Mapping Museums, *ETL* process documentation. *

Nick Larsson †

Version	Date	Comment
1.0	25th June 2018	First draft

Abstract

This document describes the principles and engineering of the Extract,Transform,Load (*ETL*) process for the Mapping Museums project. [1]

The requirements and the design philosophy behind the development of the *ETL* tools are described along with their usage.

1 TODO

- Verify loaded data. Will be done when test scripts have been created.

2 Introduction

Mapping Museums (MM) is a four-year project that will produce the first authoritative database of museums that opened and closed during a period of rapid expansion and change, and will provide the first evidence-based history of independent museums and their links to wider cultural, social, and political concerns.

The *ETL* process is supporting the database used as a backend for the web application allowing researchers to carry out analytics and explore the

*The project is funded by the Arts and Humanities Research Council.

†N. Larsson is a researcher at Department of Computer Science and Information Systems, Birkbeck University, London, UK. E-mail: nick@dcs.bbk.ac.uk

[1]Mapping museums, `http://blogs.bbk.ac.uk/mapping-museums/`

different aspects of the dataset. The Mapping Museums (MM) project employed an agile methodology with new data being collected and added during the course of the project. The *ETL* process must be able to respond to these changes in a resonable time which requires a declarative system with high degree of automation.

3 Outline of the MM *ETL* process

3.1 Overview of the MM tasks

The tasks involved in the MM *ETL* process are as follows:

- Extract the source data(s) to a common format
- Data quality control of source data
- Create transformation rules for source data
- Generate transformation code
- Quality control of loadable data
- Load data
- Verify loaded data

Figure 1 shows the MM workflow and the tasks.

4 Software and tools

The project is used a semantic approach to support agility and linking of external data to aid analytics. The software tools used to implement the MM ETL process are illustrated in Table 1.

4.1 Overview of the ETL file and script structure

The top directory, see Figure 2, consists of a code directory and a staging directory. The staging directory has a RDF directory and versions VX where the RDF files will be generated. This area needs to be defined as a loading area in the Virtuoso installation. The etljob directory is where the workflow manager generates jobs in case you have such a tool installed. The code directory has a subdirectory for each data subset so that dependencies can

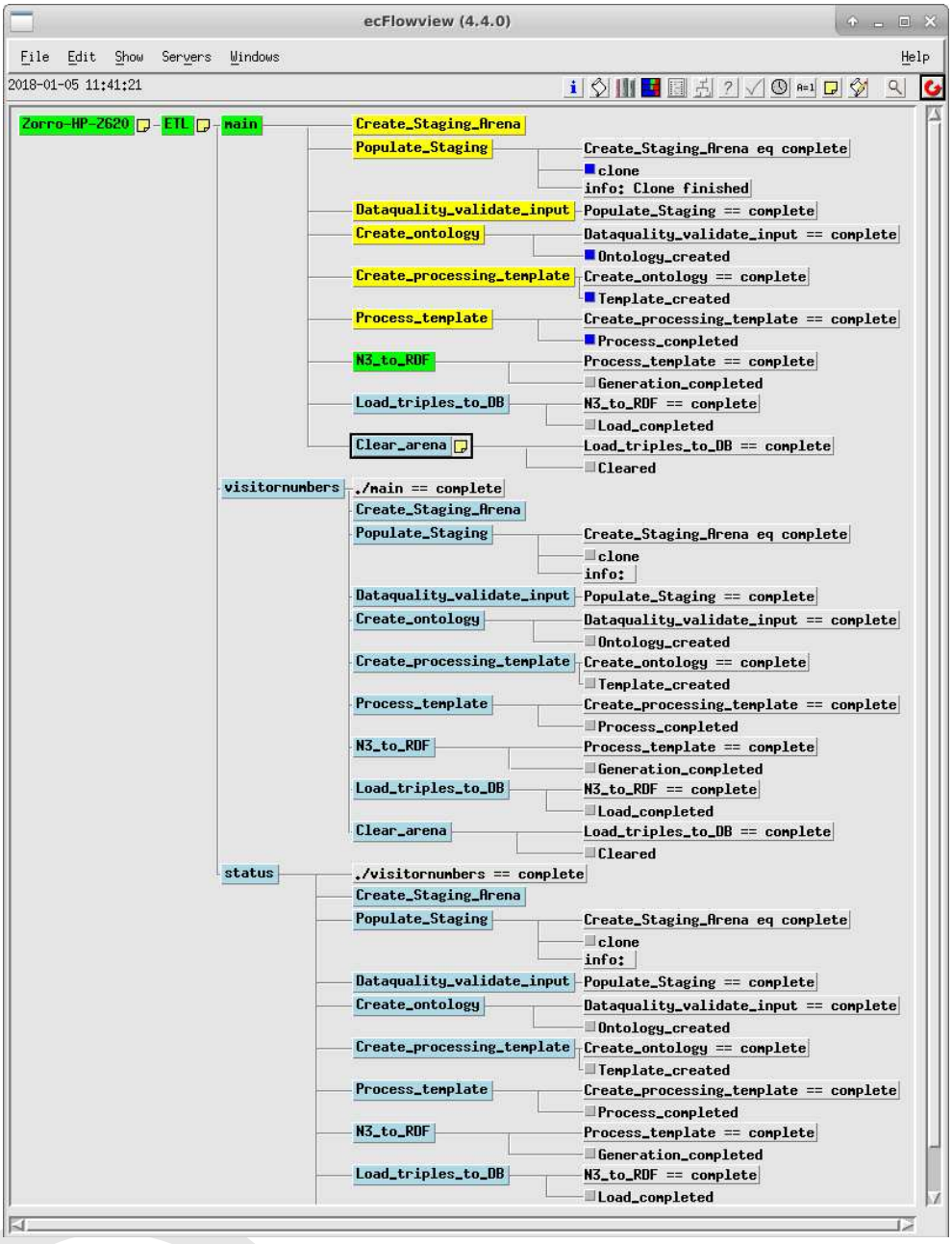

Figure 1: MM ETL workflow

Figure 2: MM ETL directory structure

be orchestrated and to keep files separated for the tasks. All tasks use the same code but run in individual directories.

The code directory contains all python code and a number of files for definitions which are replicated in all subdirectories. The files are summarised in Table 2. By looking at the script that runs all tasks in directory code we can determine or adjust which tasks are to be executed, variable TASKS.

The files in Table 2 are replicated for each directory in order to process a subset of the data. The processing is done by the python programs in the 'code' directory which is the project BIN. Table 3 lists the programs and their purpose.

4.2 Installation and requirements

This installation assumes all software in Table 1 is installed on the same filesystem.

4.2.1 MM ETL system

Use git to clone the project to a local directory from this link:`http://XXXX`

Software	Use
RDF2RDF	Converter between various semantic formats.
Korn shell	Standard shell to runs all scripts.
Virtuoso	Used as the backend data store to hold configurations
Graffoo yEd	Used as a front end to visualise the knowledge graph
GraphML	Used to create the knowledge graph and supported by Graffoo
RDF/RFS	Used to encode the data and model for storage in the data store
Python	Used for programming the MM ETL system.

Table 1: Software used to implement the MM ETL process

4.2.2 Configuring the MM ETL scripts

The following variables will need configuration to the actual location on the filesystem:

Directory: code

- file:etl.def The **STAGING_ARENA** defines where the files for uploading to the database are created. **VERSION** is the version of the dataset and all RDF will be moved to a directory with this name. Version in this case also means a new graph as each version is in separate graphs. If you are using Venv this is also where the Python environment should be initialised.
- file:suite.env **WDIR** (Working directory) should be defined to point to the directory where the data subset is processed, i.e. a subdirectory of code. This definition needs to be placed in each suite.env in each subdirectory of code. Additionally in each suite.env the **XBIN** (executables) directory needs defining to the directory code where all the python programs reside.
- file:common.def **FNAME** needs to point to the main spreadsheet in the main subdirectory. This variable is used in other directories than main so need to be defined on a global level.

File	Use
common.def	Used for defining names for all subdirectories
etl.def	Initialises the python environment and staging
suite.env	Used for defining names for a specific subdirectory
suite.ksh	Runs all tasks for this directory

Table 2: Definition files

File	Use
makeGMLNodesFromSheet.py	Creates an ontolgy from the spreadsheet encoded in graphML (Graffo)
DQLprocess.py	Data quality processing of CSV input
readontology.py	Reads the graphML ontoly and generates a data statement template
processMuseums.py	Processes the data statement template together with the spreadsheet to generate n3 statements

Table 3: Program files

Script	Use
Create_ontology.ksh	Creates an ontolgy from the spreadsheet encoded in graphML (Graffo)
Create_processing_template.ksh	Reads the graphML ontoly and generates a data statement template
Process_template.ksh	Processes the data statement template together with the spreadsheet to generate n3 statements
N3_to_RDF.ksh	Converts the n3 statements into RDF/XML which will verify its syntactic validity
Load_triples_to_DB.ksh	Loads the RDF files into the database
suite.ksh	The script that runs all of the above scripts

Table 4: Script files

4.2.3 RDF2RDF

Download and install from: `http://www.l3s.de/~minack/rdf2rdf/`

4.2.4 Virtuoso

Virtuoso can be downloaded from : `https://shop.openlinksw.com/license_generator/virtuoso-download/`

The version used in this project is 7.5. Follow the instructions in the INSTALL file.

4.2.5 yEd

yED can be downloaded from: `https://www.yworks.com/downloads#yEd`
This is not strictly necessary for defining the MM ETL process but may be preferable as an editor compared to a text editor.

4.2.6 Python

Python 2.7.12 has been used and required packages are listed in appendix A.
CULL UNNCESSARY PACKAGES

4.2.7 Ksh

Use the version of your OS as this is a stable software.

5 *ETL* tasks in detail

For purposes of documentation a **demo** directory has been added to the structure in Figure 2. The following describes the functionality by running the tasks in the demo directory.

5.1 Extract the source data(s) to a common format

Source data arrives as Excel sheet which is converted to a csv file. The csv file is encoded to UTF-8 and gets a new column delimiter, $. A header is added describing the data. Each cell in the table is separated by a $ and text is quoted, see Figure 3.

Our demo example as shown in Figure 3 below:

Each column has a datatype, see line 3, Figure 3. This could be any datatype in the XSD definition or a datatype from our datatype model shown in Table 5.

```
"Project_id"$"Name"$"Open"
"Project_id"$"Name"$"Open"
xsd:string$xsd:string$xsd:positiveInteger
"visible"$"visible"$"visible"
0$1$2
"mm.domus.YH023"$"Abbeydale Industrial Hamlet"$1970
"mm.domus.YH021"$"Kelham Island Museum"$1982
```

Figure 3: CSV header

The content of the header lines is as follows:

1. Label of data (not used)
2. Predicate name of relation [A-Z,a-z,_]
3. Data type
4. Visibility (not used)
5. Order of columns are numbered from zero starting from left

5.2 Data quality control of source data

Scripts and programs:*DQLprocess.py*

The csv file is read by the *DQLprocess.py* program and the datatype annotation determines the check applied. Integers,decimals and the types from our datatype model are checked. Errors and missing values are reported on and optionally removed. The exit status of the program will be the number of errors detected. Data quality control is only done in the main data subset.

5.3 Create transformation rules for source data

Scripts and programs:*Create_ontology.ksh*,*makeGMLNodesFromSheet.py*

The ontology for the dataset is built by analysing the csv file and creating a graphML file that represents the classes as nodes and the relationships as edges. This is traditionally how tabular data is represented in semantic databases: Each row is a new class individual and each column is a datproperty to the class. This is done in the **G**raffoo language where each

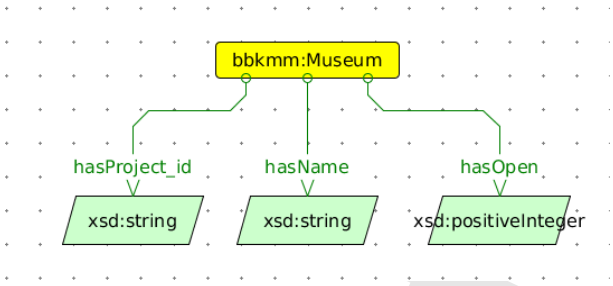

Figure 4: Ontology generated from csv file

node represents a class and subject,a vector describes the predicate and another type of node describes the attribute and type.

Graffoo knowledge graphs are encoded in **g**raphML standard and we create a **g**raphML file describing the column in the csv file. This is done by the *makeGMLNodesFromSheet.py* in the task *Create_ontology.ksh*. The program takes two templates for generating data and object properties which are common to the ETL process as well as a csv file to analyse, i.e. the file in Figure 3, *demo.csv*. Running the task creates the ontology *demoontology.graphml* by combining the output with a header and root node (P1) and trailer (P2). The root node is always node 0 (zero). The resulting ontology can bee seen in Figure 4. The *makeGMLNodesFromSheet.py* program can handle a subset of XSD types:integer,date,string,boolean and decimal. In addition the datamodel contains some abstract data types to handle date ranges,lists (bag of words) and hierarchies, see Table 5. These datatypes are understood by the web application and speeds up the modeling of data which naturally falls in to these categories.

The models above generate RDF data from the column data. The namelist is a collection of all values in the column, the range:datatype has both start and end present on each row as in "1987;1999" separated by a semicolon. The NamedHierarchy data is modeled on the UNIX file system and values are specified as a string with levels separated by slashes: "/LocalAuthority/Independent".In this way a complete graph is generated for all columns in the csv file which represents the ontology.

MM Datatype Model	Use
bbkmm:NameList	Used to create a bag of words from the column in the source data
range:datatype	Used to create time range model to hold start and end dates. The range data is typically positiveInteger or date
hier:NamedHierarchy	Used to create a subclass hierarchy from the column data.

Table 5: Complex datatypes

5.4 Generate transformation code

Scripts and programs:*Create_processing_template.ksh,readontology.py*

With the ontology and csv data in place the transformation from csv to data statements for the chosen technology, in our case **R**DF/**R**DFS, can be generated. This is done in the next task in *suite.ksh Create_processing_template.ksh*. The task runs the program *readontology.py* which takes as input the ontology created in the previous task and produces a data statement template in the "N3" notation. The ontology is analysed and the data statements are created from the information in the graph. This creates the link between the data source columns and the data model to be filled in by the next execution stage as well as declaring all classes and predicates. See Figure 5. Two lists are generated to hold predicates and datatypes for easy access from the web application, *bbkmm:PredicateList_demo* and *bbkmm:DataTypeList_demo*.

5.5 Execute transformation code to generate loadable data

Scripts and programs:*Process_template.ksh,processMuseums.py*

This step creates the RDF statements as seen in Figure 5.

5.6 Quality control of loadable data

Scripts and programs:*N3_to_RDF.ksh*,*RDF2RDF*

The loadable data is cleaned of empty data statements resulting from empty cells in the csv file and transformed to **R**DF/XML. This transformation will catch any syntactic errors in the data statements and ensure that the loader mechanism of the database will not abort.

5.7 Load data

Scripts and programs:*Load_triples_to_DB.ksh*

Prior to loading the data if you are reloading the same version of the dataset, as set in the **VERSION** variable, the existing version should first be deleted. This can be done via ISQL in Virtuoso but I have done this manually via the web Conducter interface. Loading of the data is done by invoking the ISQL command language interpreter and sending a command to load all files in the staging area. The log file will confirm that no errors were encountered. This is not done in the demo.

5.8 Verify loaded data

Verifying the data will be made by collecting queries that the graphical user interface makes to populate menus and drive visualisation. If the data returned from these queries is identical or diffs to added classes or predicates the loading is deemed successful.

6 URI scheme

This section will describe how the URIs are generated from the model to ensure uniqueness and aid in debugging/readability. And example of the URI generation function can be seen in Figure 5, **CLASSURI**. The function takes the classname for the resource and a nodename. The nodename is the name of the node in the graphML definition of the ontology, in this case the root node zero, for the Museum class. The generated urls can be seen in Figure 6.

In Figure 7 the constituents are shown. The prefix,class and node are composed from the ontology and at the end the individual museum id (ProjectId)is appended from the processed row in the spreadsheet. The uniqueness is assured if no rows are the same. There cannot be any other nodes

Parameter	Content
clazzname	Class name from graphML file
nodename	Node id from graphML file
propertyname	Name as declared in graphML file
clazzdata	Any data supplied in the graphML file
rangetype	The XSD type for the property value
symboltable	The hashtable used as the programs symboltable
rowmodel	The data row currently being processed
matrixmodel	The tabular model as a whole matrix,i.e. the sheet

Table 6: Extension method property call parameters and content

with the same Node number so traceability to class and spreadsheet row is obvius and the readabilty for humans greatly helps in debugging.

7 Extension methods

The system provides an extension mechanism to accomodate more complex data transformations via the clazzes directory. This is used to generate the ONS geo classification in the geonames directory. Any class mentioned in the graphML file will be loaded from the clazzes module directory. The class name will inherit the prefix from the ontology, i.e. bbkmm, for the expected classname with the colon replaced by and underscore: **bbkmm_English_CA**. To separate functionality one can just declare the classes with a different prefix. Properties declared in the graphML file on the class will be called with the interface, again with the prefix replaced by underscore, as shown in Figure 8. Table 6 explains the parameters in the property interface.

The execution of the property method can return any valid N3 statement or, more powerful, a statement template to be executed in the calling program. The latter can leverage all defined methods in the calling program.

7.1 Terms and acronyms

RDF The Resource Description Framework *RDF* is a family of World Wide Web Consortium (W3C) specifications [1] originally designed as a metadata data model. It has come to be used as a general method for conceptual description or modeling of information that is implemented in web resources, using a variety of syntax notations and data serialization formats. `http://en.wikipedia.org/wiki/Resource_Description_Framework`.

N3 A shorthand non-XML serialization of Resource Description Framework models, designed with human-readability in mind. `https://en.wikipedia.org/wiki/Notation3`

Ontology In computer science and information science, an ontology encompasses a representation, formal naming, and definition of the categories, properties, and relations of the concepts, data, and entities that substantiate one, many, or all domains. `https://en.wikipedia.org/wiki/Ontology_(information_science)`.

ETL In computing, extract, transform, load (ETL) refers to a process in database usage and especially in data warehousing.
`https://en.wikipedia.org/wiki/Extract,_transform,_load`.

```
#################################################################
#
#$Content classes
#
#################################################################


bbkmm:Museum a    owl:Class, rdfs:Class;
rdfs:label           "bbkmm:MuseumClass"@en ;
rdfs:comment         "bbkmm:MuseumClass"@en .

 bbkmm:hasProject_id rdf:type owl:DatatypeProperty ;
                rdfs:label "bbkmm:hasProject_id"@en ;
                rdfs:comment "bbkmm:hasProject_id"@en ;
                rdfs:domain bbkmm:Museum ;
                rdfs:range xsd:string .

 bbkmm:hasName rdf:type owl:DatatypeProperty ;
                rdfs:label "bbkmm:hasName"@en ;
                rdfs:comment "bbkmm:hasName"@en ;
                rdfs:domain bbkmm:Museum ;
                rdfs:range xsd:string .

 bbkmm:hasOpen rdf:type owl:DatatypeProperty ;
                rdfs:label "bbkmm:hasOpen"@en ;
                rdfs:comment "bbkmm:hasOpen"@en ;
                rdfs:domain bbkmm:Museum ;
                rdfs:range xsd:positiveInteger .
#################################################################
#
#$Individuals
#
#################################################################

${classURI("bbkmm:Museum",n0)}  ${property("bbkmm:hasProject_id",
                                           "xsd:string",
                                           0,
                                           "visible")} .
${classURI("bbkmm:Museum",n0)}  ${property("bbkmm:hasName",
                                           "xsd:string",
                                           1,
                                           "visible")} .
${classURI("bbkmm:Museum",n0)}  ${property("bbkmm:hasOpen",
                                           "xsd:positiveInteger",
                                           2,
                                           "visible")} .


bbkmm:PredicateList_demo bbkmm:contents
("bbkmm:hasProject_id" "bbkmm:hasOpen" "bbkmm:hasName") .

bbkmm:DataTypeList_demo bbkmm:contents
( "bbkmm:hasProject_id#xsd:string" "bbkmm:hasOpen#xsd:positiveInteger" "bbkmm:hasName#xsd:string") .
```

Figure 5: Code transformation statements/macros

```
################################################################
#$Content classes
bbkmm:Museum a    owl:Class, rdfs:Class;
rdfs:label           "bbkmm:MuseumClass"@en ;
rdfs:comment         "bbkmm:MuseumClass"@en .


 bbkmm:hasProject_id rdf:type owl:DatatypeProperty ;
                rdfs:label "bbkmm:hasProject_id"@en ;
                rdfs:comment "bbkmm:hasProject_id"@en ;
                rdfs:domain bbkmm:Museum ;
                rdfs:range xsd:string .

 bbkmm:hasName rdf:type owl:DatatypeProperty ;
                rdfs:label "bbkmm:hasName"@en ;
                rdfs:comment "bbkmm:hasName"@en ;
                rdfs:domain bbkmm:Museum ;
                rdfs:range xsd:string .

 bbkmm:hasOpen rdf:type owl:DatatypeProperty ;
                rdfs:label "bbkmm:hasOpen"@en ;
                rdfs:comment "bbkmm:hasOpen"@en ;
                rdfs:domain bbkmm:Museum ;
                rdfs:range xsd:positiveInteger .

################################################################
<http://bbk.ac.uk/MuseumMapProject/def/Museum/id/n0/mm.domus.YH021> a bbkmm:Museum .
<http://bbk.ac.uk/MuseumMapProject/def/Museum/id/n0/mm.domus.YH021>
   bbkmm:hasProject_id "mm.domus.YH021"^^xsd:string .

<http://bbk.ac.uk/MuseumMapProject/def/Museum/id/n0/mm.domus.YH021>
   bbkmm:hasName "Kelham Island Museum"^^xsd:string .

<http://bbk.ac.uk/MuseumMapProject/def/Museum/id/n0/mm.domus.YH021>
   bbkmm:hasOpen "1982"^^xsd:integer .

bbkmm:PredicateList_demo bbkmm:contents ("bbkmm:hasProject_id" "bbkmm:hasOpen" "bbkmm:hasName") .
bbkmm:DataTypeList_demo bbkmm:contents
( "bbkmm:hasProject_id#xsd:string" "bbkmm:hasOpen#xsd:positiveInteger" "bbkmm:hasName#xsd:string") .
```

Figure 6: Code transformation statements/macros

```
${classURI("bbkmm:Museum",n0)}  -> translates to
<http://bbk.ac.uk/MuseumMapProject/def/Museum/id/n0/mm.domus.YH021> a bbkmm:Museum .
{prefix.............................}{Class}{Node}{ProjectId.....}
```

Figure 7: Code transformation statements/macros

```
    def bbkmm_propertyname(self,clazzname,nodename,propertyname,clazzdata,rangetype,
                           symboltable,rowmodel,matrixmodel)
```

Figure 8: Property value generation from executing class method