transformation of the aggregated content to the Europeana Semantic Elements (ESE) schema and the Europeana Data Model (EDM), and to the maintenance of an OAI-PMH repository to facilitate harvesting by Europeana.

The ingestion tool is based on the MINT metadata aggregation platform that was developed and is maintained by the leader of the WP, IVM Laboratory of the National Technical University of Athens. Its design is based on a specialisation of a general metadata ingestion work-flow, as it is described in detail in D5.1. The Metadata Gateway (D5.3) offers remediation capabilities for metadata in the target schema as well as Europeana's, used for the online publication of records, enrichment through external services and Linked Data experiments (WP2).

APPROACH

MINT is a platform that offers functionalities to the end-user to support data and metadata interoperability. The main mechanism for exposing these data to other services, e.g. Europeana, is through the OAI-PMH protocol. This protocol is designed for interoperating mainly on the metadata level by exposing a pre-defined set of HTTP verbs that can be used between two arbitrary systems for exchanging data. In many

cases though, the need for also exposing core services of the MINT platform arises, e.g. to provide external services with the functionality of adding data, manage users and so on. This level of interoperation is achieved by the exposure of two developer APIs, one being based on the REST design principles and the second one based on a message queue architecture.

The REST API exists mainly for exposing data that are stored in MINT to external services, without providing extensive mechanisms for altering these data or adding new. On the contrary, the second API provides all the functionalities for creating, updating and deleting entities inside the database of the MINT platform.

By making this distinction between the two APIs, it is possible to create one lightweight API based on an as neutral as possible architecture like the one offered by REST and HTTP, which also supports the connection with different MINT instances at the same time. The lack of functionalities related to updating and deleting, allows the API to connect to any MINT database, without introducing procedures that could create inconsistencies to the running instance.

The second API offers to the external service all the possible functionalities that are implemented as part of the MINT platform, but based on its architecture it is possible to take care of all the CRUD operations without introducing inconsistencies. The penalty of this is the additional complexity of the overall architecture and the various components that have to be installed apart from MINT in order to expose this API, e.g. the external Message queue.

MINT DATABASE REST API

Introduction

REST (REpresentation State Transfer) describes an architectural style of networked systems such as Web applications. It was first introduced in 2000 in a Ph.D dissertation by Roy Fielding, one of the principal authors of the HTTP specification. REST refers to a collection of architecture constraints and principles. An application or design, if it meets those constraints and principles, is RESTful. REST has emerged in the last few years alone as a predominant Web service design model.

REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that is uniquely addressable using a URI (Universal Resource Identifier). It is an item of interest, a conceptual identity that is exposed to the clients. Example resources include application objects, database records, algorithms, and so on. A representation of a resource is typically a document that captures the current or intended state of a resource. All resources share a uniform interface for the transfer of state between client and server. Standard HTTP methods such as GET, PUT, POST, and DELETE are used.

One of the most important REST principles for Web applications is that the interaction between the client and server is stateless between requests. Each request from the client to the server must contain all of the information necessary to understand the request. The client wouldn't notice if the server were to be restarted at any point between the requests. Additionally, stateless requests are free to be answered by any available server, which is appropriate for an environment such as cloud computing. The client can cache the data to improve performance[1].

Another important REST principle is the layered system, which means a component cannot see beyond the immediate layer with which it is interacting. By restricting knowledge of the system to a single layer, a boundary is placed on the overall system complexity, promoting substrate independence. REST architectural constraints, when applied as a whole, generate an application that scales well to a large number of clients. It also reduces interaction latency between clients and servers. The uniform interface simplifies the overall system architecture and improves the visibility of the interactions between subsystems. REST simplifies implementation for both the client and server.

REST's client–server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. REST lets intermediate processing by constraining messages be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability[2].

RESTful web services

A RESTful web service (also called a RESTful web API) is a web service implemented using HTTP and the principles of REST. It is a collection of resources, with four defined aspects:

- the base URI for the web service,
- the Internet media type of the data supported by the web service,

1 *A multi-tier architecture for building RESTful Web services,* Bruce Sun, 2009

2 *Architectural Styles and the Design of Network-based Software Architectures*, Fielding, Roy Thomas, UoC, Irvine, 2000

- the set of operations supported by the web service using HTTP methods,
- the hypertext driven API.

A concrete implementation of a REST Web service follows four basic design principles:

- Use HTTP methods explicitly.
- Be stateless.
- Expose directory structure-like URIs.
- Transfer XML, JavaScript Object Notation (JSON), or both.

The explicit use of HTTP methods in a way that is consistent with the protocol definition (RFC 2616) is one of the key characteristics of a RESTful Web service. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

- To create a resource on the server, use POST.
- To retrieve a resource, use GET.
- To change the state of a resource or to update it, use PUT.
- To remove or delete a resource, use DELETE.

MINT Database Overview

Following is an enumeration of MINT's database schema elements that correspond to resources in the context of the REST API. Descriptive properties and relations between resources are listed in parentheses.

Database

- Actors
  - Organizations (Name, Country, Users,etc.)
  - Users(Name, email, role, etc.)
- Subjects
  - Uploads(Database ID, size ,number of items, user, organization, etc.)
  - Mappings(Database ID, size ,type of schema, user, organization, etc.)
  - Transformations(Database ID, size ,number of items, user, organization, etc.)
  - Publications(Database ID, size ,number of items, user, organization, etc.)

MINT API INDEX

MINT's REST API provides access to resources (data entities) via URI paths. To use the API, an application will make an HTTP request and parse the response. The MINT REST API uses JSON as its communication format, and the standard HTTP methods GET and POST.

The following index documents the current version of the REST API, listing the resources and methods it exposes and, the appropriate URL structure (for a hypothetical MINT API setup at http://api.mint-projects.eu) along with request query parameters where applicable. Every string passed to and from the MINT API needs to be UTF-8 encoded.

databases

Gets a list in JSON format of the databases – projects running in the server.

URL structure

http://api.mint-projects.eu/servlets/databases

dbaseorgs

Gets a list in JSON format of the organizations participating in the project.

Parameters

- database

URL structure

http://api.mint-projects.eu/servlets/dbaseorgs?database=mint

orgusers

Gets a list in JSON format of the users-members of an organization, or of the project database, depending on the parameters.

Parameters

- database (mandatory)
- organization

URL structure

http://api.mint-projects.eu/servlets/orgusers?database=mint

http://api.mint-projects.eu/servlets/orgusers?database=mint&organization=NTUA

uploads

Gets a list in JSON format of the uploads made in a project database , or by an organization, or by a user.

Parameters

- database (mandatory)
- organization
- user

URL structure

http://api.mint-projects.eu/servlets/uploads?database=mint

http://api.mint-projects.eu/servlets/uploads?database=mint&organization=NTUA&user=admin

mappings

Gets a list in JSON format of the mappings made in a project database , or by an organization.

Parameters

- database (mandatory)
- organization

URL structure

http://api.mint-projects.eu/servlets/mappings?database=mint&organization=NTUA

publications

Gets a list in JSON format of the publications made in a project database , or by an organization, or by a user.

Parameters

- database (mandatory)
- organization
- user

URL structure

http://api.mint-projects.eu/servlets/publications?database=mint

http://api.mint-projects.eu/servlets/publications?database=mint&organization=NTUA&user=admin

transformations

Gets a list in JSON format of the transformations made in a project database , or by an organization, or by a user.

Parameters

- database (mandatory)
- organization
- user

URL structure

http://api.mint-projects.eu/servlets/transformations?database=mint

http://api.mint-projects.eu/servlets/transformations?database=mint&organization=NTUA&user=admin

publication

Gets the publication identified by the publicationId in the parameter in a zip file.

Parameters

- database (mandatory)
- application id (mandatory)

URL structure

http://api.mint-projects.eu/servlets/publication?database=mint&publicationId=1001

getransformation

Gets the transformation identified by the transformationId in the parameter .

Parameters

- database (mandatory)
- transformationId (mandatory)

URL structure

http://api.mint-projects.eu/servlets/getransformation?database=mint&transformationId=1

getupload

Gets the upload identified by the transformationId in the parameter .

Parameters

database (mandatory)

uploadId (mandatory)

URL structure

http://api.mint-projects.eu/servlets/getupload?database=mint&uploadId=1

getmapping

Gets the mapping identified by the mappingId in the parameter .

Parameters

database (mandatory)

mappingId (mandatory)

URL structure

http://api.mint-projects.eu/servlets/getmapping?database=mint&mappingId=1

authenticateuser

Authenticate a user to the database server.

Parameters

- database (mandatory)
- login (mandatory)
- password (mandatory)

URL structure

http://api.mint-projects.eu/servlets/authenticateuser&user=user&password=pass

adduser

Add a user to the database .

Parameters

- database (mandatory)
- login (mandatory)
- password (mandatory)
- firstname
- lastname
- organization

URL structure

curl -d "database=mint&login=user&password=pass&organization=NTUA" http://api.mint-projects.eu/servlets/adduser

addDatabase

Adds a database to the server configuration of known databases.

Parameters

- project (mandatory)
- user (mandatory)
- password (mandatory)
- url (mandatory)

URL structure

http://api.mint-projects.eu/servlets/addDatabase&project=roject&user=user&password=pass&url=database.localhost:port

MESSAGE QUEUE BASED API

The message Queue based API is built using RabbitMQ[3] as the MQ implementation of choice and a number of message queue design principles. The API supports two main way of interacting with MINT, the first is through a Remote Procedure Call (RPC) like procedure, while the second is based on the Work Queue framework. RPC calls to MINT ensure that the procedure is blocked until the operation is finished, while the work queues pattern allows queuing of an operation on MINT without blocking the external service that made the call. In any case all the functionalities are exposed through both patterns and it is also possible to use a combination of both. For example the creation of a user could be an RPC like call, while the import of a new dataset which usually takes time, could be a work Queue based call to MINT.

MINT PI is part of the metadata interoperability services suite, offering a scalable mechanism for structured data processing. It is built using RabbitMQ acting as a message broker in its core and utilizes a number of message queue patterns based on the AMQP[4], a standard wire-level protocol and semantic framework for high performance enterprise messaging. AMQP is an Open Standard for Messaging Middleware.

By complying with the AMQP standard, middleware products written for different platforms and in different languages can send messages to one another. AMQP addresses the problem of transporting value-bearing messages across and between organisations in a timely manner. The approach of using message queues instead of a distributed parallel processing mechanism like Hadoop[5], was decided based on the requirement to scale on both large and small datasets without reducing the efficiency of the overall system.

RPC Queue implementation

Work queues can be used to distribute time consuming tasks among multiple workers without blocking the client who submitted these jobs. There are cases though, where it is mandatory to block the calling method until the operation ends and get back a result, for example when we create a user and we need to wait until the operation ends and the ID is returned to the calling client. For this reason an RPC pattern can be used, in the current case this pattern is implemented on top of RabbitMQ which results to the creation of a scalable RPC system.

In general doing RPC over RabbitMQ is straightforward. A client sends a request message and a server replies with a response message. In order to receive a response the client needs to send a 'callback' queue address with the request. Creating one callback queue for every RPC request is inefficient and for this reason it is preferred to create one callback queue for each client instead. That raises a new issue, having received a response in that queue it's not clear to which request the response belongs. That's when the correlation_id property is used, being set to a unique value for every request. Later, when we receive a message in the callback queue we'll look at this property, and based on that we'll be able to match a response with a request. If we see an unknown correlation_id value, we may safely discard the message - it doesn't belong to our requests. The overall approach is depicted in Figure 1.

3 http://www.rabbitmq.com/

4 http://amqp.org/

5 http://hadoop.apache.org/

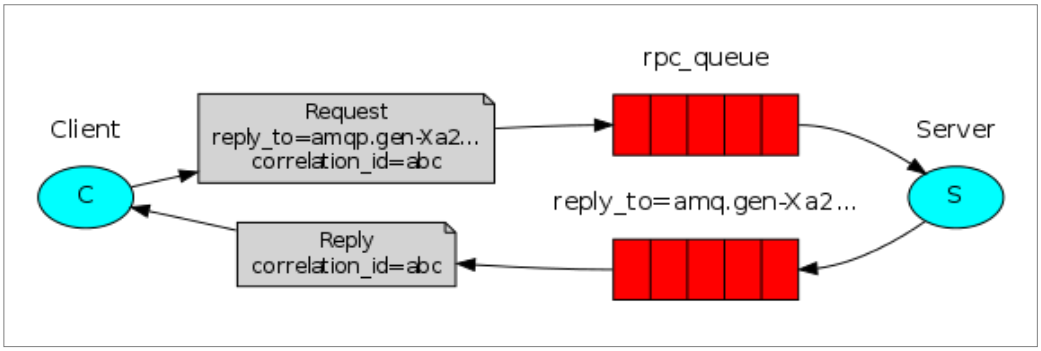

Figure 1. The RPC Queue pattern

Work Queue implementation

The main idea behind Work Queues is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead, it is possible to schedule the task to be performed at a later stage. To do that, we need to encapsulate a task as a message and send it to a queue. A worker process running in the background will pop the tasks and eventually execute the job. When you run many workers the tasks will be shared between them. This concept is especially useful in web applications where it's impossible to handle a complex task during a short HTTP request window. In the case of MINT, a number of processing intensive operations are exposed through a working queue, e.g. the transformation of a dataset. Working queues is a very straight forward and easy queue pattern as it is also depicted in Figure 2.

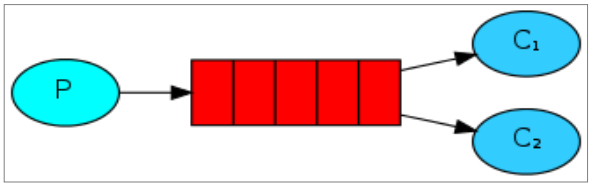

Figure 2. Working Queue pattern

One of the main advantages of using Working Queues is the ability to easily parallelize work. At any point it is possible to add more workers and scale easily. In order to make sure a message is never lost, RabbitMQ supports message acknowledgments. An ack(nowledgement) is sent back from the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it. If a consumer dies without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die. There aren't any message timeouts; RabbitMQ will redeliver the message only when the worker connection dies. This allows for the handling of messages where execution takes substantially long time, which is the case for a number of processing tasks in the context of MINT.

Common characteristics between RPC and Work Queues implementations

There are also a number of desired characteristics of Message Queues that are shared between the two proposed implementations; these are the following:

- **Message Durability**: It's the property of the message queue of ensuring that a message is not lost in the case were the actual Queue crashes. This is achieved by persisting in specific time intervals the delivered to the queue messages on disk.
- **Fair Dispatch**: A number of QoS strategies are supported by RabbitMQ to ensure a fair work dispatch on available workers.

Both queue implementations are based on the Strategy or Policy Pattern in order to ensure the extensibility of the exposed functionalities without adding more complexity to the API implementation. In this way a developer is able to extend the API without having to know the intrinsic details of the MQ implementation. The pattern is presented in a UML diagram in Figure 3.

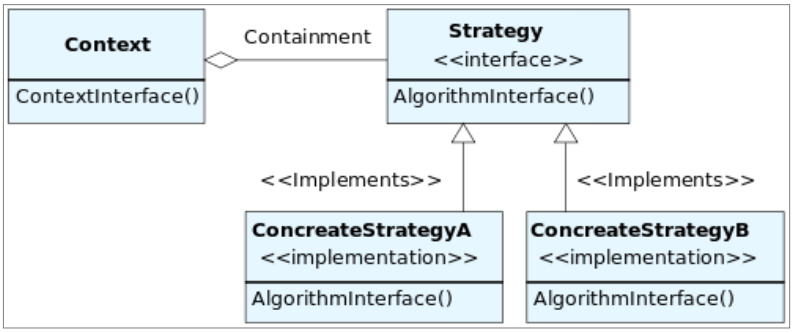

Figure 3. The strategy or policy pattern

Command Messages Index

In both cases, the queues are used as an interoperable and scalable mechanism for exchanging messages that represent commands to be executed in the context of MINT. These messages are defined in XML using the XML Schema presented in Appendix A, defining all the implemented commands together with the needed parameters for their execution, the expected results and any error messages. Based on this XML Schema the following commands are supported by the MINT interoperability API using any of the two available Queue configurations.

- **CreateUser**: This message is created whenever the external service wants to create a new user in the context of MINT.
  - **Parameters**

Name	Type	Is Mandatory
username	String	True
password	String	True
First name	String	False
Last name	String	False
Email	String	False
Phone	String	False

Organization	String	False
jobRole	String	False
systemRole	string	false

- **Return Value:** the id of the newly created user
- **Error:** if an error has occurred, then the system will respond with an error message containing all the generated exceptions.

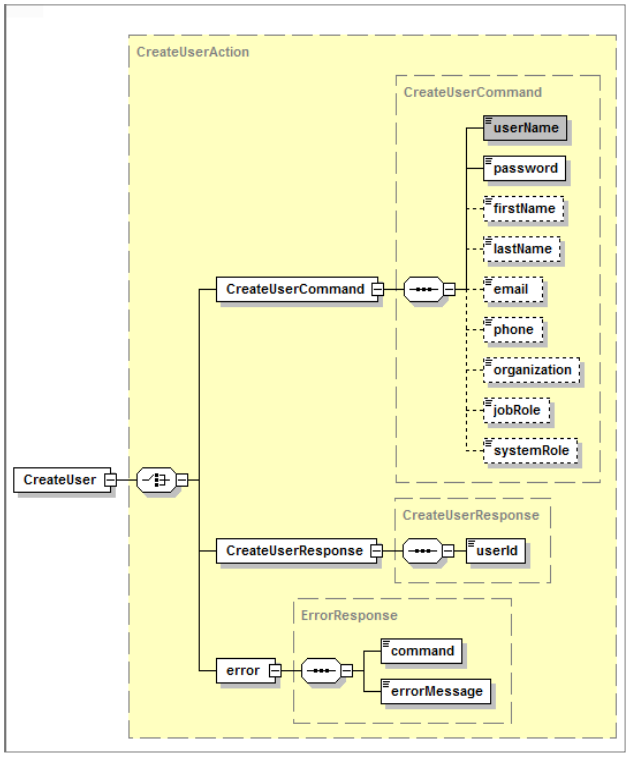

Figure 4. The CreateUser Command structure

- **CreateOrganization**: This message is created whenever the external service wishes to create a new organization in the MINT application context.
  - **Parameters**

Name	Type	Is Mandatory
Name	String	True
userId	String	True
englishName	String	True
Type	String	True
country	String	true

  - **Return Value**: The id of the newly created organization

- **Error**: if an error has occurred, then the system will respond with an error message containing all the generated exceptions.

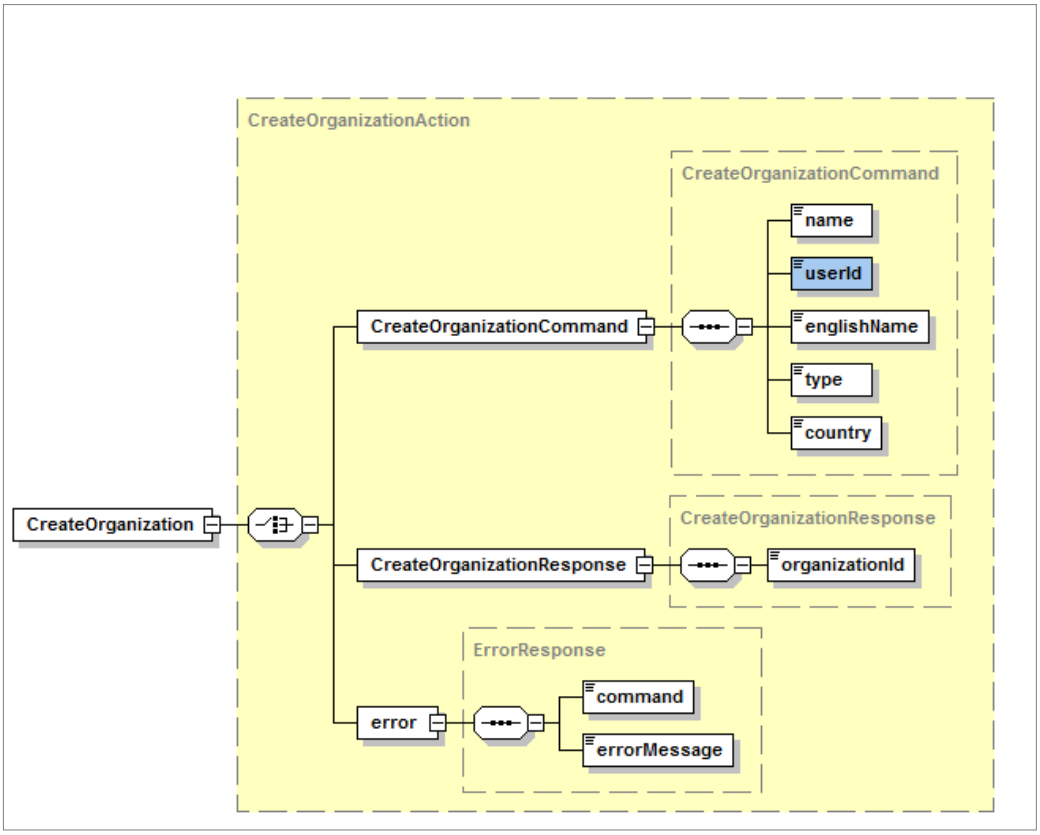

Figure 5. The CreateOrganization Command structure

- **UserExists:** This message is created every time a command is executed in order to check if a particular user exists in MINT or not.
  - **Parameters**: The only parameter of this command is the id of a user to be checked if it exists or not.
  - **Return Value:** true or false depending on the existence of the user id in the MINT application context.
  - **Error:** if an error has occurred, then the system will respond with an error message containing all the generated exceptions.

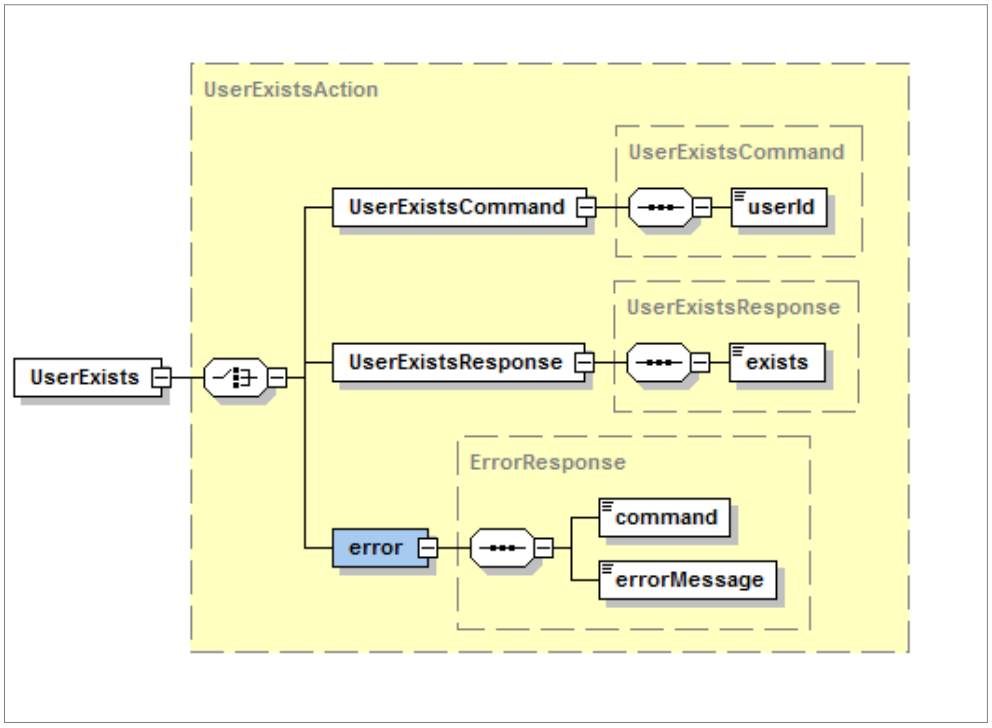

Figure 6. The UserExists command structure

- **OrganizationExists**: This message is created every time a command is executed in order to check if a particular organization exists in MINT or not.
  - **Parameters**: The only parameter of this command is the id of an organization to be checked if it exists or not.
  - **Return Value**: true or false depending on the existence of the organization id in the MINT application context.
  - **Error**: if an error has occurred, then the system will respond with an error message containing all the generated exceptions.

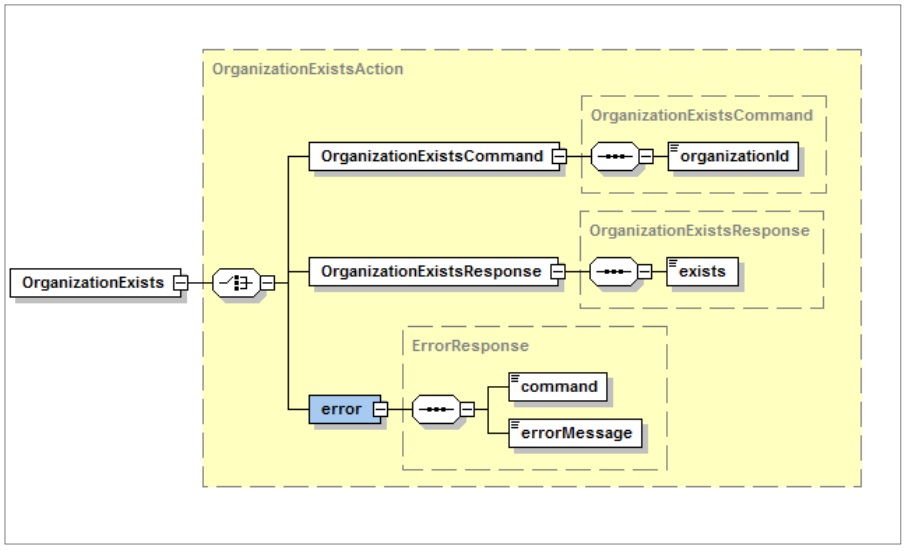

Figure 7. The OrganizationExists command structure

- **ImportExists**: This message is created every time a command is executed in order to check if a particular import exists in MINT or not.
    - **Parameters**: The only parameter of this command is the id of an import to be checked if it exists or not.
    - **Return Value**: true or false depending on the existence of the import id in the MINT application context.
    - **Error**: if an error has occurred, then the system will respond with an error message containing all the generated exceptions.

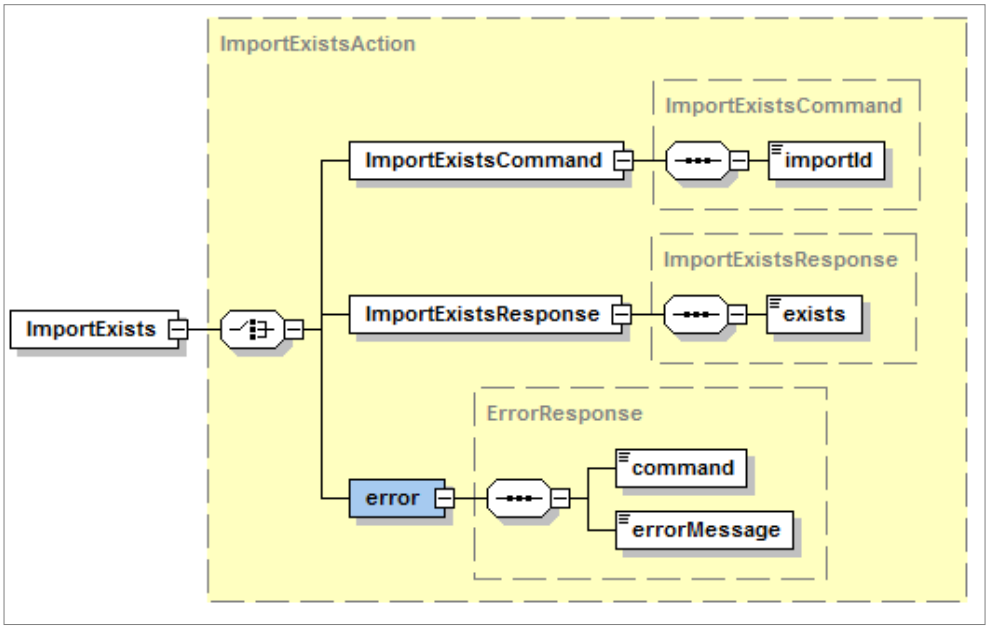

Figure 8. the Import Exists command structure

- **CreateImport**: This message is created every time a command is executed in order to create a new import in the MINT application context.
    - **Parameters:** The command requires the userID of the user who initiates the import together with the organizationID of the user's organization. Finally a url is given for a JDBC connection to a database, e.g. from the Repox DB, in order to fetch and import the data into MINT.
    - **Return Value:** The id of the newly created import.
    - **Error:** if an error has occurred, then the system will respond with an error message containing all the generated exceptions.

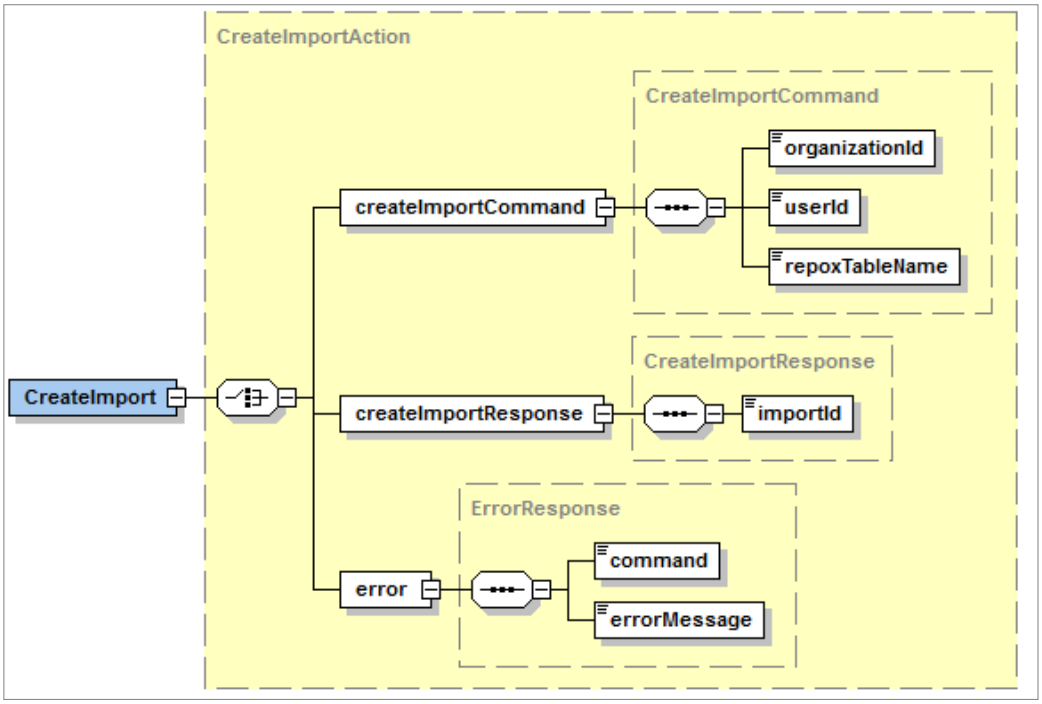

Figure 9. The Create Import command structure

- **PublishTransformation**: This message is created every time a command is executed in order to publish a transformed dataset from MINT, this process also includes any further transformation steps that are defined as part of the overall aggregation workflow.
  - **Parameters:** none
  - **Return value:** The id of the transformed dataset that is published together with a URL from where an archive of the published dataset can be retrieved.
  - **Error**: if an error has occurred, then the system will respond with an error message containing all the generated exceptions.

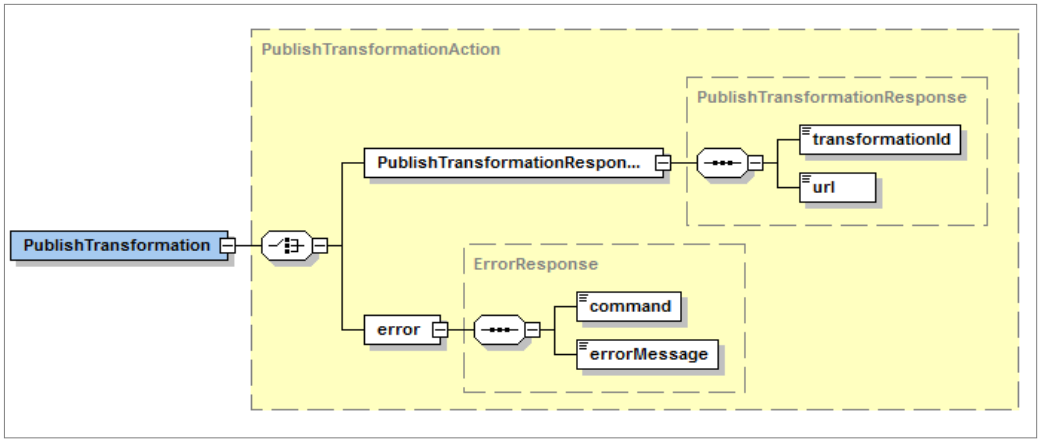

Figure 10. The Publish Transformation command structure

- **GetImports**: This message is created every time a command is executed in order to fetch a list of Imports for a specific organization that exists in the application context of MINT.

- **Parameters:** The command requires the ID of the organization for which a list of import IDs will be created and delivered.
- **Return Value**: A list of import IDs.
- **Error**: if an error has occurred, then the system will respond with an error message containing all the generated exceptions.

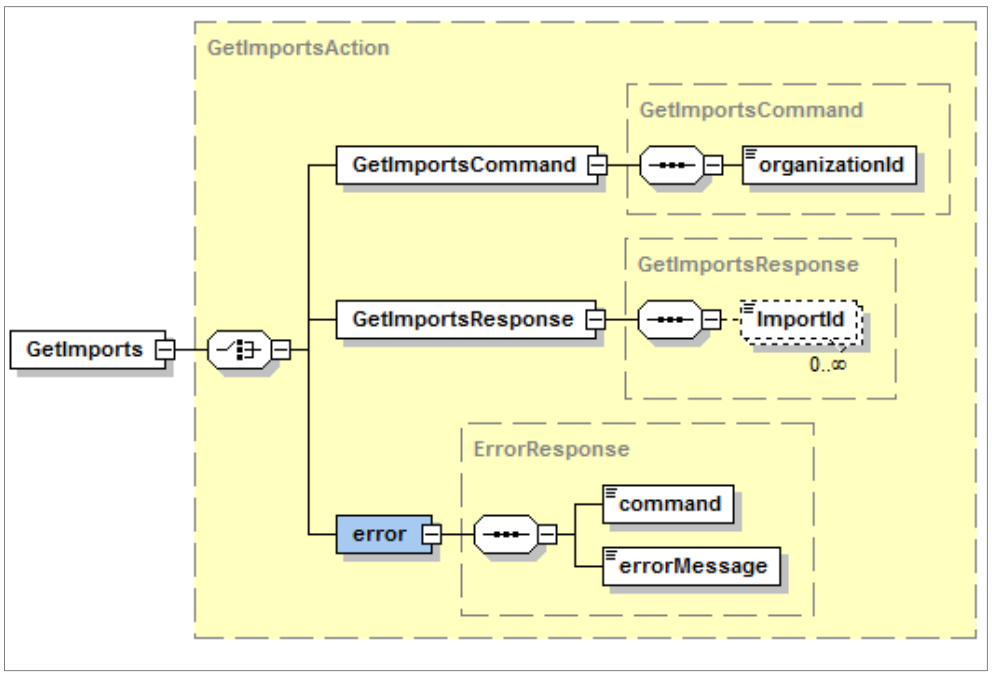

Figure 11. The GetImports command structure

XML AND RDF METADATA GATEWAY

The Linked Heritage Metadata Gateway, documented in deliverable D5.3, also constitutes an API for the remediation of XML and RDF records. The Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH), is a low barrier mechanism for repository interoperability. Data providers setup repositories that expose structured metadata, to which service providers make OAI-PMH service requests to harvest that metadata. The protocol consists of a set of six verbs or services that are invoked within HTTP. In the context of an aggregation, OAI-PMH provides a mechanism for technical interoperability between the ingestion platform and other modules or platforms (e.g. Europeana United Ingestion Mechanism).

The Linked Heritage Metadata Gateway is capable of managing heterogeneous collections of metadata records while exposing services for mapping and transforming from one metadata schema to another. In order to extend the functionalities of the OAI-PMH protocol and thus to expose metadata through an interoperable mechanism, MINT implements the defined OAI-PMH verbs on top of the underlying, domain-specific data layer. An issue that arises in the case of aggregations is that while being able to manage collections of metadata records, the OAI-PMH verbs operate on an item level, something that makes the implementation of the appropriate verbs, directly on top of a collection-based data layer a challenging task. For this reason, and also to design and introduce a set of robust, enhanced metadata processing services, an export mechanism is introduced in the MINT platform, facilitating scalable and reliable data delivery and exchange between different data layers and repositories.

The overall architecture for the delivery of content to Europeana consists of a data processing layer responsible for iterating the records that are stored inside the Linked Heritage Platform repository, transforming and manipulating them, together with a data layer which is exposed to Europeana by using an implementation of the OAI-PMH protocol. The overall architecture is depicted in Figure 12 and further documentation and implementation details can be found in D5.3.

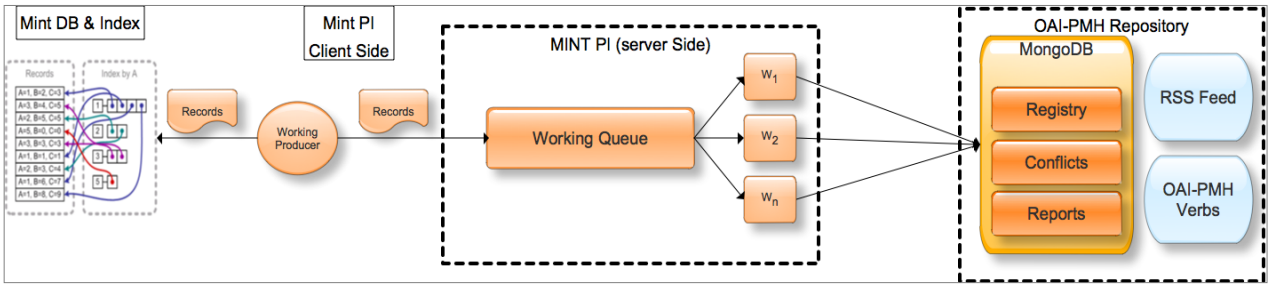

Figure 12. Data Delivery Architecture

The Gateway also allows for the RDFization of LIDO records and the production of enriched and linked resources that can be used for the project's Linked Data experiments. RDF enables providers to publish various types of data on the Web in a unified format. This data intends to be machine-interpretable so that web developers can access it by programming languages in order to create applications for i.e., crawling, aggregating, summarizing, or highlighting contained information. While publishing RDF data on the web is vital to the growth of the Semantic Web, using the information to improve the collective utility of the Web is the true goal of the Semantic Web. To accomplish this goal, Linked Data principles define a set of standardized interfaces for working with RDF data in a web-based programming environment. A set of appropriate repositories (triple-stores) have been tested and deployed for WP2 Linked Data experiments, offering SPARQL-endpoints as well as developer APIs (more in D5.3).

CONCLUSION

Deliverable D5.2 presents the architecture and technical specifications for MINT's APIs. A set of standardized interfaces offer a simple and structured way to access the functionalities of the Linked Heritage technology platform and the Metadata Gateway's repository capabilities. The architecture design is based on predominant models such as REST and, standard wire-level protocols such as the AMPQ, while data exchange follows open standards including XML, JSON, RSS and the Atom Publishing Protocol. The developer APIs have been used for the integration of the Terminology Management Platform (TMP) developed in WP3, the online publication of Linked Heritage content through Europeana and, in an experimental integration of the whole MINT platform with the newly introduced Europeana United Ingestion Manager (UIM).

APPENDIX A

Definitions of terms and abbreviations

Reader is assumed to have basic understanding of XML, XSLT, RDF and the MINT metadata aggregation platform. Following is a glossary of technical terms and abbreviations used in the document.

AMQP (Advanced Message Queuing Protocol) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

APACHE LUCENE is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.

ATOM PUBLISHING PROTOCOL is a simple HTTP-based protocol for creating and updating web resources.

ATOM SYNDICATION FORMAT is an XML language used for web feeds.

CRUD In computer programming, create, read, update and delete (CRUD) are the four basic functions of persistent storage.

HADOOP is an open source project from the Apache Software Foundation that provides a software framework for distributing and running applications on clusters of servers. It is inspired by Google's MapReduce programming model as well as its file system.

JSON (JavaScript Object Notation) is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects.

MESSAGE BROKER is either a complete messaging system or software that works with existing messaging transports in order to add routing intelligence and data conversion capabilities. A rules engine analyzes the messages and determines which application should receive them, and a formatting engine converts the data into the structure required by the receiving application.

MESSAGING MIDDLEWARE refers to software that provides an interface between applications, allowing them to send data back and forth to each other asynchronously.

MONGODB is a scalable, high-performance, open source NoSQL database

OAI-PMH (Open Archives Initiative Protocol for Metadata Harvesting) is a protocol developed by the Open Archives Initiative. It is used to harvest (or collect) the metadata descriptions of the records in an archive so that services can be built using metadata from many archives. OAI-PMH uses XML over HTTP.

OAICAT open source software project is a Java Servlet web application providing a repository framework that conforms to the OAI-PMH v2.0. This framework can be customized to work with arbitrary data repositories by implementing some Java interfaces.

REST (REpresentational State Transfer) is a style of software architecture for distributed systems such as the World Wide Web.

RPC (remote procedure call) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

RSS (originally RDF Site Summary, often dubbed Really Simple Syndication) is a family of web feed formats used to publish frequently updated works—such as blog entries, news headlines, audio, and video—in a standardized format.

An RSS document (which is called a "feed", "web feed", or "channel") includes full or summarized text, plus metadata such as publishing dates and authorship.

SPARQL ENDPOINT is a conformant SPARQL protocol service as defined in the SPROT specification. A SPARQL endpoint enables users (human or other) to query a knowledge base via the SPARQL language.

STRATEGY PATTERN (also known as the policy pattern) is a particular software design pattern, whereby algorithms can be selected at runtime. Formally speaking, the strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

TRIPLESTORE is a purpose-built database for the storage and retrieval of triples, a triple being a data entity composed of subject-predicate-object. Much like a relational database, one stores information in a triplestore and retrieves it via a query language. In addition to queries, triples can usually be imported/exported using Resource Description Framework (RDF) and other formats

WIRE-LEVEL PROTOCOL is a low-level interface (wire level being the hardware level; actual wires, circuits, etc.). It typically refers to programming interfaces (APIs) in a network directly above the physical layer that are used strictly for transport or interconnection.

WORK QUEUE is a framework for building large master-worker applications that span many computers including clusters, clouds, and grids.

APPENDIX B

MINT API command messages XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
	<xs:complexType name="PublishTransformationAction">
		<xs:choice>
			<xs:element name="PublishTransformationResponse"
type="PublishTransformationResponse"/>
			<xs:element name="error" type="ErrorResponse"/>
		</xs:choice>
	</xs:complexType>
	<xs:element name="PublishTransformation" type="PublishTransformationAction"/>
	<xs:complexType name="PublishTransformationResponse">
		<xs:sequence>
			<xs:element name="transformationId" type="xs:string"/>
			<xs:element name="url" type="xs:string"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="CreateImportResponse">
		<xs:sequence>
			<xs:element name="importId" type="xs:string"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="CreateImportCommand">
		<xs:sequence>
			<xs:element name="organizationId" type="xs:string"/>
			<xs:element name="userId" type="xs:string"/>
			<xs:element name="repoxTableName" type="xs:string"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="PublicationResponse">
		<xs:sequence>
			<xs:element name="url" type="xs:anyURI"/>
			<xs:element name="includedImport" maxOccurs="unbounded">
```

```
                        <xs:complexType>
                            <xs:simpleContent>
                                <xs:extension base="xs:boolean">
                                    <xs:attribute name="importId"/>
                                </xs:extension>
                            </xs:simpleContent>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="PublicationCommand">
                <xs:sequence>
                    <xs:element name="organizationId" type="xs:string"/>
                    <xs:element name="userId" type="xs:string"/>
                    <xs:element name="includedImports">
                        <xs:complexType>
                            <xs:sequence>
                                <xs:element name="importId" type="xs:string"
maxOccurs="unbounded"/>
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="PublicationAction">
                <xs:choice>
                    <xs:element name="PublicationCommand"
type="PublicationCommand"/>
                    <xs:element name="PublicationResponse"
type="PublicationResponse"/>
                    <xs:element name="error" type="ErrorResponse"/>
                </xs:choice>
            </xs:complexType>
            <xs:complexType name="CreateImportAction">
                <xs:choice>
                    <xs:element name="createImportCommand"
```

```
type="CreateImportCommand"/>
                        <xs:element name="createImportResponse"
type="CreateImportResponse"/>
                        <xs:element name="error" type="ErrorResponse"/>
                </xs:choice>
        </xs:complexType>
        <xs:complexType name="CreateOrganizationAction">
                <xs:choice>
                        <xs:element name="CreateOrganizationCommand"
type="CreateOrganizationCommand"/>
                        <xs:element name="CreateOrganizationResponse"
type="CreateOrganizationResponse"/>
                        <xs:element name="error" type="ErrorResponse"/>
                </xs:choice>
        </xs:complexType>
        <xs:complexType name="CreateUserCommand">
                <xs:sequence>
                        <xs:element name="userName" type="xs:string"/>
                        <xs:element name="password" type="xs:string"/>
                        <xs:element name="firstName" type="xs:string" minOccurs="0"/>
                        <xs:element name="lastName" type="xs:string" minOccurs="0"/>
                        <xs:element name="email" type="xs:string" minOccurs="0"/>
                        <xs:element name="phone" type="xs:string" minOccurs="0"/>
                        <xs:element name="organization" type="xs:string" minOccurs="0"/>
                        <xs:element name="jobRole" type="xs:string" minOccurs="0"/>
                        <xs:element name="systemRole" minOccurs="0">
                                <xs:simpleType>
                                        <xs:restriction base="xs:string">
                                                <xs:enumeration value="publisher"/>
                                                <xs:enumeration value="annotator"/>
                                        </xs:restriction>
                                </xs:simpleType>
                        </xs:element>
                </xs:sequence>
        </xs:complexType>
        <xs:element name="CreateImport" type="CreateImportAction"/>
```

```
        <xs:element name="CreateOrganization" type="CreateOrganizationAction"/>
        <xs:element name="CreateUser" type="CreateUserAction"/>
        <xs:complexType name="ErrorResponse">
                <xs:sequence>
                        <xs:element name="command" type="xs:string"/>
                        <xs:element name="errorMessage" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
        <xs:element name="ImportExists" type="ImportExistsAction"/>
        <xs:complexType name="ImportExistsResponse">
                <xs:sequence>
                        <xs:element name="exists" type="xs:boolean"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="ImportExistsCommand">
                <xs:sequence>
                        <xs:element name="importId" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="ImportExistsAction">
                <xs:choice>
                        <xs:element name="ImportExistsCommand"
type="ImportExistsCommand"/>
                        <xs:element name="ImportExistsResponse"
type="ImportExistsResponse"/>
                        <xs:element name="error" type="ErrorResponse"/>
                </xs:choice>
        </xs:complexType>
        <xs:element name="OrganizationExists" type="OrganizationExistsAction"/>
        <xs:complexType name="OrganizationExistsAction">
                <xs:choice>
                        <xs:element name="OrganizationExistsCommand"
type="OrganizationExistsCommand"/>
                        <xs:element name="OrganizationExistsResponse"
type="OrganizationExistsResponse"/>
                        <xs:element name="error" type="ErrorResponse"/>
```

```
            </xs:choice>
        </xs:complexType>
        <xs:complexType name="OrganizationExistsResponse">
            <xs:sequence>
                <xs:element name="exists" type="xs:boolean"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="OrganizationExistsCommand">
            <xs:sequence>
                <xs:element name="organizationId" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
        <xs:element name="UserExists" type="UserExistsAction"/>
        <xs:complexType name="UserExistsAction">
            <xs:choice>
                <xs:element name="UserExistsCommand"
type="UserExistsCommand"/>
                <xs:element name="UserExistsResponse"
type="UserExistsResponse"/>
                <xs:element name="error" type="ErrorResponse"/>
            </xs:choice>
        </xs:complexType>
        <xs:complexType name="UserExistsResponse">
            <xs:sequence>
                <xs:element name="exists" type="xs:boolean"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="UserExistsCommand">
            <xs:sequence>
                <xs:element name="userId" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
        <xs:element name="GetImports" type="GetImportsAction"/>
        <xs:element name="GetTransformations" type="GetTransformationsAction"/>
        <xs:element name="Publication" type="PublicationAction"/>
        <xs:complexType name="CreateUserResponse">
```

```
            <xs:sequence>
                <xs:element name="userId" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="CreateOrganizationResponse">
            <xs:sequence>
                <xs:element name="organizationId" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="CreateOrganizationCommand">
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="userId" type="xs:string"/>
                <xs:element name="englishName" type="xs:string"/>
                <xs:element name="type" type="xs:string"/>
                <xs:element name="country" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="GetImportsAction">
            <xs:choice>
                <xs:element name="GetImportsCommand"
type="GetImportsCommand"/>
                <xs:element name="GetImportsResponse"
type="GetImportsResponse"/>
                <xs:element name="error" type="ErrorResponse"/>
            </xs:choice>
        </xs:complexType>
        <xs:complexType name="GetImportsResponse">
            <xs:sequence>
                <xs:element name="ImportId" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="GetTransformationsAction">
            <xs:choice>
                <xs:element name="GetTransformationsCommand"
```

```
type="GetTransformationsCommand"/>
                        <xs:element name="GetTransformationsResponse"
type="GetTransformationsResponse"/>
                        <xs:element name="error" type="ErrorResponse"/>
                </xs:choice>
        </xs:complexType>
        <xs:complexType name="GetTransformationsResponse">
                <xs:sequence>
                        <xs:element name="transformationId" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="GetTransformationsCommand">
                <xs:sequence>
                        <xs:element name="organizationId" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="GetImportsCommand">
                <xs:sequence>
                        <xs:element name="organizationId" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="CreateUserAction">
                <xs:choice>
                        <xs:element name="CreateUserCommand"
type="CreateUserCommand"/>
                        <xs:element name="CreateUserResponse"
type="CreateUserResponse"/>
                        <xs:element name="error" type="ErrorResponse"/>
                </xs:choice>
        </xs:complexType>
</xs:schema>
```