Realizing a Semantic Web Application

Emanuele Della Valle

Center of **E**xcellence **F**or **R**esearch, **I**nnovation, **E**ducation and industrial **L**ab partnership - Politecnico di Milano

http://www.cefriel.it
http://swa.cefriel.it
emanuele.dellavalle@cefriel.it
http://emanueledellavalle.org

Goal

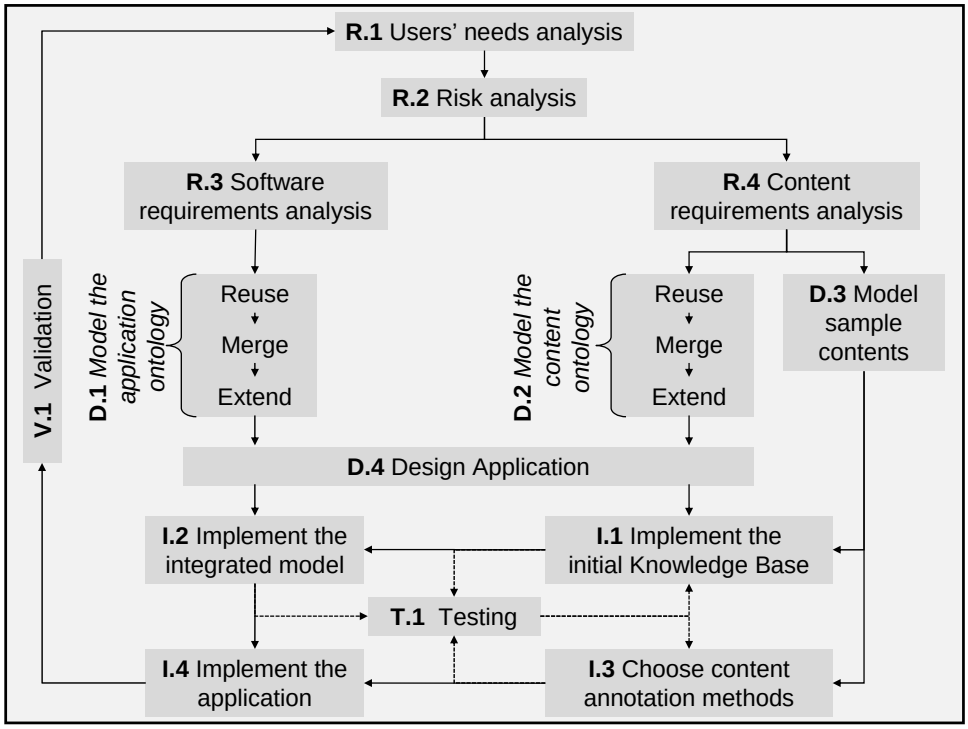
- We will develop together an application of the Semantic Web we named **Music Event Explorer** o simply *meex*
- We will challenge the Semantic Web technologies in realizing a new service for Web users
  - Using
  - Transforming and
  - Combining existing data

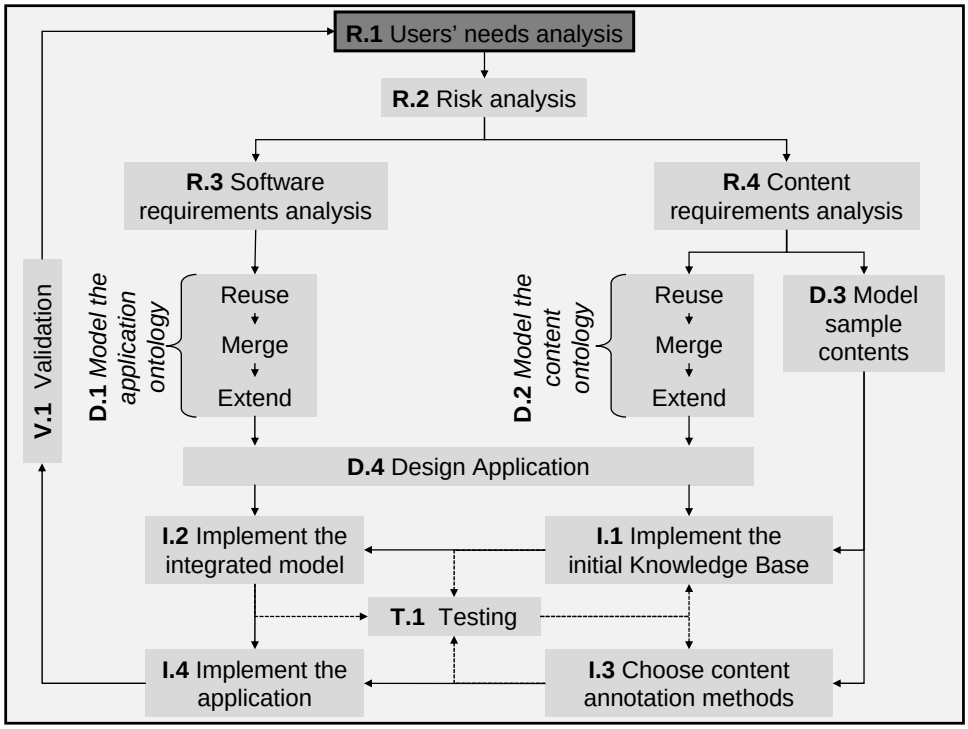
Ingredients

- **OWL** as modelling language for the data sources;
- **RDF** as unified data model;
- **GRDDL** as a standard approach to translate in RDF the data stored in XML data sources;
- **D2RQ** as tool to translate in RDF the data stored in relational data sources;
- **Jena** as application framework to merge the various data in a single RDF model and manipulate it;
- **SPARQL** as standard query language to access RDF data;
- A **RDF storage** to guarantee persistency
- A **OWL reasoner** to infer new knowledge;
- **Exhibit** as user interface.

Approach

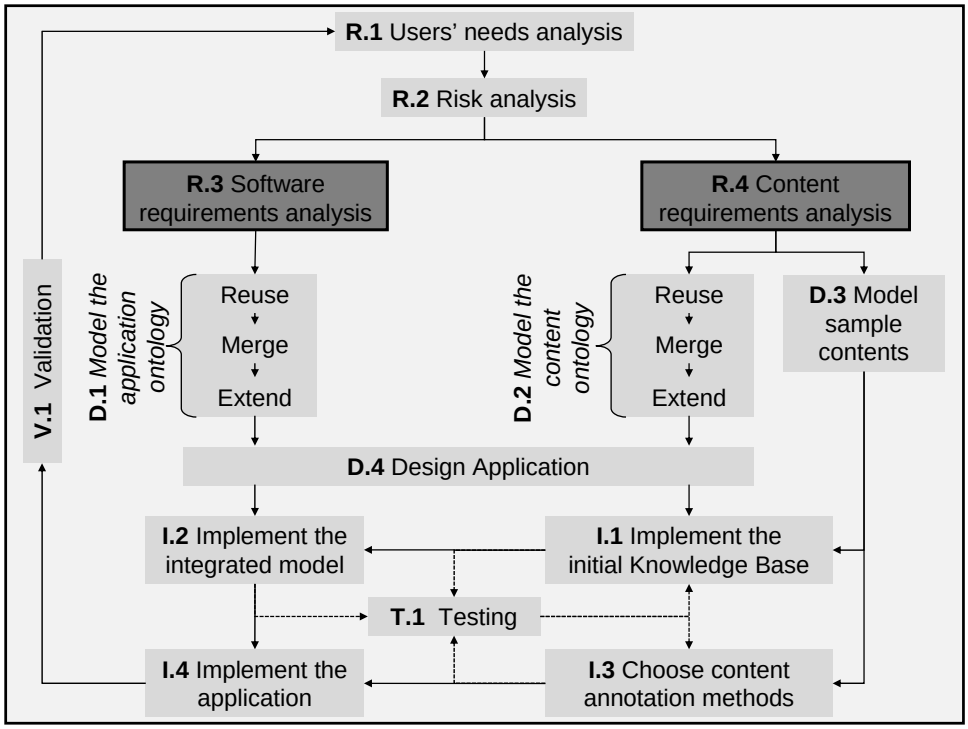
- In order to realize meex
  1. We start from the user need
  2. We derive user requirements
  3. We develop the ontologies and the software components
- While developing we will explain the use of Semantic Web technologies and tools.
- A demonstrative installation of the application, together with the source code, is available at
  - http://swa.cefriel.it/meex


R.1 Users' needs analysis
R.2 Risk analysis
R.3 Software requirements analysis
R.4 Content requirements analysis
V.1 Validation
D.1 Model the application ontology
Reuse
Merge
Extend
D.2 Model the content ontology
Reuse
Merge
Extend
D.3 Model sample contents
D.4 Design Application
I.2 Implement the integrated model
I.1 Implement the initial Knowledge Base
T.1 Testing
I.4 Implement the application
I.3 Choose content annotation methods


R.1 Users' needs analysis
R.2 Risk analysis
R.3 Software requirements analysis
R.4 Content requirements analysis
V.1 Validation
D.1 Model the application ontology
Reuse
Merge
Extend
D.2 Model the content ontology
Reuse
Merge
Extend
D.3 Model sample contents
D.4 Design Application
I.2 Implement the integrated model
I.1 Implement the initial Knowledge Base
T.1 Testing
I.4 Implement the application
I.3 Choose content annotation methods

Users' needs

- Imagine the users need to explore music events related to a given music style
  - An event is a concert, a show or a workshop at which one or more artist participate.
  - An artist is either a single musician or a band.
- For instance, if a user is interest in Celtic music meex
  - finds the artists that play Celtic Music
  - searches for events of those artists
  - allows the users to explore the events related to each artist as a list, on a time line and on a map



Content requirements analysis

- Given we are developing a Semantic Web application is cruscial we reuse data already available on the Web
  - EVDB - http://eventuful.com
  - MusicBrainz - http://musicbrainz.org
  - MusicMoz - http://musicmoz.org

EVDB

- EVDB is a Web 2.0 website that makes available information about event all around the world
- For each event it knows
  - The start data
  - The end data
  - The place in terms of address and geographic coordinates
- EVDB offers a Web API in the form of a REST service
  - see http://api.evdb.com

MusicBrainz

- MusicBrainz
  - is a Web 2.0 website that gathered a large amount of information about music
  - offers information about
    - artists and bands
    - songs, albums and tracks
    - relations among artists and bands
- The data of MusicBrainz are available as a PostgreSQL dump
  - see http://musicbrainz.org/doc/DatabaseDownload

MusicMoz

- MusicMoz
  - is another Web 2.0 website dedicated to music
  - offers information about
    - artists and bands including their nationality
    - music styles and their taxonomic relationships
    - the styles each artist or band plays
  - reuses MusicBrainz identifier for artists and bands
- The data of MusicMoz are available as large XML files
  - see http://musicmoz.org/xml/

meex needs to merge this data

- meex in order to be able to manipulate all this data at the same time needs to merge the data of the three data sources.
- The artists and bands information from MusicBrainz should be linked to
  - the music styles they play from MusicMoz
  - the events related to them from EVDB

Data Licences

- The data of all three data sources are freely usable, we just need to make sure that the logos of the three applications appears on each page of meex
- EVDB requests also to include a link to the permalink of the event on EVDB website
- MusicBrainz request also that derived data are made available in Creative Commons.
- Read out more here
  - EVDB - http://api.eventful.com/terms
  - MusicMoz - http://musicmoz.org/xml/license.html
  - MusicBrainz - http://musicbrainz.org/doc/DatabaseDownload

Application requirements analysis (1)

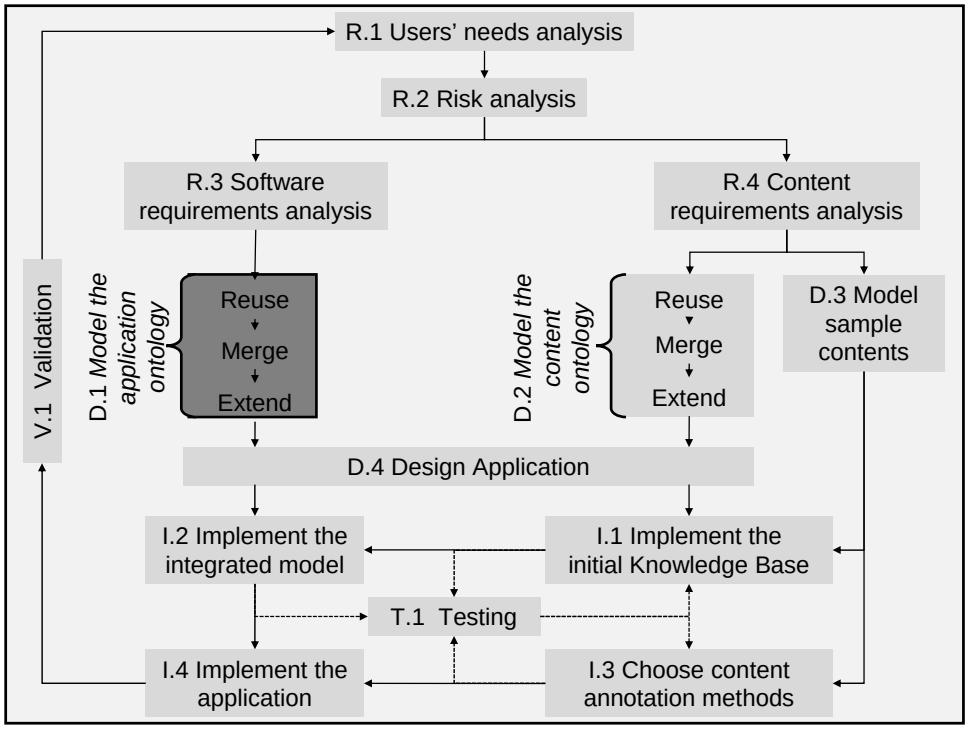
- In this step (namely R.3) we should elicit
  - functional requirements of the application
    - as grouping and filtering data
  - non-functional requirements of the application
    - as performance and scalability w.r.t. number of users
- However this is just a tutorial, therefore we concentrate on functional requirements, leaving non-functional requirements underspecified

Application requirements analysis (2)

- Meex
  - must enable a user to explore data in the form of
    - a list
    - a chronological graphic
    - a geographic map
  - for each event must show
    - name
    - begin and end date
    - place
  - for each artist must show
    - name
    - nationality
    - music styles he/she plays
    - related artists
  - must allow users to
    - filter and rank results

Model the Application Ontology

- As first design step (namely D.1) we model the **application ontology**
- meex must manage information related to
  - artists
  - events at which the artists participate and
  - music styles the artists play



Modeling **Performer** in OWL

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix meex: <http://swa.cefriel.it/meex#> .

meex:Performer    a  owl:Class ;
                  rdfs:label  "Performer" .

meex:fromCountry  a owl:DatatypeProperty ;
                  rdfs:domain meex:Performer ;
                  rdfs:range
                  <http://www.w3.org/2001/XMLSchema#string> .

meex:relatedPerformer  a  owl:ObjectProperty ;
                       rdfs:domain  meex:Performer ;
                       rdfs:range   meex:Performer .

[more to follow]
```

Meex.n3

Modeling **Style** in OWL

```
[follows]

meex:Style  a  owl:Class .
              rdfs:label  "Music Style" .

meex:performsStyle  a  owl:ObjectProperty ;
                       rdfs:domain  meex:Performer ;
                       rdfs:range   meex:Style .

[more to follow]
```

Meex.n3

Modeling **Event** in OWL

```
[follows]

meex:Event  a  owl:Class ;
            rdfs:label  "Event" .

meex:performsEvent  a  owl:ObjectProperty ;
                       rdfs:domain  meex:Performer ;
                       rdfs:range   meex:Event .

meex:hasWhen  a  owl:ObjectProperty ;
                 rdfs:domain  meex:Event ;
                 rdfs:range   gd:When .

meex:hasWhere  a  owl:ObjectProperty ;
               rdfs:domain  meex:Event ;
               rdfs:range   gd:Where
```

Meex.n3

- For each event we should model begin and end date together with the place, but an XML schema defined by Google exists; thus we decide to reuse it by merging it

Modeling **When** in OWL

```
[namespace declaration]

gd:When  a  owl:Class;
         rdfs:label  "Time" .

gd:startTime  a  owl:DatatypeProperty ;
              rdfs:domain  gd:When ;
              rdfs:range
                  <http://www.w3.org/2001/XMLSchema#string> .

gd:endTime    a  owl:DatatypeProperty ;
              rdfs:domain  gd:When ;
              rdfs:range
                  <http://www.w3.org/2001/XMLSchema#string> .

[more to follow]
```

GoogleSchema.n3

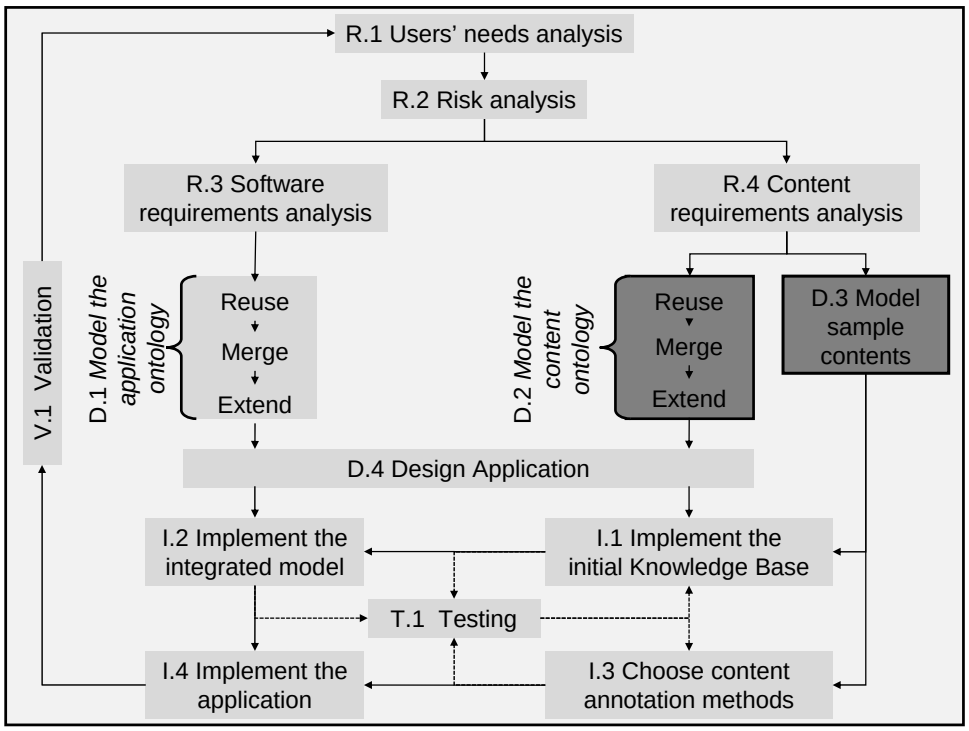
Modeling **Where** in OWL

```
gd:Where  a  owl:Class; rdfs:label  "Location" .

gd:postalAddress  a  owl:DatatypeProperty ;
                  rdfs:domain  gd:Where ;
                  rdfs:range
                   <http://www.w3.org/2001/XMLSchema#string>.

gd:hasGeoPt  a  owl:ObjectProperty ;
             rdfs:domain  gd:Where ;
             rdfs:range   gd:GeoPt  .

gd:GeoPt a  owl:Class ; rdfs:label  "Geo-referenced Point" .

gd:lat   a  owl:DatatypeProperty ;
          rdfs:domain  gd:GeoPt ;
          rdfs:range <http://www.w3.org/2001/XMLSchema#string>.

gd:lon    a  owl:DatatypeProperty ;
          rdfs:domain  gd:GeoPt ;
          rdfs:range <http://www.w3.org/2001/XMLSchema#string>.

gd:label  rdfs:subPropertyOf  rdfs:label .
```

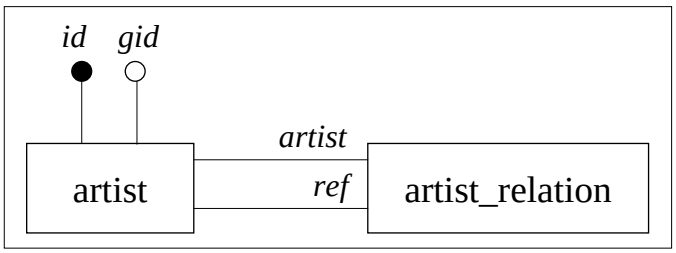
GoogleSchema.n3

Model the content ontology

- We keep following our approach and we model the content ontology (step P.2)
- The content ontology models in OWL the data of the three data sources used by meex
- In the mean time we also model the sample contents (step P.3) that we will use to test meex during its implementation (see test-first method from Agile manifesto)



Modeling MusicBrainz schema in OWL




```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix mb:      <http://musicbrainz.org/> .

mb:Artist  a  owl:Class ;
           rdfs:label  "MusicBrainz Artist and Band" .

mb:artist_relation  a  owl:ObjectProperty ;
                    rdfs:domain  mb:Artist ;
                    rdfs:range   mb:Artist .
```

MusicBrainz.n3

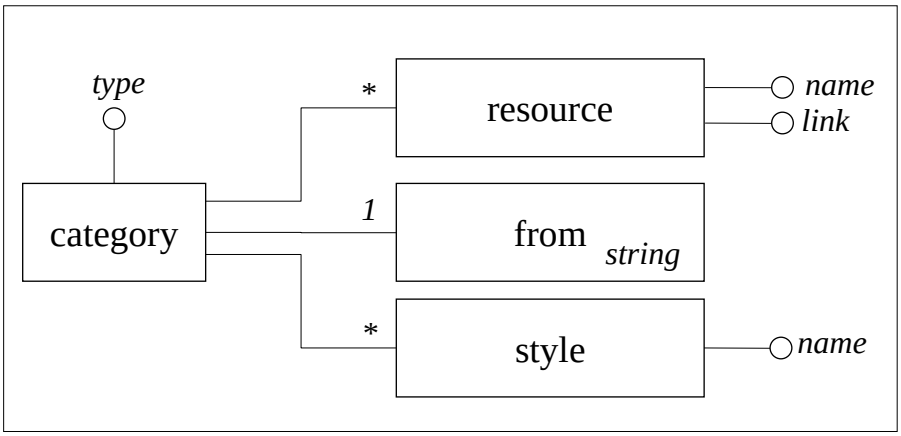
Sample data for MusicBrainz in OWL

```
mb:artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d.html
     a  mb:Artist ;
     rdfs:label  "The Beatles" ;
     mb:related_artist
          mb:artist/ebfc1398-8d96-47e3-82c3-f782abcdb13d.html ,
          mb:artist/618b6900-0618-4f1e-b835-bccb17f84294.html .

mb:artist/ebfc1398-8d96-47e3-82c3-f782abcdb13d.html
     a mb:Artist ;
     rdfs:label  "The Beach Boys" .

mb:artist/618b6900-0618-4f1e-b835-bccb17f84294.html
     a mb:Artist ;
     rdfs:label  "Eric Clapton" .
```

SampleInstance-MusicBrainz.n3

- Please note that we choose to build the URI using the ID that MusicBrainz uses to identify the artists. This allows for easier reuse of meex data in other applications

MusicMoz schema



Modeling MusicMoz schema in OWL

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix mm:      <http://musicmoz.org/> .
@prefix mb:      <http://musicbrainz.org/> .
mm:from  a  owl:DatatypeProperty ;
         rdfs:domain  mb:Artist ;
         rdfs:range <http://www.w3.org/2001/XMLSchema#string>.
mm:Style  a owl:Class ;
          rdfs:label  "MusicMoz Music Style" .
mm:hasStyle  a  owl:ObjectProperty ;
             rdfs:domain  mb:Artist ;
             rdfs:range   mm:Style .
```

MusicMoz.n3

Sample data for MusicMoz in OWL

```
mb:artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d.html
      mm:from "England" ;
      mm:hasStyle mm:style/British-Invasion ,
                  mm:style/Rock ,
                  mm:style/Skiffle .

mm:style/British-Invasion  a  mm:Style ;
      rdfs:label  "British Invasion" .
```

SampleInstance-MusicMoz.n3

- Please note that also in this case we use the ID derived from MusicBrainz

Modeling EVDB schema in OWL

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix evdb:    <http://eventful.com/> .
@prefix gd:      <http://schemas.google.com/g/2005> .
evdb:Event  a  owl:Class ;
            rdfs:label  "Eventful Event" .
evdb:hasWhen  a  owl:ObjectProperty ;
              rdfs:domain  evdb:Event ;
              rdfs:range   gd:When .
evdb:hasWhere  a  owl:ObjectProperty ;
               rdfs:domain  evdb:Event ;
               rdfs:range   gd:Where .
```

EVDB.n3

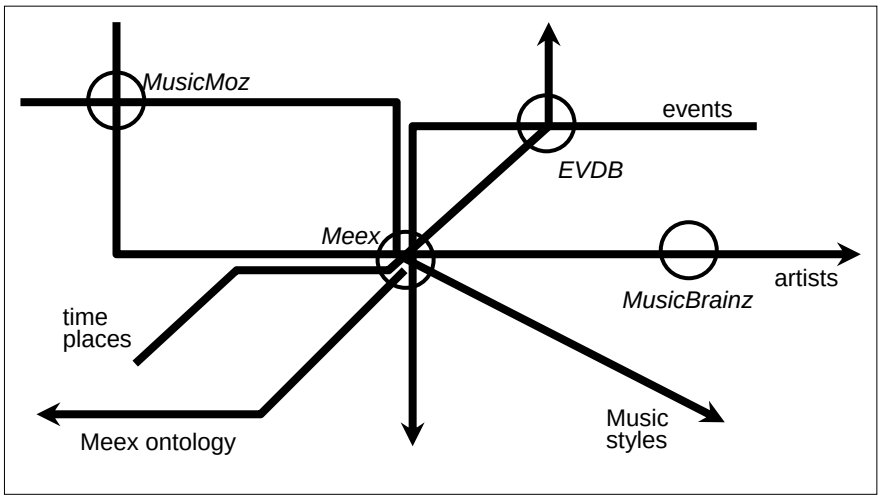
- Please note that we reuse the concepts When and Where we model in the application ontology by merging Google schema (see GoogleSchema.n3).

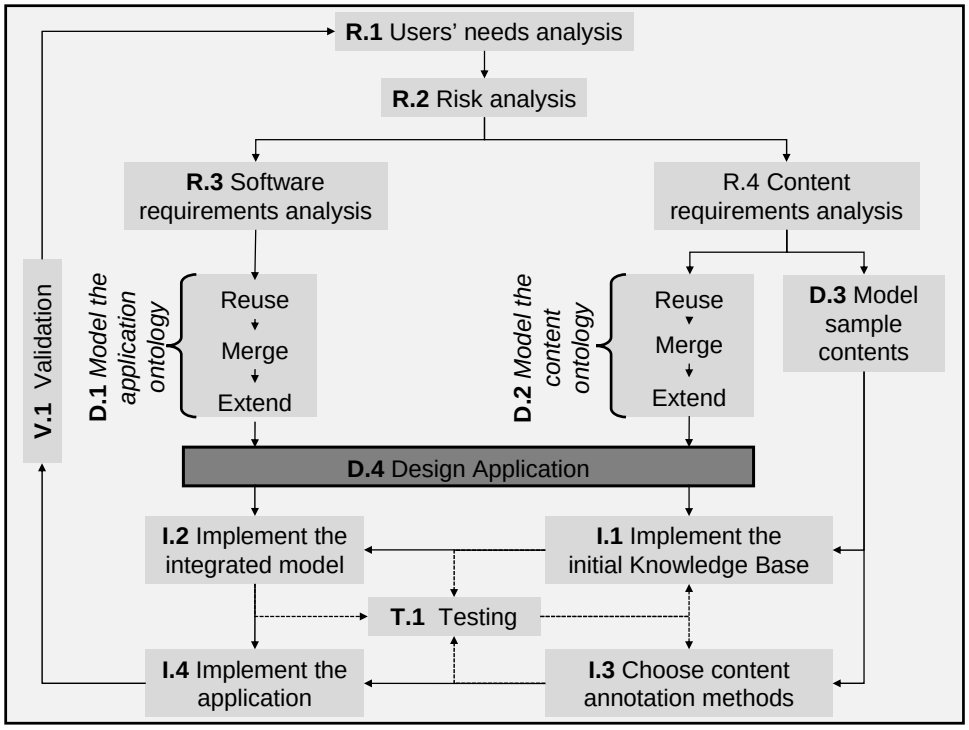
Sample data for EVDB in OWL

```
evdb:events/E0-001-008121669-0@2008022719  a  evdb:Event ;
 gd:label  "Tell Me Why: A Beatles Commentary" .
 evdb:hasWhen evdb:events/E0-001-008121669-0@2008022719_When;
 evdb:hasWhere evdb:events/E0-001-008121669-0@2008022719_Where.

evdb:events/E0-001-008121669-0@2008022719_When
 gd:startTime "2008-02-28" ;
 gd:endTime   "2008-02-28" .

evdb:events/E0-001-008121669-0@2008022719_Where
 gd:hasGeoPt evdb:events/E0-001-008121669-0@2008022719_GeoPt ;
 gd:label  "The Wilmington Memorial Library" ;
 gd:postalAddress  "175 Middlesex Avenue, Wilmington, USA" .

evdb:events/E0-001-008121669-0@2008022719_GeoPt
 gd:lat  "42.556943" ;
 gd:lon  "-71.165576" .
```

SampleInstance-EVDB.n3

"Application Connected by Concepts"

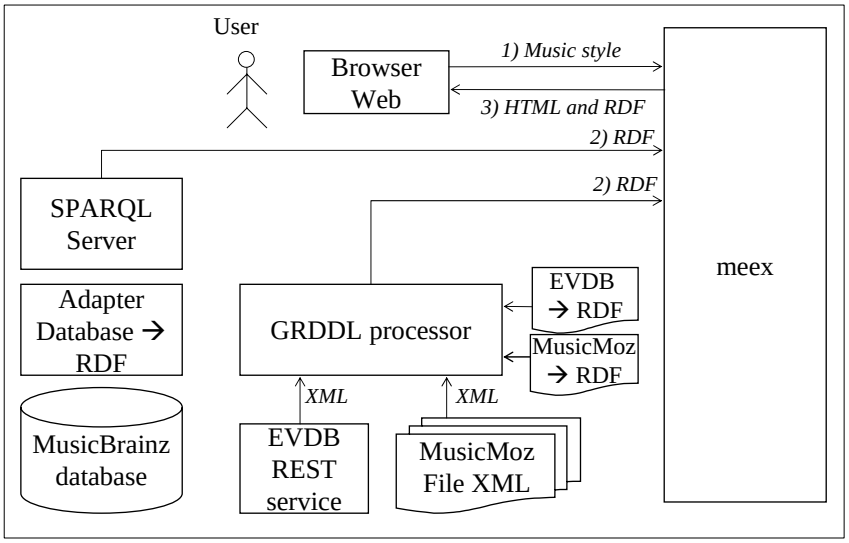




Summary

- We are done with the modeling of ontologies and sample contents
- We can now design meex (step D.4 of our approach)
- In order to design meex architecture
  - We first **design** its **interfaces** in terms of
    - both graphic user interface
    - and connection to the three data sources
  - Secondly we **design** how it works **inside** in terms of
    - components and
    - execution semantics

meex interfaces



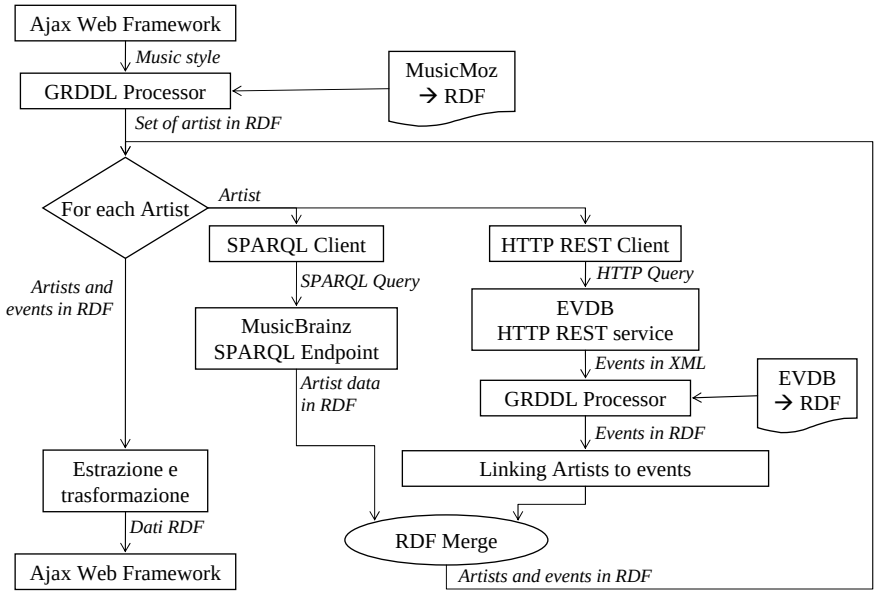
How we access the data

- In order to get RDF data out from the three external data source we can use different techniques
  - For **MusicBrainz** database we can use tools that enable to query non-RDF databases as virtual RDF graphs using a standard SPARQL endpoint
  - For **MusicMoz** XML files we can use a GRDDL processor using the XSLT `MusicMoz->RDF`
  - For **EVDB** we can use a GRDDL processor applying the XSLT `EVDB->RDF` to the XML file obtained using the EVDB REST service

User Interface

- In order to collect users' input and to present results back to the users, we can use Web 2.0 technologies and develop an AJAX interface
- Such AJAX interface must allow for
  - Inserting the music style, the resulting events will refer to
  - Exploring the events found by meex
  - Filtering the events based on
    - Artists
    - Their nationality
    - The music style they play

Designing how meex works inside



Execution Semantics (1)

1. The user requests a music style
2. meex access the local copy of MusicMoz and using the GRDDL processors obtains a set of artist that plays the given music style

[more to follow]

Execution Semantics (2)

[follows]

3. For each artist meex :
   a) uses the SPARQL client to query the MusicBrainz SPARQL endpoint and it obtains the artist name and his/her relationships with other artist
   b) invokes the EVDB REST service, it obtains the events that refer to the artist in XML and uses the GRDDL processor to obtain this data in RDF
   c) links the data about each artist to the data about the events that refers to him/her

[more to follow]

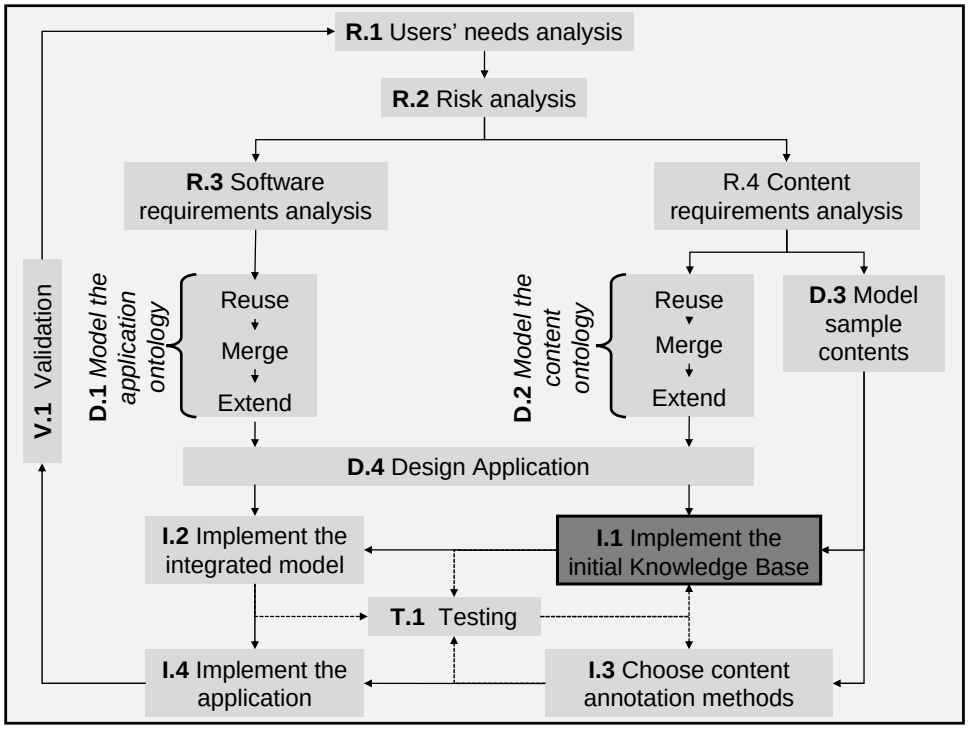
Execution Semantics (3)

[follows]

4. When all the peaces of information about artists and events are available in the RDF storage, meex extracts them and serializes them in the format of the Ajax Web framework
5. The ajax Web framework allows the user for exploring the events found by meex
6. When the user decides to start a new exploration, meex starts over from the beginning

Two important internal components

- The **RDF storage**
  - must be initialized with both the application and the content ontology
  - is filled in with the data meex loads from the three data source given the music style requested by the user
- The **reasoner**
  - allows all query in meex to be express in terms of the application ontology even if data are loaded from the data sources using the content ontology
- NOTE: the reasoner support the semantic integration of the data loaded from the external data sources. The meex's programmer can ignore that multiple and heterogeneous data sources were used to load data



Implement the initial Knowledge Base (1)

- We start implementing meex by setting up the initial knowledge base (step I.1)
- We need to select tools
  - to read and write RDF in the RDF/XML and RDF/N3 syntax
  - to manipulate programmatically RDF
  - to store RDF
  - to reason on OWL
  - to interpret SPARQL

Implement the initial Knowledge Base (2)

- We choose Jena because
  - offers API
    - to read and write different RDF syntax
    - provides a programmatic environment for RDF, RDFS and OWL, SPARQL a
  - guarantees RDF model persistence through several relational database adapters
  - includes a rule-based inference engine which implement OWL semantics
  - includes ARQ, a query engine that supports SPARQL
- In order to use the RDF storage and the OWL reasoner from Jena we need to configure them as shown in the following slides

Configuring the RDF storage

- We choose to use Derby (from Apache) as relational database underneath the RDF storage.

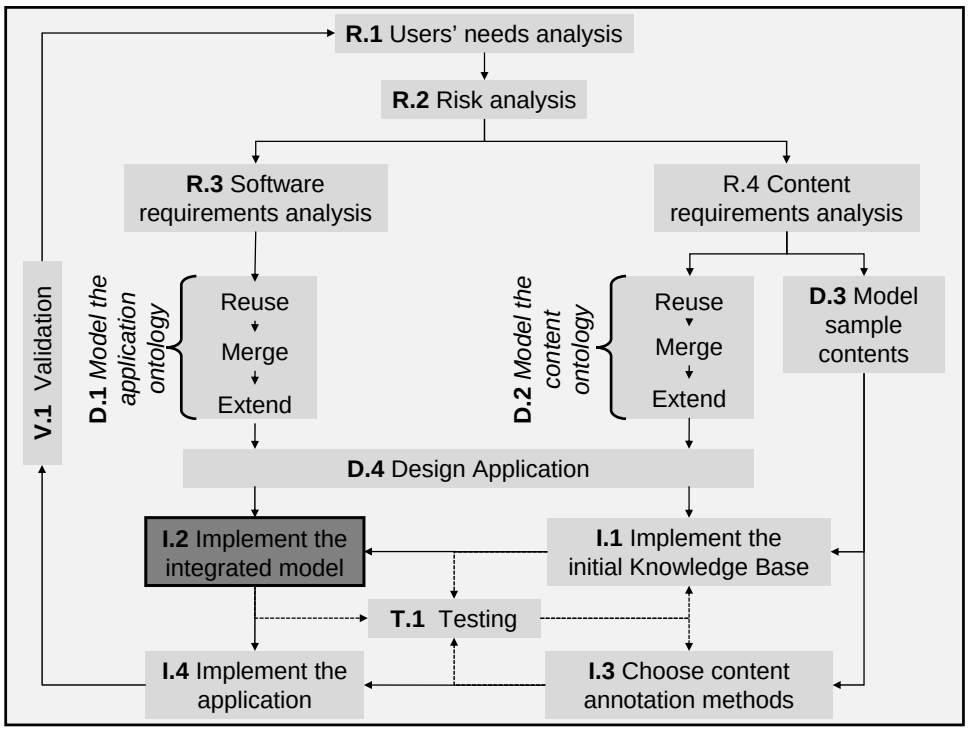
```
1. Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
2. DBConnection con = new DBConnection(
     "jdbc:derby:C:/Meex/RDFStorage;create=true",
     "sa", "", "Derby");
3. Model model =
     ModelFactory.createModelRDBMaker(con).
     createDefaultModel();
```

- With row 1 we tell Jena where to find the JDBC driver
- With row 2 we define the JDBC connection
- With row 3 we instantiate the object `model` of Jena we will use to access and manipulate the RDF model in the storage

Configuring the OWL reasoner

```
1. Reasoner reasoner = ReasonerRegistry.getOWLMicroReasoner();
2. model = ModelFactory.createInfModel(reasoner, model);
```

- Jena offers numerous options to configure the internal rule-based inference engine with different expressivity-performance tradeoffs
- We need simple reasoning features (i.e., subClassOf and subPropertyOf transitive closure), the OWL Micro configuration is, therefore, the most appropriate one
- With row 1 we instantiate a OWL micro reasoner
- With row 2 we instantiate a model with inference support using the model previously created and the OWL micro reasoner



Implement the integrated model (1)

- We move on with the implementation of meex realizing the integrated model (step I.2)
- In the integrated model we merge application and content ontology
  - Our intent is to integrate semantically the heterogeneous data coming from the external data sources
- In order to realize the integrated model we need to define a **bridge ontology** using the properties
  - `rdfs:subclassOf`
  - `rdfs:subpropertyOf`
- to connect classes and properties in the application ontology to those in the content ontology

Implement the integrated model (2)

```
1. mb:Artist rdfs:subClassOf meex:Performer .
2. mb:related_artist rdfs:subPropertyOf meex:relatedPerformer.
3. mm:Style rdfs:subClassOf meex:Style .
4. mm:hasStyle rdfs:subPropertyOf meex:performsStyle .
5. mm:from rdfs:subPropertyOf meex:fromCountry .
6. evdb:Event rdfs:subClassOf meex:Event.
7. evdb:hasWhen rdfs:subPropertyOf meex:hasWhen.
8. evdb:hasWhere rdfs:subPropertyOf meex:hasWhere.
```

- In rows 1 and 2 we connect the ontology of MusicBrainz to the application ontology, i.e.
  - the classes `mb:Artist` and `meex:Performer`
  - the properties `mb:related_artist` and `meex:relatedPerformer`.
- Likewise, in rows 3, 4 and 5, we connect the ontology of MusicMoz to the application ontology and
- in rows 6, 7 and 8 we connect the ontology of EVDB to the application ontology

Implement the integrated model (3)

- Thanks to this bridge ontology, when data loaded from the external data sources are inserted in the RDF storage (using the data source specific ontologies), the OWL micro reasoner infers the triples that represent the same data in the application ontology
- meex can, therefore, query the RDF storage homogeneously in the terms of application ontology without caring of the heterogeneous formats of the three data sources
- To give an idea of the differences, in the next slide we compare the data expressed
  - in MusicBrainz ontology and
  - in the application ontology

Implement the integrated model (4)

```
mb:artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d.html
     a  mb:Artist ;
     rdfs:label  "The Beatles" ;
     mb:related_artist
          mb:artist/ebfc1398-8d96-47e3-82c3-f782abcdb13d.html,
          mb:artist/618b6900-0618-4f1e-b835-bccb17f84294.html.
```

SampleInstance-MusicBrainz.n3

```
mb:artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d.html
   a meex:Performer ;
   rdfs:label  "The Beatles" ;
   meex:relatedPerformer
      mb:artist/ebfc1398-8d96-47e3-82c3-f782abcdb13d.html ,
      mb:artist/618b6900-0618-4f1e-b835-bccb17f84294.html .
```

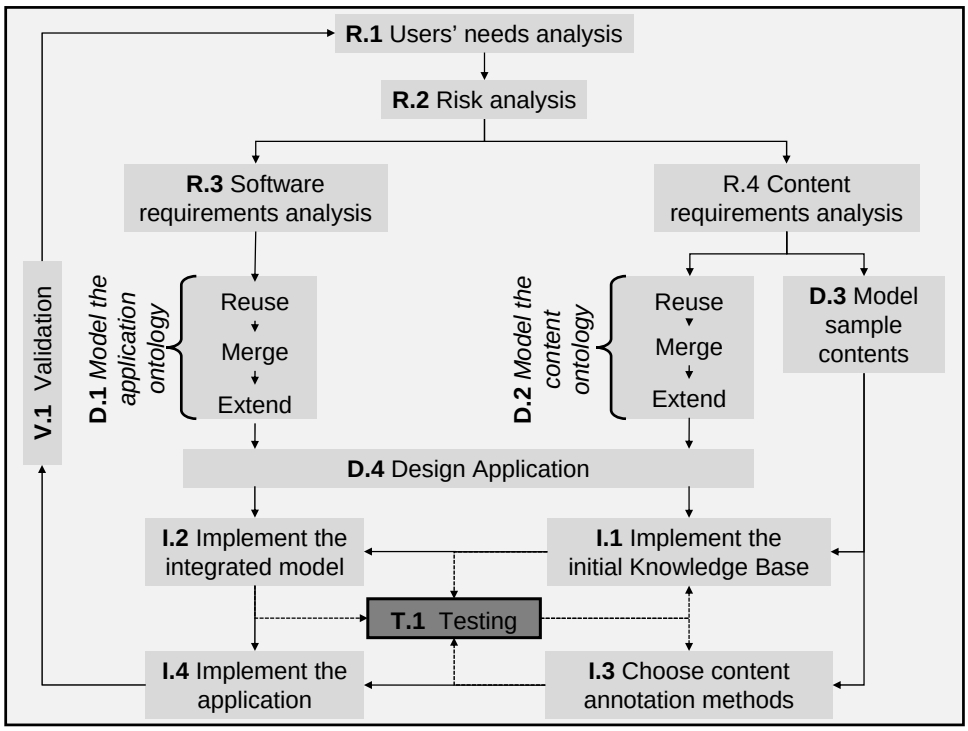
Dati-di-MusicBrainz-inferiti-usando-l-ontologia-ponte.n3

Implement the integrated model (5)

- Now that we have configure both the RDF storage and the reasoner we can load all ontologies

```
model.read("Meex.n3", "", "N3");
model.read("Google.n3", "", "N3");
model.read("MeexBindings.n3", "", "N3");
model.read("MusicBrainz.n3", "", "N3");
model.read("MusicMoz.n3", "", "N3");
model.read("EVDB.n3", "", "N3");
```

- Note that the `read` method of `model` requires:
  - The name of the file to load,
  - The base URI (in our case all URI are absolute) and
  - The RDF syntax in which data are serialized

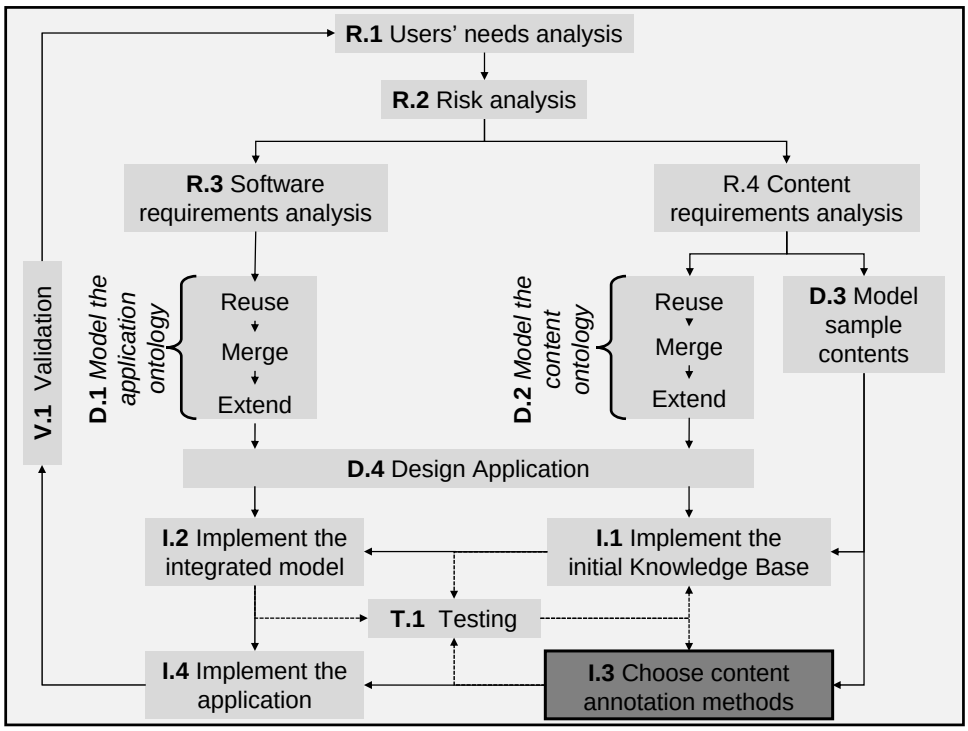


Testing the integrated model

- A simple test, which we can perform to verify the semantic soundness of all the ontologies we modelled, consists in loading in the model the example we produced (in step D.3) and extracting the entire content of the RDF storage in a single file using the `write` method

```
model.write("Dump.n3","N3");
```

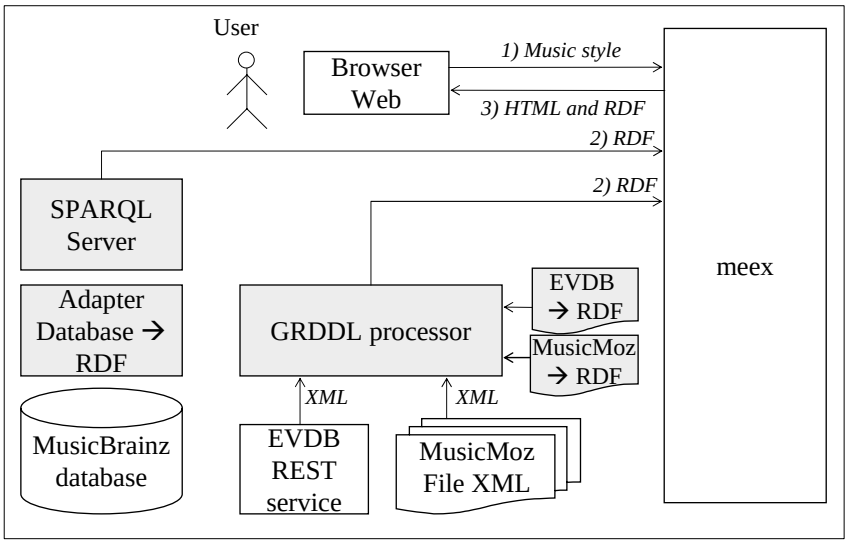
- If we open the file `Dump.n3` we can verify the presence of all the inferred triple we presented in slide 11



Choose content annotation methods

- Following the proposed approach, next step (i.e. I.3) suggests to choose content annotation methods
- The contents we choose for meex are already annotated at data source level, we (only) need to lift the data from XML or relational database as instances of the content ontology
- In the following slide we show how to implement and configure all the component necessary to allow meex to load data from the external data sources

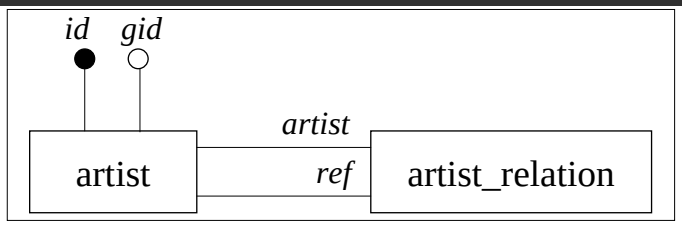
meex interfaces (1)



Importing annotations from MusicBrainz

- The annotations of MusicBrainz are stored as dump of PostgreSQL database
- So, first of all we install the relational database PostgreSQL
  - necessary documentation is available on PostgreSQL and MusicBrainz official websites
- When the database is available we need to install and configure
  1. a translator from relational database to RDF
  2. a SPARQL endpoint
- We choose **D2RQ** as translator and **Joseki** as SPARQL server

Configuring D2RQ for MusicBrainz (1)



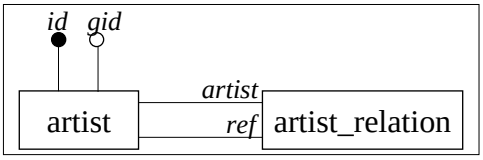

```
@prefix map: <http://swa.cefriel.it/meex/D2RQ-MusicBrainz.n3#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix d2rq: <http://www.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/0.1#>.
@prefix mb: <http://musicbrainz.org/> .

map:database a d2rq:Database;
  d2rq:jdbcDriver "org.postgresql.Driver";
  d2rq:jdbcDSN "jdbc:postgresql://localhost:5432/MusicBrainzDB";
  d2rq:username "postgres";
  d2rq:password "sw-book".

[more to follow]
```

D2RQ-MusicBrainzDB.n3

Configuring D2RQ for MusicBrainz (1)

```
[follows]

map:artist a d2rq:ClassMap;
  d2rq:dataStorage map:database;
  d2rq:class mb:Artist;
  d2rq:uriPattern "http://musicbrainz.org/artist/@@artist.gid@@.html";

map:artist_name a d2rq:PropertyBridge;
  d2rq:belongsToClassMap map:artist;
  d2rq:property rdfs:label;
  d2rq:column "artist.name".

map:artist_relation a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:artist;
 d2rq:property mb:artist_relation;
 d2rq:join "artist.id = artist_relation.artist“;
 d2rq:join "artist_relation.ref = artist2.id";
 d2rq:uriPattern "http://musicbrainz.org/artist/@@artist2.gid@@.html".
```

D2RQ-MusicBrainzDB.n3



Configuring Joseky for MusicBrainz

```
1. []  rdf:type            joseki:Service ;
       rdfs:label          "SPARQL for MusicBrainzDB" ;
       joseki:serviceRef   "MusicBrainz" ;
       joseki:dataset      _:MusicBrainzDS ;
       joseki:processor    joseki:ProcessorSPARQL_FixedDS .

2. _:MusicBrainzDS rdf:type ja:RDFDataset ;
       ja:defaultGraph _:MusicBrainzModel ;
       rdfs:label "MusicBrainz Dataset" .

3. _:MusicBrainzModel rdf:type d2rq:D2RQModel ;
       rdfs:label "MusicBrainz D2RQ Model" ;
       d2rq:mappingFile <file:D2RQ-MusicBrainzDB.n3> ;
       d2rq:resourceBaseURI <http://musicbrainz.org/> .
```

joseki-config.ttl

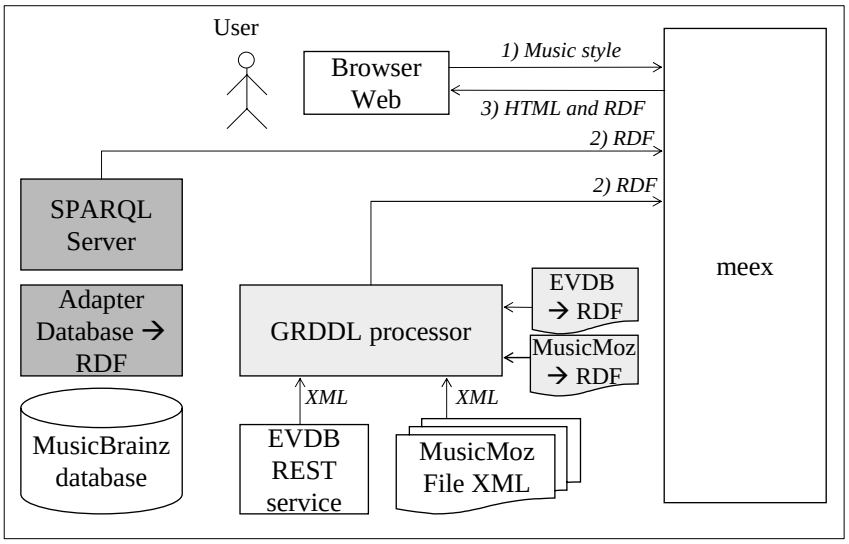
- With row 1 we expose a SPARQL endpoint giving the name of the service and the URL at which it will become accessible http://localhost:2020/MusicBrainz
- With row 2 and 3 we configure the SPARQL endpoint to expose MusicBrainz via D2RQ using the configuration file D2RQ-MusicBrainzDB.n3 (see previous slide)

Testing the SPARQL endpoint

```
1. String sparqlQueryString = "PREFIX mb: <http://musicbrainz.org/>\n“
                            + "DESCRIBE <" + artist + ">";

2. Query query = QueryFactory.create(sparqlQueryString);

3. QueryExecution qexec = QueryExecutionFactory.sparqlService
                        ("http://localhost:2020/MusicBrainz", query);

4. Model resultModel = qexec.execDescribe()
```

- We choose **ARQ** to test the MusicBrainz SPARQL endpoint submitting a `DESCRIBE` SPARQL query to obtain the description of an artist
- With row 1 we define the SPARQL query in which the variable `artist` contains the URI of the artist we want to be described
- With row 2 and 3 we instantiate a query model and we configure the `QueryExecution` to send the query to the endpoint at the URL http://localhost:2020/MusicBrainz
- With row 4 we execute the query and we obtain a Jena model as a result

meex interfaces (2)



Importing annotations from MusicMoz and EVDB

- The MasicBrainz SPARQL endpoint is ready, let's imporing annotations from MusicMoz and EVDB. They both exchange data in XML.
- In the design steps we chose to use a GRDDL processor to convert from XML in RDF (in the RDF/XML syntax)
- The GRDDL recommendation requires the XML documents to directly refer to the XSLT that performs the translation.
  - Neither MusicMoz nor EVDB XML files originally include the reference request by GRDDL
  - We can programmatically add it
    - In the following slide we show an excerpt of the modified XML files for MusicMoz
- We can proceed likewise for EVDB

Importing annotations from MusicMoz (1)

```
<musicmoz
 xmlns:grddl='http://www.w3.org/2003/g/data-view#‘
 grddl:transformation="file:///[...]/musicmoz-to-rdf.xsl">
 <category name="Bands_and_Artists/B/Beatles,_The“
           type="band">
  <resource name="musicbrainz"
            link="http://musicbrainz.org/artist/
                 b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d.html"/>
  <from>England</from>
  <style number="1">British Invasion</style>
  <style number="2">Rock</style>
  <style number="3">Skiffle</style>
 </category>
 <style><name>British Invasion</name></style>
 <style><name>Rock</name></style>
 <style><name>Skiffle</name></style>
</musicmoz>
```

Excerpts from the files musicmoz.bandsandartists.xml and musicmoz.lists.styles.xml

Importing annotations from MusicMoz (2)

```
<xsl:template match="musicmoz/category[(@type='band' or
                 @type='artist‘) and resource/@name='musicbrainz']">
 <xsl:variable name="artist_uri“
                     select="resource[@name='musicbrainz']/@link"/>
 <xsl:for-each select="style">
  <xsl:variable name="style_reformatted“
              select="concat('http://musicmoz.org/style/',text())"/>
   <rdf:Description rdf:about="{$artist_uri}">
     <mm:hasStyle rdf:resource="{$style_reformatted}"/>
   </rdf:Description>
 </xsl:for-each>
 <rdf:Description rdf:about="{$artist_uri}">
  <mm:from><xsl:value-of select="from"/></mm:from>
 </rdf:Description>
</xsl:template>
<xsl:template match="musicmoz/style">
 <xsl:variable name="style_reformatted"
               select="concat('http://musicmoz.org/style/', name)"/>
 <mm:Style rdf:about="{$style_reformatted}">
  <rdfs:label><xsl:value-of select="name"/></rdfs:label>
 </mm:Style>
</xsl:template>
```

Excerpts from the file musicmoz-to-rdf.xsl

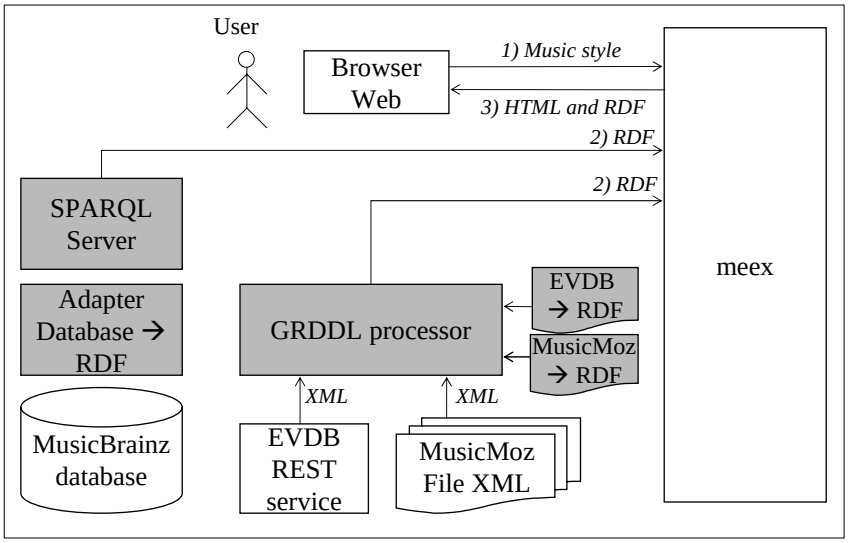
Importing annotations from MusicMoz (3)

- As GRDDL processor we choose **GRDDL Reader**, the GRDDL processor for Jena.

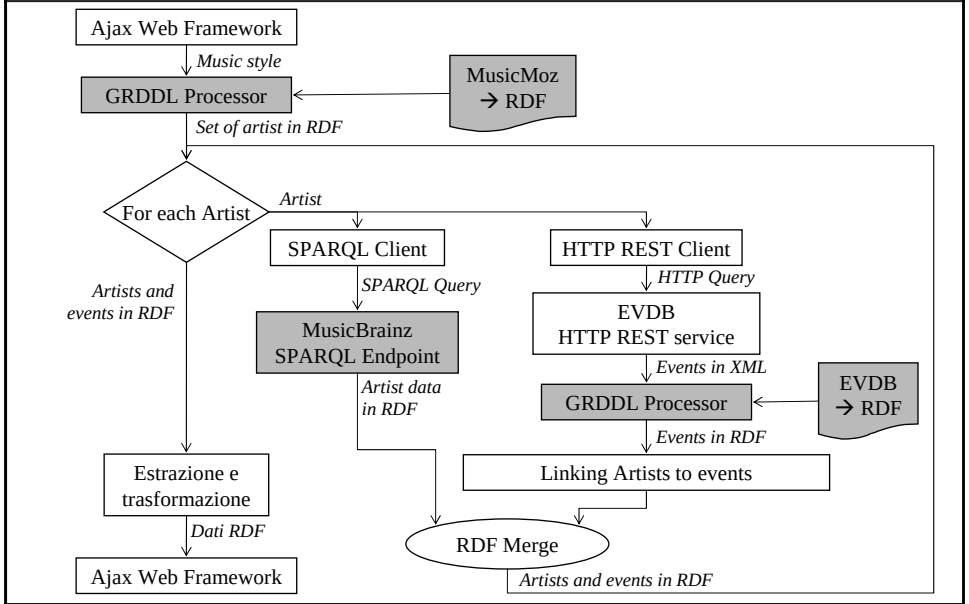
```
1. Model mmModel = ModelFactory.createDefaultModel();
2. RDFReader reader = mmModel.getReader("GRDDL");
3. reader.read(mmModel, "file:///.../musicmoz.bandsandartists.xml");
4. reader.read(mmModel, "file:///.../musicmoz.lists.styles.xml");
5. model.add(mmModel);
```

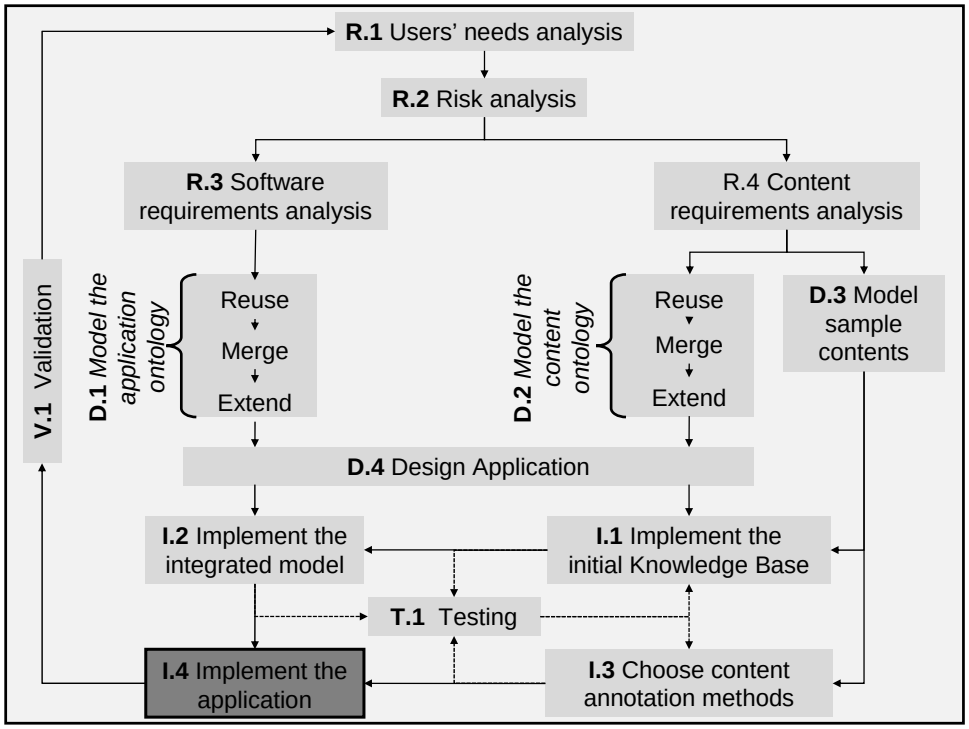
- With row 1 we instantiate a Jena model that will momentarily contain the RDF data produce by the GRDDL processor
- With row 2 we instantiate a `RDFReader` that uses a GRDDL processor to load RDF data
- With row 3 and 4 we load in the RDF model instantiate in row 1 the data contained in the XML files of MusicMoz using the RDF reader configured for GRDDL
- With row 5 we merge the loaded RDF data with those already present in the RDF storage

So far so good! (1)



So far so good! (2)



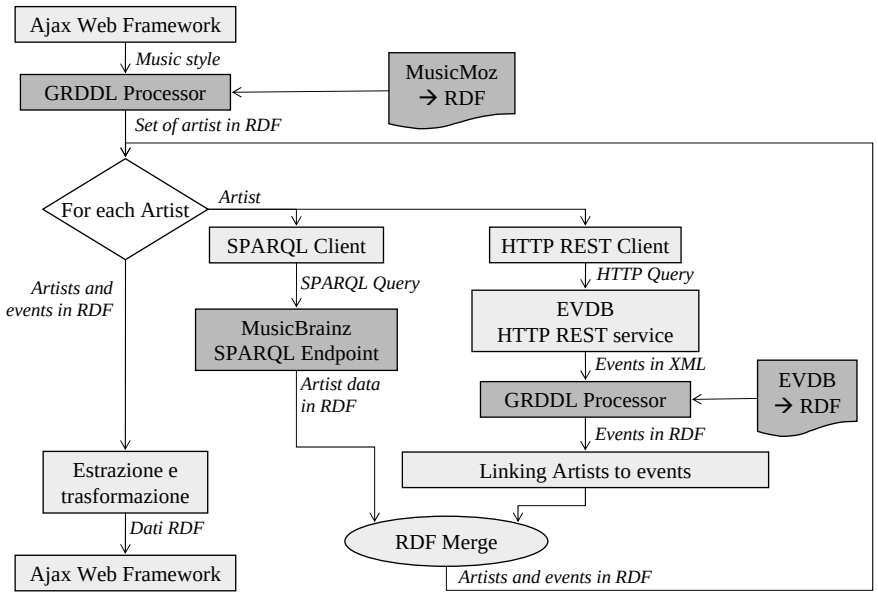


What's left?

- All the business logic that coordinates the interaction among the internal component is still to be implemented
- NOTE: implementing the business logic requires
  - both writing many lines of pure Java code
  - and work with several Semantic Web technologies

  we will focus our attention to the Semantic Web technologies
- The complete Java code is available on the website of our Semantic Web book for downloading.

What's left?



MEMO: Execution Semantics (1)

1. The user requests a music style
2. meex access the local copy of MusicMoz and using the GRDDL processors obtains a set of artist that plays the given music style

[more to follow]

Step 2: from the music style to the artists

- The step 2. of meex execution semantics requires to query MusicMoz for the artist that plays the music style requested by the users
- The following Java code shows how to encode the SPARQL query in terms of the application ontology

```
String sparqlQueryString =
      "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n"
          + "PREFIX meex: <http://swa.cefriel.it/meex#>\n"
          + "SELECT DISTINCT ?performer \n"
          + "WHERE { ?performer meex:performsStyle ?style.\n"
          + "        ?style rdfs:label \"" + style + "\".}";
```

MEMO: Execution Semantics (2)

[follows]

3. For each artist meex :
   a) uses the SPARQL client to query the MusicBrainz SPARQL endpoint and it obtains the artist name and his/her relationships with other artist
   b) invokes the EVDB REST service, it obtains the events that refer to the artist in XML and uses the GRDDL processor to obtain this data in RDF
   c) links the data about each artist to the data about the events that refers to him/her

[more to follow]

Step 3.a: querying MusicBrainz

- The step 3.a of meex execution semantics requires to query MusicBrainz for the data that describe an artist including the related artists

```
String sparqlQueryString =
    "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n”
  + "PREFIX mb: <http://musicbrainz.org/>\n”
  + "DESCRIBE <"+ artist + ">";

SPARQLClient sparqlClient = new SPARQLClient(null);

try {

  return sparqlClient.executeDescribeQuery(sparqlQueryString,
                                  Config.MusicBrainzSPARQLEndpoint);

} finally {

  sparqlClient.closeQuery();

}
```

Excerpts from the file MusicBrainz.java

Step 3.b: querying EVDB

- The step 3.b of meex execution semantics requires to invoke the EVDB REST service, obtain the list of events in XML and use the GRDDL processor to obtain the RDF

```
invokeHttpEndpoint(performerLabel, eventsFilename);
prepareForGRDDL(eventsFilename);
Model m = GRDDLProcessor.ApplyGRDDLTransformation(eventsFilename);
```

```
private static void invokeHttpEndpoint(String keywords,
                String outputFilename) throws IOException {
  URL url = new URL(
      "http://api.evdb.com/rest/events/atom?sort_order=relevance&"
    + "keywords=" + URLEncoder.encode(keywords, "UTF-8")
    + "&category=music&app_key="+Config.EVDBKey);
  URLConnection conn = url.openConnection();
  conn.setDoOutput(true);
  BufferedReader in = new BufferedReader(new InputStreamReader(
                      conn.getInputStream()));
  […]
  while ((inLine = in.readLine()) != null)
    writer.write(inLine + "\n");
}
```

Excerpts from the file EVDB.java

Step 3.c: linking artists to events

- The step 3.c of meex execution semantics requires to link the artist information retrieved from MusicMoz and MusicBrainz to the event information retrieved from EVDB
- We can use the following SPARQL `CONSTRUCT` query to create the links

```
String sparqlQueryString =
    "PREFIX meex: <http://swa.cefriel.it/meex#>\n"
  + "CONSTRUCT {<" + performer + "> meex:performsEvent ?event.}\n“
  + "WHERE {?event a meex:Event.}";
```

MEMO: Execution Semantics (3)

[follows]

4. When all the peaces of information about artists and events are available in the RDF storage, meex extracts them and serializes them in the format of the Ajax Web framework
5. The ajax Web framework allows the user for exploring the events found by meex
6. When the user decides to start a new exploration, meex starts over from the beginning

Step 4: preparing the data for the GUI

- We choose **Exhibit** as Ajax Web framework because
  - allows facet browsing
  - allows grouping and filtering events by
    - artist name
    - artist nationality
    - the style the artist plays
    - the related artists
  - includes different views
    - an ordered list
    - a chronological graph
    - a geographic map

Step 4: configuring Exhibit

- We can configure Exhibit by the means of two files:
  - an HTML page that controls the look and feel and
  - a JSON file that contains the data to be explored by the user
- In this tutorial we focus on the preparation of the JSON file. We refer to Exhibit documentation and the website of our Semantic Web book for the preparation of the HTML page of Exhibit for meex
- A JSON file is a simple text file that contains data organized in set of recors. In the following slide we show the information of The Beatles expressed in JSON.

Step 4: a sample JSON file

```
1. type:              "Event",
2. label:             "1964 The Tribute Tribute to Beatles",
3. eventful_link:     "http://eventful.com/events/
                                          E0-001-006129372-
   5",
4. when_startTime:    "2008-01-25",
5. when_endTime:      "2008-01-26",
6. where_label:       "Paramount Theater",
7. where_address:     "17 South Street, New York 10940,
                                              United
   States",
8. where_latlng:      "41.4544,-74.471",
9. performer_label:   "The Beatles",
10.fromCountry:        "England",
11.styles:            ["Skiffle", "British Invasion", "Rock"],
12.relatedPerformers:["The Beach Boys", "Eric Clapton"]
```

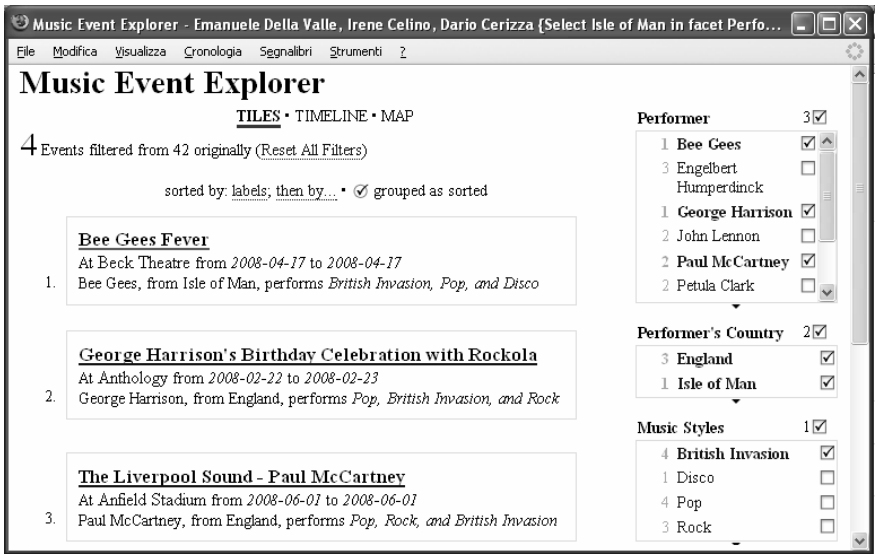
Step 4: serializing RDF in JSON

- In order to serialize RDF in JSON
  - we extract the information we loaded in the RDF storage using the SPARQL query shown in the following slide
  - we serialize the result in JSON
- NOTE: as we've already said several time, the query can be expressed in terms of the application ontology even if the data were loaded in other heterogeneous formats

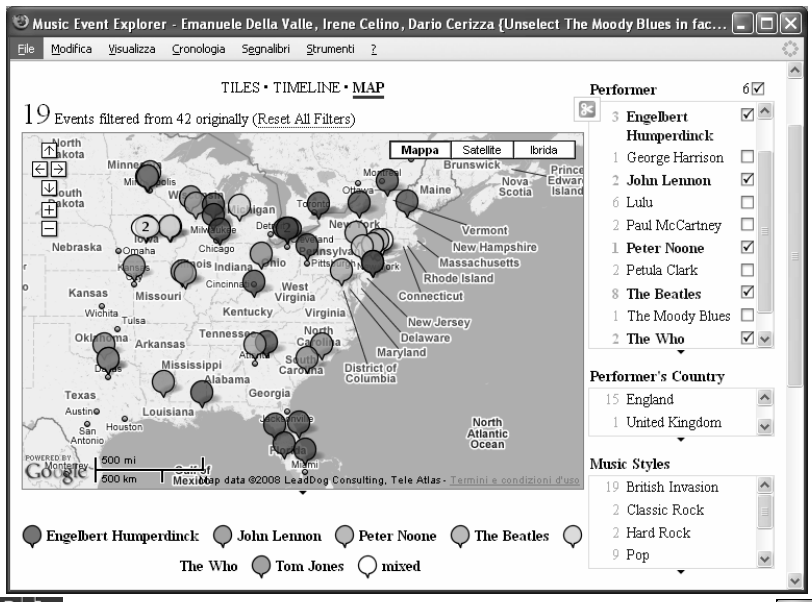
Step 4: extracting the data

```
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX meex:    <http://swa.cefriel.it/meex#>
PREFIX gd:      <http://schemas.google.com/g/2005>
SELECT DISTINCT ?event ?event_label ?when_startTime
   ?when_endTime ?where_label  ?where_address ?where_lat
   ?where_lon ?performer ?performer_label ?fromCountry
WHERE {
?event rdfs:label ?event_label;
       meex:hasWhen ?when;
       meex:hasWhere ?where.
?when gd:startTime ?when_startTime;
       gd:endTime ?when_endTime.
?where gd:label ?where_label;
       gd:postalAddress ?where_address;
       gd:hasGeoPt ?geoPt.
?geoPt gd:lat ?where_lat;
       gd:lon ?where_lon.
?performer meex:performsEvent ?event;
       rdfs:label ?performer_label;
       meex:fromCountry ?fromCountry.}
```

Step 5 and 6



Step 5 and 6



Tools employed (1)

- Jena
  - Application Framework
  - http://jena.sourceforge.net
- Derby
  - Relational database for the RDF storage
  - http://db.apache.org/derby
- PostgreSQL
  - Relational database for MusicBrainz
  - http://www.postgresql.org
- D2RQ
  - Translator from relational database to RDF
  - http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2rq

Tools employed (2)

- Joseki
  - SPARQL Endpoint Server
  - http://www.joseki.org
- ARQ
  - SPARQL query engine for Jena
  - http://jena.sourceforge.net/ARQ
- GRDDL Reader
  - GRDDL processor
  - http://jena.sourceforge.net/grddl
- Exhibit
  - Ajax Web Framework
  - http://static.simile.mit.edu/exhibit

Thank you for paying attention

Realizing a Semantic Web Application

Emanuele Della Valle

Center of **E**xcellence **F**or **R**esearch, **I**nnovation, **E**ducation and industrial **L**ab partnership - Politecnico di Milano

http://www.cefriel.it
http://swa.cefriel.it
emanuele.dellavalle@cefriel.it
http://emanueledellavalle.org

Credits and Links

- CREDITS
  - Dario Cerizza [dario.cerizza@cefriel.it]
    - who help in concieving, designed and developed meex
  - Irene Celino [irene.celino@cefriel.it]
    - who help in concieving and support the design and development of meex
  - All the people involved in CEFRIEL's Semantic Web Activities
- Links
  - Visit http://swa.cefriel.it
  - Try http://swa.cefriel.it/Squiggle
  - Try http://swa.cefriel.it/SOIP-F