DAQUINO NOTES

XML

Stands for **eXtensible Markup Language**. It is a way to **annotate** texts in a **machine-readable** way. It is an international standard for data **exchange** across applications (e.g. between libraries, museums).

Not a full Markup Language, rather it is a **meta-markup language**, meaning that it provides you with a syntax, but how to annotate texts **is up to you**.

You must top up XML with a **schema**. There are many schemas that one can use. Schemas address elements, attributes, hierarchy, order, and allowed repetitions of elements.

Building blocks: **elements** with opening tag and closing tags (with a slash).

Value of the element: the text included between the two tags. The element can include both a text or other elements!

Hierarchy and Nesting: There is always a root node. The root element can be whatever you want. In html, for example, is <html>. Structure is hierarchical, like a tree with nodes. Children nodes can include more nodes or texts and can have siblings. This hierarchy is represented with indented blocks, that are not mandatory, but help with readability. Overlap is the biggest problem. XML is not kind if you forget something.

Milestones: empty elements that act as a placeholders, with no closing tag. The opening tag, tho, ends with / (<open/>).

Attributes: can be used to replace elements and record information. When mentioning an attribute in a text we write @attributeName, to distinguish them from element names. They appear only in the opening tag and are in the form `attributeName="value(s)".` Multiple attributes can appear in an element (or none). **No punctuation signs** should be used to separate values in an attribute value. The value of attributes **may be unique** in the document (e.g. @id includes a value that identifies the element at hand and distinguishes it from other similar elements in the document).

Element and attribute names are **case sensitive.**

XSD Schemas: Schemas (or XSDs) are XML documents. Elements can be defined as complex (when they include children), can include a specific sequence of elements. For each child, we must specify the expected value (e.g. a string). A document that respects the schema rules is called valid.

DTDs: A variant of schemas is DTD (Document Type Definition). It does the same job, but it uses a different syntax to express rules. To declare which DTD is used in your XML document, use the DOCTYPE declaration to specify the root element and the local DTD. A document that respects the DTD rules is called valid.

Prolog: `<?xml version= "1.0" eoncoding="UTF-8"?>`

Well-formed: A XML file w/ correct syntax is called well-formed. You can use a XML Validator.

TEI

It is both a **schema** to annotate literary texts and a **community** of scholars (mainly philologists). The website includes a set of guidelines on how to use the XSD by thematic areas and examples. The first version of the guidelines was released in 1990 (currently v4.6). The reference guidelines are called P5 Guidelines.

Module name	Formal public identifier
analysis	Analysis and Interpretation
certainty	Certainty and Uncertainty
core	Common Core
corpus	Metadata for Language Corpora
dictionaries	Print Dictionaries
drama	Performance Texts
figures	Tables, Formulae, Figures
gaiji	Character and Glyph Documentation
header	Common Metadata
iso-fs	Feature Structures
linking	Linking, Segmentation, and Alignment
msdescription	Manuscript Description
namesdates	Names, Dates, People, and Places
nets	Graphs, Networks, and Trees
spoken	Transcribed Speech
tagdocs	Documentation Elements
tei	TEI Infrastructure
textcrit	Text Criticism
textstructure	Default Text Structure
transcr	Transcription of Primary Sources
verse	Verse

The schema is divided in (thematic) **modules**, i.e. groupings of XML elements and attributes (~500 elements in total). Many modules can be used in the same document. The **core** module includes those elements that are likely to be used to describe any XML document.

@xml:id	(identifier) provides a unique identifier for the element bearing the attribute.
@n	(number) gives a number (or other label) for an element, which is not necessarily unique within the document.
@xml:lang	(language) indicates the language of the element content using a 'tag' generated according to BCP 47.
@rend [att.global.rendition]	(rendition) indicates how the element in question was rendered or presented in the source text.
@style [att.global.rendition]	contains an expression in some formal style definition language which defines the rendering or presentation used for this element in the source text
@rendition [att.global.rendition]	points to a description of the rendering or presentation used for this element in the source text.
@xml:base	provides a base URI reference with which applications can resolve relative URI references into absolute URI references.
@xml:space	signals an intention about how white space should be managed by applications.
@source [att.global.source]	specifies the source from which some aspect of this element is drawn.
@cert [att.global.responsibility]	(certainty) signifies the degree of certainty associated with the intervention or interpretation.
@resp [att.global.responsibility]	(responsible party) indicates the agency responsible for the intervention or interpretation, for example an editor or transcriber.

Global attributes: they can be used with any element (while non-global attributes can be used only with specific sets of attributes).

```
1 <TEI xmlns="http://www.tei-c.org/ns/1.0">
2  <teiHeader xml:lang="en">
3  <!-- ... -->
4  </teiHeader>
5  <text xml:lang="en">
6  <!-- ... -->
7  </text>
8 </TEI>
```

Mandatory elements: -root element with the ns declaration <TEI xmlns= "http.//www.tei-c.org/ns/1.0">

-the element teiHeader that includes all the metadata about the digital edition of the text

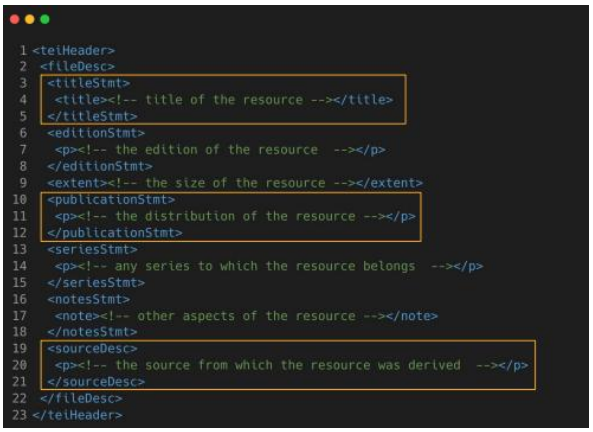
-the text element

When annotating multiple texts: the root element will be teiCorpus (not TEI) with as many teiHeader as the number of the documents.

SUGGESTED ELEMENTS BY THE PROF

The professor suggests those elements to use, but if you need more specific ones, add them:

- **teiHeader** includes the metadata of the digital edition, such as people's responsibilities, the bibliographic citation, and the strategies used to encode the text (e.g. naming conventions, critical choices in the transcription). Only fileDesc is mandatory.
    - **fileDesc** describes the current electronic file, with the exception of sourceDesc, which describes the original source. titleStmt, publicationStmt and sourceDesc are mandatory.



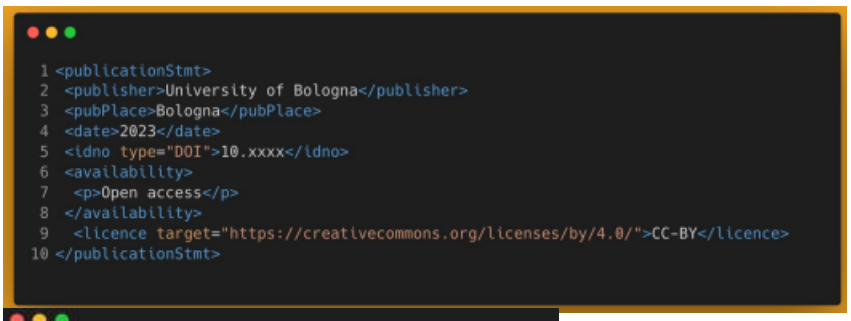
```
<teiHeader>
 <fileDesc>
  <titleStmt>
   <title><!-- title of the resource --></title>
  </titleStmt>
  <editionStmt>
   <p><!-- the edition of the resource  --></p>
  </editionStmt>
  <extent><!-- the size of the resource --></extent>
  <publicationStmt>
   <p><!-- the distribution of the resource --></p>
  </publicationStmt>
  <seriesStmt>
   <p><!-- any series to which the resource belongs  --></p>
  </seriesStmt>
  <notesStmt>
   <note><!-- other aspects of the resource --></note>
  </notesStmt>
  <sourceDesc>
   <p><!-- the source from which the resource was derived  --></p>
  </sourceDesc>
 </fileDesc>
</teiHeader>
```

- **titleStmt** includes the titile, can include the author and all the people that contributed to the digital edition - each recorded in respStmt, including the role (resp) and the name of the person.
- **editionStmt** includes information about the current edition of the digital text (not the edition of the original text). Edition includes a string and/or children elements. respStmt can be used to record people and roles.

```
<!-- a string -->
<extent>between 1 and 2 Mb</extent>

<!-- or one or more measurements -->
<extent>
 <measure unit="MiB" quantity="2">About 2 megabytes</measure>
 <measure unit="pages" quantity="1">1 page of source material</measure>
</extent>
```

Extent describes the extent of the digital file, e.g. in terms of MB, GB or other units of measures. Notice the usage of the element measure and its attributes.



```
<publicationStmt>
 <publisher>University of Bologna</publisher>
 <pubPlace>Bologna</pubPlace>
 <date>2023</date>
 <idno type="DOI">10.xxxx</idno>
 <availability>
  <p>Open access</p>
 </availability>
  <licence target="https://creativecommons.org/licenses/by/4.0/">CC-BY</licence>
</publicationStmt>
```

- **publicationStmt** includes information about the publisher of the digital edition and the rights/licenses under which the resource is available. NB. The digital edition of a text that is under copyright cannot be open access

```
<sourceDesc>
  <bibl>
    <title level="a">Resistance</title>. In
    <author>Muse</author>,
    <title level="b">Resistance</title>.
    <date>2009</date>.
  </bibl>
</sourceDesc>
```

- **sourceDesc** includes the bibliographic reference of the original source. Details can include authors, title, dates, publisher, etc

```
<encodingDesc>
  <projectDesc>
    <p><!-- The purpose of the digital resource --></p>
  </projectDesc>
  <editorialDecl>
    <correction>
      <p><!-- If the text includes corrections--></p>
    </correction>
    <normalization>
      <p><!-- If any changes happened to uniform ecitorial choices --></p>
    </normalization>
    <interpretation><!-- If any analytical feature was added to the text--></interpretation>
  </editorialDecl>
</encodingDesc>
```

- **encodingDesc** specifies the methods and editorial principles which guided the transcription or encoding of the text, such as corrections (if any error was corrected), normalisation strategies (e.g. en-us spelling), or interpretations (e.g. if you add elements of linguistic analysis).

```xml
<TEI xmlns="http://www.tei-c.org/ns/1.0">
 <teiHeader>
    <!-- ... -->
 </teiHeader>
 <text>
  <front>
    <!-- front matter of copy text, if any, goes here -->
  </front>
  <body>
    <!-- body of copy text goes here -->
  </body>
  <back>
    <!-- back matter of copy text, if any, goes here -->
  </back>
 </text>
</TEI>
```

▪ <**text**> the transcription of the text. It may include a front matter , a text body and a back matter <back>.

ELEMENTS IN ALL TEI DOCS

```xml
<p>This is a paragraph mentioning
  a <foreign xml:lang="fr">croissant</foreign>,
  a <hi rend="italic">highlight</hi>,
  and some <emph>important concept</emph>.
</p>

<q>It is followed by a quotation from one of
 your favourite authors</q>
```

Some elements can be used in any TEI document regardless of the type/genre of literary text. These include: paragraphs, quotations , graphically highlighted terms: , foreign terms: , emphasis , etc…

```xml
<body>
 <div type="part" n="1">
    <div type="chapter" n="1">
        <!-- text of part 1, chapter 1 -->
    </div>
    <div type="chapter" n="2">
        <!-- text of part 1, chapter 2 -->
    </div>
 </div>
 <div type="part" n="2">
  ...
 </div>
</body>
```

** :** A prose text can be divided in several structures, e.g. parts, chapters, sections. In this case divisions are represented with the element

and the attributes @type and @num further specify their peculiarities and differentiate them. Values are free from naming conventions.

```xml
<div n="19" type="chap">
 <head rend="bold" type="main">Chapter 19</head>
 <p>To say that Deronda was romantic would be to
   misrepresent him: but under his calm and
   somewhat self-repressed exterior ...</p>
</div>
```

<headings>: Inside a division, titles of all levels can be represented with and further specified via the attributes @rend, @type and more.

```xml
<list rend="numbered">
 <item>a butcher</item>
 <item>a baker</item>
 <item>a candlestick maker, with
 <list rend="bulleted">
   <item>rings on his
fingers and bells on his toes</item>
  </list>
 </item>
</list>
```

<**list**>: Bullet lists are represented via the element and its children . Lists can be ordered or unordered. Items can include other lists.

```xml
<gap reason="illegible" unit="word" quantity="2"
  resp="#editor04"/>
<!-- notice the attribute resp -->

<l>
 <del rend="overstrike">Inviolable</del>
 <add place="below">Inexplicable</add>
 splendour of Corinthian white and gold
</l>
```

Deletions, **corrections**: Omissions, changes (deletions and additions), as well as unclear texts are recorded in elements like (words omitted), <del> and <add>.

```xml
<div type="letter">
 <opener>
  <dateline>
   <placeName>Newport</placeName>
   <date when="1761-05-27">May ye 27th 1761</date>
  </dateline>
  <salute>Gentlemen</salute>
 </opener>
 <p>Capt Stoddard's Business
<lb/>calling him to Providence, have
<lb/>got him to look at Hopkins brigantine
<lb/>&amp;amp; if can agree to Purchase her, shall
<lb/>be much oblig'd for your further
<lb/>assistance herein, &amp;amp; will acquiesce with
<lb/>whatever you &amp;amp; he shall Contract
<lb/>for — I Thank you for your
<lb/>
  <unclear>Line</unclear> respecting the brigantine &amp;amp;
Beg<lb/>leave to Recommend the Bearer
<lb/>to you for your advice &amp;amp; Friendship
<lb/>in this matter</p>
 <closer>
  <salute>I am your most humble servant</salute>
  <signed>Joseph Wanton Jr</signed>
 </closer>
 <postscript>
  <label>P.S.</label>
  <p>I have Mollases, Sugar,
  <lb/>Coffee &amp;amp; Rum, which
  <lb/>will Exchange with you
  <lb/>for Candles or Oyl</p>
 </postscript>
</div>
```

Letters: Letters include peculiar logical elements such as <opener> and <closer> and <postscript>. Line breaks in a paragraph can be recorded via <lb>, a milestone element written in the short version. In the example, also notice the element <unclear> to mark words difficult to read.

```xml
<text>
 <body>
   <lg type="quatrain">
    <l>My Mistres eyes are nothing like the Sunne,</l>
    <l>Currall is farre more red, then her lips red</l>
    <l>If snow be white, why then her brests are dun:</l>
    <l>If haires be wiers, black wiers grown on her head:</l>
   </lg>
    ...
   <lg type="couplet">
    <l>And yet by heaven I think my love as rare,</l>
    <l>As any she bell'd with false compare.</l>
   </lg>
 </body>
</text>
```

Verses: Poems can be characterized by ungrouped lines (element <l>). When grouped, lines are included in the element <lg>. The attribute @type can specify the structure or unit, e.g. a sonnet or a stanza.

```xml
<body>
 <div1 type="act" n="1">
  <head>Act One</head>
  <div2 type="scene" n="1">
   <stage>Pa Ubu, Ma Ubu</stage>
   <sp>
    <speaker>Pa Ubu</speaker>
    <p>Pschitt!</p>
   </sp>
  </div2>
  <div2 type="scene" n="2">
   <stage>A room in Pa Ubu's house, where a magnificent
       collation is set out</stage>
  </div2>
 </div1>
 <div1 type="act" n="2">
  <head>Act Two</head>
  <div2 type="scene" n="1">
   <head>Scene One</head>
  </div2>
  <div2 type="scene" n="2">
   <head>Scene Two</head>
  </div2>
 </div1>
</body>
```

Performances: Performance texts may include descriptions of the stage and the sequence of speakers' dialogues. Here like in other situations, the hierarchy of text divisions can be further specified by using the elements <div1> and <div2> (not mandatory).

VALIDATION

There are several ways to validate a XML/TEI document

- Online validator: tells you whether the schema is respected (valid) and the XML is well-formed

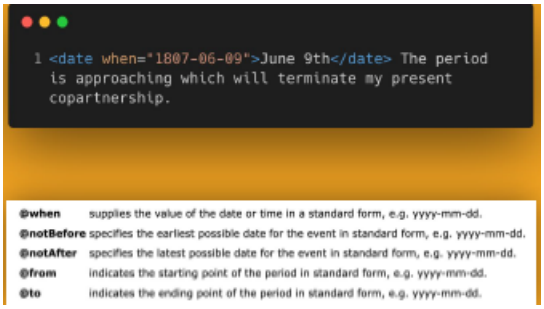
- XML/TEI plugin for VS Code: tells you whether the XML is valid, well-formed, and suggests TEI elements

- Python library: a shell script to validate XML/TEI docs.

NAMED ENTITIES

So far we have seen elements representing **metadata** and the **logical structure** of a text. TEI also provides elements to record **occurrences of real-world entities** (or abstract concepts) in the text. See the guidelines.

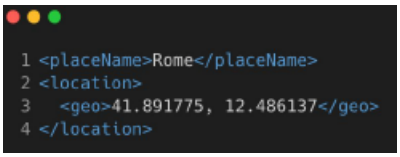
Why is this useful? **indexes** of annotated entities can be created automatically (people, dates, places, etc.) and XML documents can be interrogated with appropriate languages to return lists of entities, their occurrence, etc.



-**Dates** are often referenced in natural language. Nonetheless, a machine readable version of thedate can be recorded via the element <date> andits attribute @when, using a standard formatyyyy-mm-dd. If the date is uncertain, other attributes can be used to frame the period.

While an element <**name**> exists and can be used to annotate any proper name, <persName> specifies the name belongs to a person. It accepts several children elements, such as <forename>, <surname> and <roleName> to further distinguish substrings.

Similarly to people, **organisations** can be annotated via <orgName> and substrings can be further defined.



Places have dedicated elements to record names <placeName>, location and coordinates <geo>.

In a text, several different strings may refer to the same real-world entity. To **preservethe relation between occurrences and the related entity**, identifiers can be used to bind them.

```
1 We went on holiday in <placeName key="Rome">Rome</placeName>
2 ...
3 Despite it is a beautiful <placeName key="Rome">city</placeName>, it
  is rather chaotic.
```

The attribute **@key** is used to record an externally-defined string identifying the referent (e.g. the english name of a place in an Atlas).

Projects may use @**key** to refer to the same terms or not, therefore the attribute @ref or @**sameAs** are preferred instead. @ref records one or more URLs of web resources describing the entity at end, e.g.Wikipedia entries.@**sameAs** records URIs identifying the entity.

URI: stands for Uniform Resource Identifier. Similar to URL (Uniform Resource Locator), URIs identify something. A URL identifies a web resource, while a URI identifies a real-world entity (or concept), which can be described in several web resources (in turn identified by URLs). A **URL** tells us where a web resource is located (i.e. on which server) but not what it is about. A URI tells us what the identifier is about (e.g. a person) and where it is located.

AUTHORITY CONTROL LISTS: In some cases, a URI identifies both a real-world entity (e.g. a place) and a web resource describing that entity. Authority control files mint such type of URIs so that projects can univocally refer to those to identify the entities, regardless of language, homonyms, etc. Some well-known authorities are VIAF, Wikidata, geonames, Getty vocabularies (AAT, ULAN, TGN).

Why do authorities matter?

Keeping track of identifiers provided by central authorities in your data is fundamental to further process data in future scenarios.

1. **Facilitate data exchange.** If you merge your edition with that of others, it will be easier to recognise which people, places, etc. are shared across editions.

2. **Enable data enrichment.** You can query central authorities to get more information about those entities (e.g. biographies) and integrate that information in your data or application (e.g. a website).

Local Authority files

Recording URIs everytime an occurrence of a name appears is cumbersome and time-consuming. A good practice is to create a local authority file in the <teiHeader> of the very same XML file of your edition (or in another one).

```
1 <teiHeader>
2   ...
3   <profileDesc>
4       <particDesc>
5         <listPerson type="cited-author">
6           <person xml:id="MG" sameAs="http://viaf.org/viaf/97859168
7                                       http://it.dbpedia.org/resource/Michail_Gorbačëv">
8             <persName xml:lang="it">Michail Gorbačev</persName>
9             <persName xml:lang="en">Michail Gorbačev</persName>
10            <birth when="1931"/>
11          </person>
12        </listPerson>
13      </particDesc>
14  <profileDesc>
15  ...
16 </teiHeader>
17 <text>
18    ...
19    <p>[Crisi del comunismo, <lb/>
20      <persName ref="#MG">Gorbaciov</persName> [<persName ref="#GGC">Cesare</persName>]]
21    </p>
22    ...
23  </text>
```

Authority lists can be created for:

- People (listPerson),

- Places (listPlace),

- Organisations (listOrg),

- Events (listEvent),

- Relations (listRelations),

- Bibliographic References (listBibl)

FACSIMILE

Usually, texts encoded in XML/TEI are the result of a transcription, either from an original analog source or from a digital version of it. A **digital facsimile** is a collection of images, usually 1 for each page, of the source and their metadata.

Pagination: When transcribing the text, the editor keeps track of the original pagination using the milestone element <pb> to identify the page breaks. The attribute @facs records the path to the file representing the page at hand.

Manage images: Usually images are served along with the files of the digital edition of the text. When working **locally**, images are saved in the same folder of the XML files. Similarly, when **publishing online** a digital edition, images are often stored together with the HTML of the text. Sometimes, images are stored in **institutional repositories**, maybe more than one, they are in different formats, shared with different licenses, and cannot be re-published by third parties (e.g. in my digital edition).

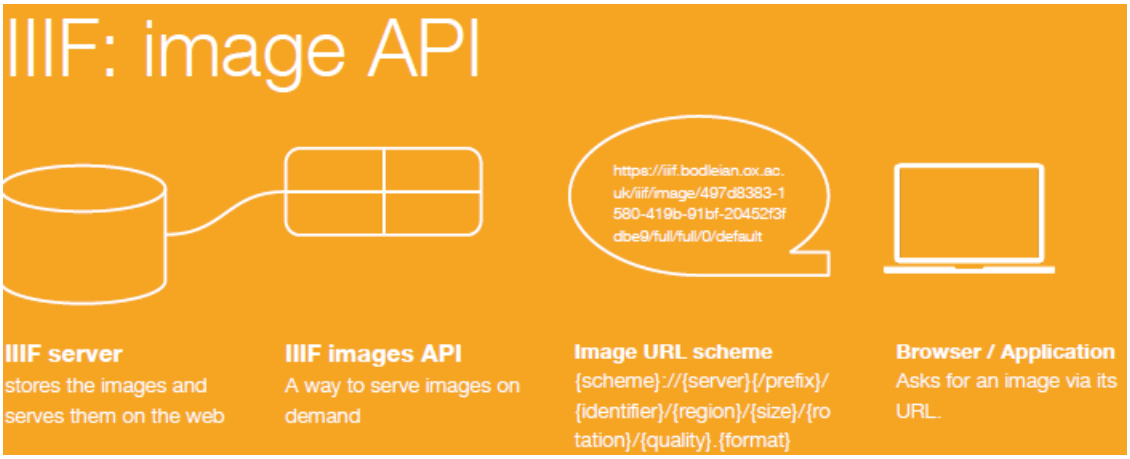
https://	{scheme}://
stacks.stanford.edu/	{server}
image/iiif/	{/prefix}/
hd288rd1737%252F0090_496-34/	{identifier}/
full/	{region}/
full/	{size}/
0/	{rotation}/
default	{quality}
.jpg	.{format}

IIF: is a standard that universalizes the way archived images are described and retrieved on the web. It is the "definitive solution" to publish images via dedicated server applications to store and serve **images with metadata** and that allow users to access **advanced features** when browsing (e.g. cropping, zooming, collate images, cite, annotate, access metadata). It is a **digital repository**. IIIF community has created a few server applications to store images and their metadata, according to IIIF specifications. Here a list of IIIF servers: https://iiif.io/get-started/image-servers/

The **image API** (Application Programming Interface) is a program that interacts with the IIIF server and serves images to users or external applications that request the image URL. The image URL is composed of a few parameters that can be modified by the user/application, e.g.

{scheme}://{server}{/prefix}/{identifier}/{region}/{size}/{rotation}/{quality}.{format}

https://stacks.stanford.edu/image/iiif/hd288rd1737%252F0090_496-34/full/full/0/default.jpg



The user application asks for the URL to the server, which communicates with its API, reads the parameters in the URL (e.g. quality, format) and returns the image file.

The image API can also return a **JSON-LD file**

including technical information about the

image.

{scheme}://{server}{/prefix}/{identifier}/info.json

JavaScript Object Notation (JSON) is a lightweight

format for data-interchange. Like XML, JSON is a

standard language for **exchanging data on the web**, which is meant to be easy to read for both humans and machines, and that it is not meant to be visualised on the web (like e.g. HTML). **JSON-LD** It's a particular type of JSON. The key **@context** includes a reference to an online schema with rules on which keys and types of values to include in the file (like XSD and DTD would work for an XML file). The context.json file is yet another JSON file. PS/ In the next classes we will see that JSON-LD is a the JSON syntax to record graph data.

Another API that packages images and metadata, so users know the origin, title of the image, and even what page of a book the image was from, likely in conjunction with the Image API. In particular, IIIF images come with manifests, that is, other JSON files including structured information about an object's metadata (e.g. title, author, date of a book), how to access the image(s) within the object, and the order in which they should appear.

{scheme}://{host}/{prefix}/{identifier}/manifest

A manifest.json file provides:

- descriptive information about an object, e.g. a book,

- the sequence (i.e. the list) of objects that compose the main object and in which order these

should be visualised by a browsing application

- the canvases, like a page or view

- the content actually displayed in the canvas, and optionally the annotations attached to it

There are many IIIF viewers. IIIF-compatible viewers generally allow users to **zoom, rotate, and resize image** objects, and play audio/visual files. Some allow **annotation** with text, audio, location, and more. Others allow **comparison** of objects from a single collection side-by-side (or even objects from multiple collections if the object's Manifest is made available to users). **The manifest tells viewer applications how to display content and in which order. The image API returns the actual image.**

CRITICAL APPARATUS

Scholarly editions of texts, especially texts of great antiquity or importance, often record some or all of the known variations among different witnesses to the text. Information about variant readings (whether or not represented by a critical apparatus in the source text) may be recorded in a series of apparatus entries, each entry documenting one variation, or set of readings, in the text.

```
<listWit>
 <witness xml:id="El">Ellesmere, Huntingdon Library 26.C.9</witness>
 <witness xml:id="Hg">Hengwrt, National Library of Wales,
   Aberystwyth, Peniarth 392D</witness>
 <witness xml:id="Ra2">Bodleian Library Rawlinson Poetic 149
   (see further <ptr target="http://example.com/msDescs#MSRP149"/>)</witness>
</listWit>
```

List of witnesses: Information about witnesses is supplied by means of a <listWit> element, including as many <witness> elements as the number of witnesses, each identified with a @xml:id attribute.

```
<app>
 <rdg wit="#El">Experience though noon Auctoritee</rdg>
 <rdg wit="#La">Experiment thouh noon Auctoritee</rdg>
 <rdg wit="#Ra2">Eryment though none auctorite</rdg>
</app>
```

Apparatus Entries: Whenever variant readings of a word or textchunks must be recorded, the element <app> isused to wrap the readings. The attribute @wit records the identifier of the text wherein thevariant reading appears. @resp and @source record the responsible person, and @cert the level of certainty.

```
<app>
 <lem wit="#El #Hg">Experience</lem>
 <rdg wit="#La" type="substantive">Experiment</rdg>
 <rdg wit="#Ra2" type="substantive">Eryment</rdg>
</app>
```

Readings: The app element must include at least one <rdg> element. The <lem> element may be used to record the base text of the source edition/witness, to indicate the preference of an editor or encoder for a particular reading, or (e.g. in the case of an external apparatus) to indicate precisely to which portion of the main text the variation applies.

How to annotate apparatus

Three different methods may be used to link a critical apparatus to the text:

- the location-referenced method

- the double-end-point-attached method, and

- the parallel segmentation method

All methods can be used inline, meaning variants are recorded in place and are scattered around the base text, or external, that is, all apparatus entries are included in an external file and a linking method is provided to reference text fragments.

```
<variantEncoding method="location-referenced" location="external"/>
```

The method for encoding the apparatus must be recorded in the <encodingDesc> element.

Location referenced external

The external apparatus references ids of elements in the base text and records variants in other witnesses.

Location-referenced internal

```
1 <l n="1">Experience
2 <app>
3   <rdg wit="#La">Experiment</rdg>
4   <rdg wit="#Ra2">Eryment</rdg>
5  </app>
6  though noon Auctoritee</l>
7 <l>Were in this world ...</l>
```

The apparatus is recorded inline in the base text. Again, only variants in other witnesses are recorded. This method, like the previous one, does not uniquely identify the text fragment to which the readings refer to. The annotated portion can be made explicit by using the <lem> element and the witness of the base text is referenced too. This works well for single words though.

Double-end-point-attached

Double end-point attachment permits unambiguous matching of each variant reading against its lemma, since both the beginning and the end of the annotated lemma are marked and unequivocally identified with @xml:id. Like the location-referenced method, it can be encoded in the base text.

```
1 <l n="1" xml:id="WBP.1">Experience<anchor xml:id="WBP-A2"/> though noon Auctoritee</l>
```

Double-end-point-attached external

```
1 <app from="#WBP.1" to="#WBP-A2">
2  <rdg wit="#La">Experiment</rdg>
3  <rdg wit="#Ra2">Eryment</rdg>
4 </app>
```

The apparatus uses the attributes @from and @to to record the ids of elements delimiting the lemma. The lemma runs from the beginning of the element indicated by @from, to the end of that indicated by @to. If no value is given for @to, the lemma runs from the beginning to the end of the element indicated by the @from attribute.

Double-end-point-attached inline

If the apparatus is encoded internally, there is no need to repeat the lemma. Otherwise, it should be made explicit.

Parallel segmentation

```
<variantEncoding method="parallel-segmentation"
 location="internal"/>
<!-- ... -->
<l n="1">
 <app>
  <lem wit="#El #Hg">Experience</lem>
  <rdg wit="#La">Experiment</rdg>
  <rdg wit="#Ra2">Eryment</rdg>
 </app> though noon Auctoritee
</l>
<l>Were in this world ...</l>
```

When the base text reading is recorded in an apparatus reading (and not in the transcription), the method is called parallel segmentation. It must be used only internally, but can be translated into the double end-point attachment method without loss of information. It does not handle well overlapping variants.

MARKUP OVERLAP

Markup overlap is a well-known problem in the XML (and TEI) community.

- Different (logical, metrical, linguistic, content) structures may apply to the same portion of text (e.g. a <persName> split across multiple verse lines <l>).

- Texts with a complex tradition require to annotate small chunks of text for which not only multiple interpretations exist, but interpretations tend to encompass overlapping chunks.

(e.g. “textA textB textC” has 2 rdgs for the chunks “textA textB” and “ textB textC”)

Several **solutions** have been proposed, among which some that have already been described:

- The usage of milestone elements (e.g. <anchor>) to frame text chunks

- The usage of dedicated elements (e.g. listPerson, standOff) to record other information

- The usage of external files to record annotations (stand-off markup)

Some solutions include non-XML languages to record and link annotations. Notably, when annotations are recorded in an external document, the latter does not include both text and annotations, but only references to the original text. Therefore, stand-off markup requires mechanisms to identify XML snippets and link them into another document. Such mechanisms are used in a plethora of scenarios that go beyond standoff markup, e.g. query XML documents to retrieve pieces of information, transform XML-like documents into other XML-like documents, etc.

Some standards part of the XML-family contribute to these tasks:

- XInclude, XLink: standards to link documents

- XPointer, **XPath**: syntaxes to identify XML snippets

- **XSLT**: transform XML-like documents into other XML-like documents

- **XQuery**: query and transform XML documents

[slide 5 continua con HTML e XML]

[slide 6 sono riguardanti la text analysis, non obbligatoria per l’esame]

XINCLUDE

XInclude is a standard that specifies a syntax for the inclusion within an XML document of data fragments placed in different resources (e.g. other XML files or text files). It defines a namespace (xi:http://www.w3.org/2001/XInclude), and two elements, <xi:include> and <xi:fallback>.

```xml
<?xml version="1.0"?>
<book xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="frontmatter.xml"/>
  <xi:include href="part1.xml"/>
  <xi:include href="part2.xml"/>
  <xi:include href="part3.xml"/>
  <xi:include href="backmatter.xml"/>
</book>
```

The element <**xi:include**> uses the attribute @href to reference a XML document (or part of it) to be included in a second XML file.

NB. in the example we assume this file and the referenced ones are in the same folder.

XPOINTER

```xml
<?xml version='1.0'?>
<!DOCTYPE price-list SYSTEM "price-list.dtd">
<price-list xml:lang="en-us">
  <item id="w001">
    <description id="w001-description">
      <p>Normal Widget</p>
    </description>
    <prices id="w001-prices">
      <price currency="USD" volume="1+">39.95</price>
      <price currency="USD" volume="10+">34.95</price>
      <price currency="USD" volume="100+">29.95</price>
    </prices>
  </item>
  <item id="w002">
    <description id="w002-description">
      <p>Super-sized widget with bells <i>and</i> whistles.</p>
    </description>
    <prices id="w002-prices">
      <price currency="USD" volume="1+">59.95</price>
      <price currency="USD" volume="10+">54.95</price>
      <price currency="USD" volume="100+">49.95</price>
    </prices>
  </item>
</price-list>
```

```xml
<?xml version='1.0'?>
<price-quote xmlns:xi="http://www.w3.org/2001/XInclude">
  <prepared-for>Joe Smith</prepared-for>
  <xi:include href="price-list.xml" xpointer="w002-description"/>
  <xi:include href="price-list.xml" xpointer="element(w002-prices/2)"/>
</price-quote>
```

The attribute @xpointer may reference fragments of the XML document, e.g. a node. To do so, it uses the XPointer mechanism, which specifies rules (syntax and functions) to identify XML nodes (elements), e.g. via their identifier or via their position in the hierarchy.

XPointer was born as an **extension** of another language, i.e. XML Path language, or XPath, that we will discuss more in depth. It includes a syntax and a number of functions to identify XML nodes (e.g. elements, text, groupings).

XPointer is more powerful than XPath, although it is less flexible and can be used in less contexts. In particular, XPointer can be used with XInclude and XLink, while XPath can be used with any of the previous one, XQuery and XSLT.

XLINK

```
<bookstore xmlns:xlink="http://www.w3.org/1999/xlink">

  <book title="Harry Potter">
    <description
    xlink:type="simple"
    xlink:href="/images/HPotter.gif"
    xlink:show="new">
    As his fifth year at Hogwarts School of Witchcraft and
    Wizardry approaches, 15-year-old Harry Potter is.....
    </description>
  </book>
</bookstore>
```

XLink is a mechanism used to create hyperlinks between XML documents. However, such linksvdo not work in a browser like hypertext links in HTML.vThe XLink namespace is http://www.w3.org/1999/xlink and has three main attributes: @type, @href, and @show.

@xlink:type="simple" creates a "HTML-like" link (click on a word and get redirected to another page or multimedia)@xlink:href attribute specifies the URL tolink to (e.g. an image or another XML file) @xlink:show="new" specifies whether the link should open in a new window.

XPATH

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book>
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```

XPATH is another standard part of the XML-family.

Like XPointer, XPath provides a notation and a series of functions that can be used to navigate through elements, text, and attributes in an XML document. It is a fundamental building block of XQuery and XSLT. Currently, XPath 3.0 is available.

Similarly to when you browse your file system, XPath provides a syntax to express a path to reach a resource. In this case the resource is not a file on your laptop but a node.

In XPath there are 7 types of nodes: **element**, **attribute**, **text**, namespace, processing-instruction, comment, and **root** nodes.

XPath allows you to traverse axes:

Parent: title > book

Children: book > title, author, year,

price

Siblings: title > author, year, price

Ancestors: title> book, bookstore

Descendants: bookstore > book,

title, author …

You always start the journey from the root element, if not specified differently. Some examples of XPATH syntax:

-Paths

bookstore Returns the node that matches the name

bookstore/book/author Select the last child node in the above string. The path starts from the current element (bookstore*). Notice* ***the path returns an element node***. To return its text use: bookstore/book/author/**text()**

bookstore//author Select the last node in the XPath string. Select a descending element regardless of its level in the hierarchy (i.e. author may not be a direct child)

//title/@lang Returns the value of the attribute of the selected element

//@lang Returns all the values of all the attributes @lang (even when several elements have the @lang attribute)

. The current node (after the last traversing path)

.. The parent node of the current one

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book>
    <title lang="en">Harry Potter</title>
  </book>
  <book>
    <title lang="it">Harry Potter</title>
  </book>
  <book>
    <title lang="en">James Bond</title>
  </book>
  <book>
    <title lang="en">Alice in Wonderland</title>
  </book>
</bookstore>
```

-Predicates

/bookstore/book All the children elements named book. The path starts from the root element. Notice that XPath can return zero, one or more nodes (in a list).

/bookstore/book[1] Returns only the n-th (first in this case) of the children elements called book

The horror!! Notice the index in XPath starts from 1, not 0.

/bookstore/book[last()] Only the last of the children elements called book

/bookstore/book[last()-1] Only the last but one of the children elements called book

//title[@lang] Return all the elements title (at any point of the hierarchy) that have an attribute @lang.

Notice that it does not return the value of @lang, but of title.

//title[@lang='it'] Return the elements title (at any point of the hierarchy) that have an attribute @lang with value equal to 'it'

-Wildcards

***** Any element node

***@** Any attribute node

node() Any kind of node (elements, attributes, etc.)

//* Any element in the document

/bookstore/* Any element child of bookstore

-Multipaths

//book/title | //book/title/@lang Select all elements title AND all the values of the attribute @lang of title elements

-Axes

Knowing which one to use is important when **traversing** a document. Several iterations over the XML hierarchy may happen, therefore the current node changes at every iteration.

/x, the first node specified is the root element (called absolute path).

//x, the node specified is a descendant of the current node

x, the element is the direct child of the current one.

For instance, the first iteration over a document would always require us to access the root element at first /bookstore/book At this point the, say, "cursor" has moved lower in the hierarchy.

To keep traversing the tree, and e.g. access the value of @lang, we can either specify and absolute path (which starts from the root) /bookstore/book/title/@lang Or we can specify relative path, starting from the current element (book) title/@lang

XPATH provides also some magic words to traverse XML axes.

child::title/text() Returns the text of title, if direct child of the context node. If the context node is

/bookstore, the path does not return anything. If the context is //book, it returns a list of strings.

Context: /bookstore

descendant::title Returns all the title nodes that are children or grandchildren of the context node (bookstore).

child::* Select all direct children (*) of the context node (i.e. all book elements)

Context: title

parent::title Returns all the book elements, if direct parent of the context node (bookstore).

Context: book[3]

preceding-sibling::book Select all previous book elements at the same level of the hierarchy if their name is book

Prof provided some link to practice with xpath (slide 8.42)

XQUERY

XQuery is the **official query language of XML**. It uses XPath to select and retrieve XML nodes in one or more documents. It is supported by most NOSQL databases, meaning, databases designed to store XML documents. It can be used to extract and retrieve information from a collection of XML documents, or to transform XML documents into other XML-like documents (e.g. into HTML documents). NB. In this tutorial we see only examples of how to use XQuery to query XML documents, while we will use XSLT to transform XML documents into HTML documents. **For the sake of your project, you are free to choose between XQuery, XSLT, or python only, to transform XML into HTML.**

[always referring to the last picture]

doc("books.xml")/bookstore/book/title First, the function doc() specifies the name of the file to open. The following XPath specifies which elements to return.

FOR LET WHILE ORDER BY RETURN **FLWOR** (read "flower") is an acronym that specifies more sophisticated constructs that can be used in XQuery. Similar constructs can be found in Python (and any other programming language). Like in Python you do not need to use all the constructs at the same time, but only when applicable.

FOR iterate over a sequence of nodes, e.g. **for $x in doc("books.xml")/bookstore/book**

LET binds a sequence to a variable, e.g. **let $enLang := 'en'**

WHERE filters nodes, e.g. **where $x/title[@lang = $enLang]**

ORDER BY sorts results, e.g. **order by $x/title**

RETURN defines results to be returned, e.g. **return $x/title**

```
1 <title lang="en">Alice in Wonderland</title>
2 <title lang="en">Harry Potter</title>
3 <title lang="en">James Bond</title>
```

for $x in doc("books.xml")/bookstore/book

let $enLang := 'en'

where $x/title[@lang = $enLang]

order by $x/title

return $x/title

If I want to change the structure of the output XML I can specify new elements in the return clause:

for $x in doc("books.xml")/bookstore/book

let $enLang := 'en'

where $x/title[@lang = $enLang]

order by $x/title

**return {data($x/title)}
**

Here you can try your queries: **https://www.videlibri.de/cgi-bin/xidelcgi**

HTML

HyperText Markup Language is the w3c standard markup language of the web. Unlike XML, it is used to encode texts to be presented on a web page. Like XML, it has a prolog, some mandatory elements <html>, <head>, and <body>, and plenty of optional ones. The syntax (i.e. the grammatical rules) is very similar to that of XML. However HTML (1) uses its own vocabulary of elements and attributes, while XML can be used with any schema/DTD, and (2) has its own rules (e.g. a title should not include a paragraph).

HTML requires 3 mandatory elements:

- **html** the root element

- **head** includes instructions (e.g. meta, links to css files). The element title is shown in the browser tab, all other children are not visible

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello world</title>
  </head>
  <body>
    <header>
      <nav>
        <ul>
          <li><a href="about.html">About me</a></li>
          <li><a href="contact.html">Contact me</a></li>
        </ul>
      </nav>
      <h1>Hello world!</h1>
    </header>
    <main>
        <p>This is my first <strong>website</strong>.</p>
    </main>
    <footer>
      <p>Check my profile on
        the <a href="https://www.unibo.it/sitoweb/marilena.daquino2">Unibo website</a></p>
    </footer>
  </body>
</html>
```

- **body** includes texts and links to multimedia to be visualized on the web page

The element **head** includes among others:

- meta (recommended) to specify the encoding of characters (if you forget it special characters will not be correctly rendered)

- title (recommended) to create a title placeholder on the browser tab

- link (optional) to explain the browser where to find CSS instructions (if any)

The element **body** includes among others:

- header (optional) includes the heading of the page, such as the main title h1 and the menu nav (both recommended)

- main (optional) includes the main content of the page (which is likely to change from one page to another)

- footer (optional) includes contacts and links

The **nav** element includes an unordered list (ul) of list items (li). If no style is applied to the HTML document, this is visualized as a bullet list.

 elements, when included in <nav>, usually include the element anchor (**a**) which is the element used to create hypertext links.

The value of **<a>** can be a text node (e.g. “about me”) or a multimedia object (e.g. an image <img>, a video <video>, an audio <audio>). When it includes a text, the default behavior of the browser is to visualise it blue and underlined (or violet if already clicked).

The **value** of @href is the URL to another web document. It can be an html page that is part of the same website (in this case the URL is relative, since the linked file is usually stored in the same folder of the linking file) or an external one (in this case the URL is absolute and includes also the http(s) protocol and the domain of the external website).

To include images, the tag <**img**> is used. The attribute @src (source), similarly to @href, provides the relative/absolute URL of the image file to be displayed. The @alt attribute includes a text description of

the image, in case it could not be visualized properly. The element cannot include text, therefore it is often written in the short form.

```
<p id="p1" class="big blue">first</p>
<p id="p2" class="blue">second</p>
<p id="p3" class="blue">third</p>
<p id="p4" class="blue">fourth</p>
<p id="p5" class="small blue">fifth</p>
```

Every element in an html page can have the attributes @**id** and @**class**. Similarly to @xml:id, @id specifies one user-defined value that is unique in the entire document (i.e. two elements should not have the same value for @id) @class, instead, allows multiple values (separated by white space) and multiple elements can share the same values.

```
.big {
    font-size: 10@px;
}

#p3 {
    text-decoration: underline;
}
```

@id and @class are mainly used in CSS to identify (groups of) HTML elements, when attributing styles.

Although there is no semantics associated to the values of such attributes, a good practice is to name them transparently, according to some internal convention, so as to facilitate their readability by external users.

Indentation and new lines in HTML files are not parsed by the browser, that is, you cannot use "return" to create empty lines. Likewise, multiple white spaces are ignored. Instead block elements must be used to see the effect on the final web page. <nav>, <h1>, <section>, <p>, etc are block elements (a new line is created after it is closed) <a> and are inline elements (i.e. the text after this element is in the same line).

NB: If you call your file index.html, the browser will interpret that as the homepage of the website (it is indeed mandatory to have a index.html file in every online website). If you upload your html and css files on a github repository and you activate github pages, you'll be able to publish your website for free.

XSLT: HOW TO TRANSFORM AN XML/TEI FILE TO AN HTML FILE

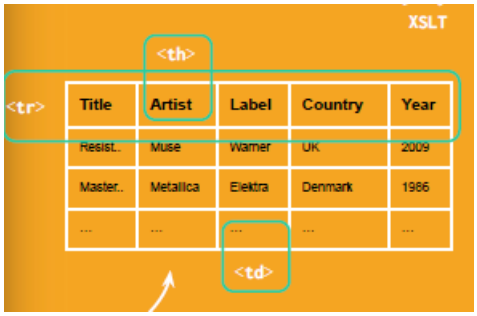
Notice that while you should have only one XML file (or collection of files) to represent the content of a digital edition, you may have as many HTML of the same content as you like, each showing different presentational aspects (e.g. a version for your personal website, another for the publisher). To this extent, there are no expectations on the structure and appearance of the final HTML, which only depend on the developer that performs the transformation. In other terms there is not just one way to transform XML into HTML.

eXtensible Stylesheet Language Transformations (XSLT) is the declarative language to convert XML documents into other XML documents, HTML and other formats (also PDF!) A stylesheet is a file that includes the rules to identify XML nodes in a source document (using XPath), specifies how to manipulate them (e.g. find all <persName> elements in a text and add them to a list), and store them in a new file. To function XSL needs a processor, which is included in browsers (e.g. Gecko in Mozilla Firefox), provided by programming languages libraries (e.g. lxml python library) or embedded in editors (e.g. Saxon in Oxygen editor).

Example:

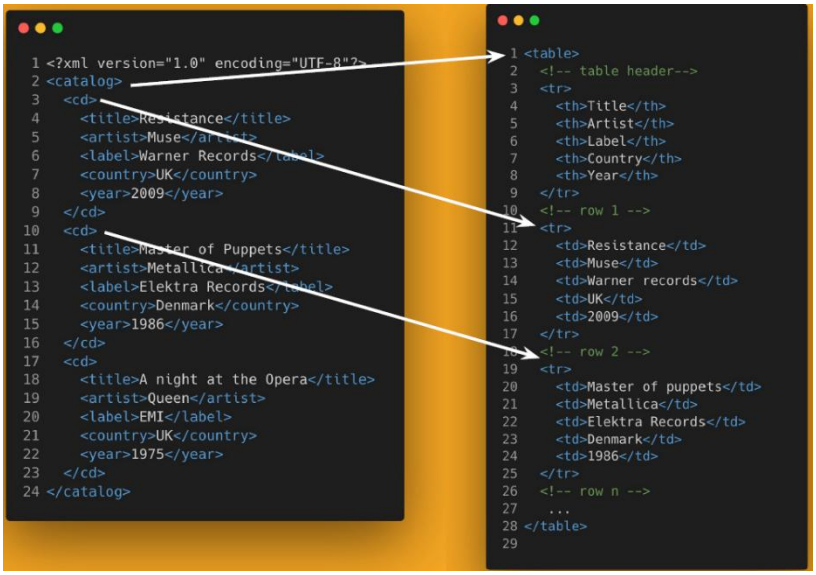
```xml
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Resistance</title>
    <artist>Muse</artist>
    <label>Warner Records</label>
    <country>UK</country>
    <year>2009</year>
  </cd>
  <cd>
    <title>Master of Puppets</title>
    <artist>Metallica</artist>
    <label>Elektra Records</label>
    <country>Denmark</country>
    <year>1986</year>
  </cd>
  <cd>
    <title>A night at the Opera</title>
    <artist>Queen</artist>
    <label>EMI</label>
    <country>UK</country>
    <year>1975</year>
  </cd>
</catalog>
```

Given a list of albums (<cd>) in a XML file, I want to create a web page which organizes them in a table. Each row of the table describes one album, and every column represents a piece of information about it (title, artist, label…).



In HTML, the element <**table**> is used to visualize a table. Every row of the table is included in the element <tr>. Content of cells are represented by <td>. <th> is used for the table headers.

We define a preliminarily mapping between XML elements and HTML elements.



catalog > table the container

N.A. > tr[1] the header

cd[1] > tr[2] the first row

cd[2] > tr[3] the second row

cd[3] > tr[4] the third row

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
      version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>My music</title>
      </head>
      <body>
        <h2>My music collection</h2>
        <table>
          <tr>
            <th>Title</th>
            <th>Artist</th>
            <th>Label</th>
            <th>Country</th>
            <th>Year</th>
          </tr>
          <xsl:for-each select="catalog/cd">
            <tr>
              <td><xsl:value-of select="title"/></td>
              <td><xsl:value-of select="artist"/></td>
              <td><xsl:value-of select="label"/></td>
              <td><xsl:value-of select="country"/></td>
              <td><xsl:value-of select="year"/></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

An XSLT file to transform the prior XML file into HTML would look like this. Notice that a XSLT file is yet another XML file. It has a prolog, a root element, and a namespace xmlns:xsl. In this example, the elements that do not use any namespace are HTML elements. The idea behind XSLT is to match XML elements with templates, i.e. every time a certain element is found in the source, a certain behaviour/transformation is triggered.

The element xsl:template always comes with the attribute @match, including an XPath that matches one or more nodes in the source XML document. "/" is an XPath that matches the whole XML document. Therefore the following instructions apply to the whole XML document. Then, create the HTML. The root <html> element is created, along with the <head> and <body> elements. In <body> we decide to create a title <h2> and we create the container element of the table <table>. In <table> we create the header, that is, the first <tr> which includes as many <th> elements as the columns we want. To dynamically populate the table with as many rows as the number of albums, we use a for loop, which in XSL is called **xsl:for-each**. The for loop prevents us to write instructions for each and every <cd> elements, and allows us to establish a single rule to be applied every time an element <cd> is found. Notice the Xpath "catalog/cd" This means the document "/" has already been matched, and the root element <catalog> is the first to be accessed by the parser, followed by its children <cd>. From this moment onwards, <cd> is the context node. Whenever it matches the element <cd> - N.B. if this is a direct child of the element <catalog> - create an element <tr>. Also, create 5 <td> children elements, which correspond to the number of cells in the row.

In each <td> element we include the value of a XML element, e.g. <title>, <artist>, etc. that are children of <cd>, using the construct xsl:value-of. Notice that the XPath. Again, it does not start with an absolute path "/", but with a relative one, since the context node is the one matched in the for loop.

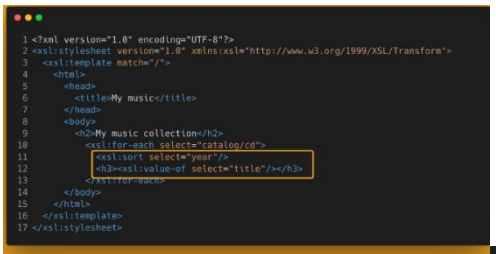
So in the end, XSLT allows you to declare what should appear in the final HTML. Sometimes the declaration does not depend on any matched pattern (see blocks 1 and 3), while in other cases a pattern (block 2) needs to be matched in order to trigger a template rule.

Like in Python, XSLT allows you to filter out elements, e.g. in a for loop, when a certain condition is met. The construct **<xsl:if>** is used to specify the condition in the attribute @test.

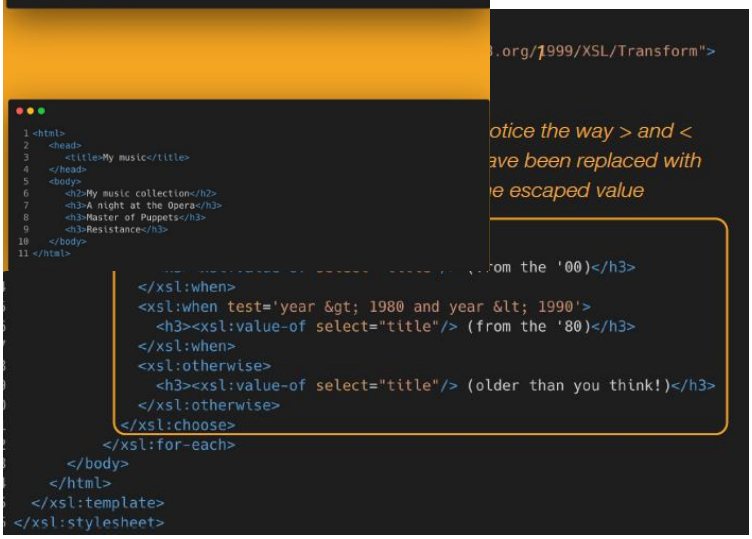
```
country/text() # the text node exists
string-length(country/text()) > 0 # the length is greater than zero
country/text() = "UK" # the text exists and is equal to "UK"

year < 2000 # year is less than 2000
year != 2009 # year is anything but 2009
year = 2009 or year = 1986 # year is either 2009 or 1986
year > 2000 and year < 2010 # year is greater than 2000 and less than 2009
```

The value of @test is either a XPath if you want to check the existence of a node, or an expression including an operator. Common operators include + , - , * , = , != , > , < , or , and . Also functions can be used, e.g. string-length.



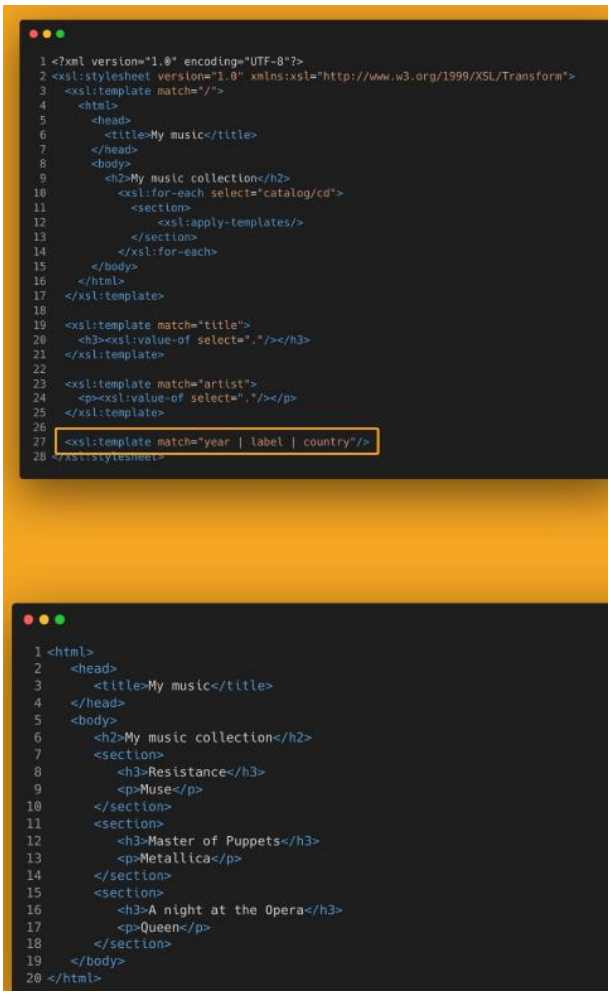
Notably, only a construct for describing the if condition exists, while there is no construct for describing the else condition. To manage several scenarios, <xsl:choose> is used. It includes one or more <xsl:when> conditions and one <xsl:otherwise> to define a default behaviour in case none of the prior conditions is met.



You can **sort** elements in a sequence (e.g. in a for loop) according to one or more parameter, thanks to <xsl:sort>.

Notice the element **xsl:sort** affects results of the parent instruction.

Templating rules can be externalized. Instead of declaring the HTML output and inject the rule we declare rules somewhere else and call them in the point of the HTML we need it with **<xsl:apply-templates>**. The order is important! In python this mechanism is similar to function declaration. Three templates are called by the above one. Notice the XPaths in @match: these imply the context node is "catalog/cd", otherwise they won't match any node.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>My music</title>
      </head>
      <body>
        <h2>My music collection</h2>
          <xsl:for-each select="catalog/cd">
            <section>
                <xsl:apply-templates/>
            </section>
          </xsl:for-each>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="title">
    <h3><xsl:value-of select="."/></h3>
  </xsl:template>

  <xsl:template match="artist">
    <p><xsl:value-of select="."/></p>
  </xsl:template>

  <xsl:template match="year | label | country"/>
</xsl:stylesheet>
```

```xml
<html>
   <head>
      <title>My music</title>
   </head>
   <body>
      <h2>My music collection</h2>
      <section>
         <h3>Resistance</h3>
         <p>Muse</p>
      </section>
      <section>
         <h3>Master of Puppets</h3>
         <p>Metallica</p>
      </section>
      <section>
         <h3>A night at the Opera</h3>
         <p>Queen</p>
      </section>
   </body>
</html>
```

This is the result of the prior transformation. Notice that elements can be ignored by applying an empty template. If not specified, the text content of those nodes is included in the final output.

To transform XML documents into HTML using XSLT there are a few technical solutions:

- **Stand-alone** (or server-side): a software + a XSL processor to produce a static HTML file.

- **Client-side**: the browser (Javascript) dynamically produces a HTML file on demand

Stand-alone transformations:

- online tools like Free Formatter

- editors with an integrated XSLT Processor (e.g. Oxygen, which

requires an Academic license) or with XSLT plugins (VS Code +

XSLT/XPath extension + Saxon processor)

- Command-line programs

- Programming languages

[slides 31 -34 on how to use command line]

Server-side transformations:

Python, among other languages, offers APIs (i.e. libraries) to manipulate XML documents, e.g. lxml. Lxml allows you to parse and query XML documents, use XPath to identify and retrieve nodes, as well as perform XSL transformations. [slide 36]

Client-side transformations

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>
<catalog>
   ...
</catalog>
```

Some browsers (e.g. Safari) allow you to perform a XSL transformation on the fly. You must specify a XSL file in the XML document and open the XML document in the browser. In some browsers (e.g. Chrome, Firefox) you will see an error is returned, due to CORS cross origin access restrictions (see below the inspector web of Chrome). This is because the browser interprets the XSL file as in a different server (what..?!) than the one hosting the XML file, then it prevents you to do it for security reasons.

To overcome CORS barriers, we use Javascript to instruct the browser on how to apply a XSL stylesheet on a XML file. To actually overcome the restrictions, we publish our files on an online (or local) web server (in our case github pages), so that the origin of the two files (XML and XSLT) is the same.

Javascript?

Javascript is a programming language widely used for web development along with HTML and CSS. A Javascript script is usually included in a .js file and referenced in a html file with the tag <script> in order to be imported. Alternatively, js code can be injected in HTML files.

We create a skeleton HTML file (e.g. index.html) where we include a JS script that renders the XML as HTML snippet on demand.

See a demo: https://marilenadaquino.github.io/tesr_dhdk/exercises/catalogue_js.html

Source code: https://github.com/marilenadaquino/tesr_dhdk/blob/main/exercises/catalogue_js.html

PS. You can use this code in your final project to show the XML to HTML transformation of your XML/TEI file.

Several works exist trying to generalize the transformation of XML/TEI documents into HTML, mainly using XSLT developed by community members, attempting to foresee any potential combination of TEI elements in a given XML document.

- Tei2html, a collection of XSL files developed for the project Gutenberg

- TEI community XSLT stylesheets

- TEIGarage conversion tool

- TEI Boilerplate

NB. For the sake of the exam, you can either use of these tools, or you can create your own XSLT stylesheet.

In case you reuse something existing, you will have to document the process you used on the project website.

SEMANTIC WEB

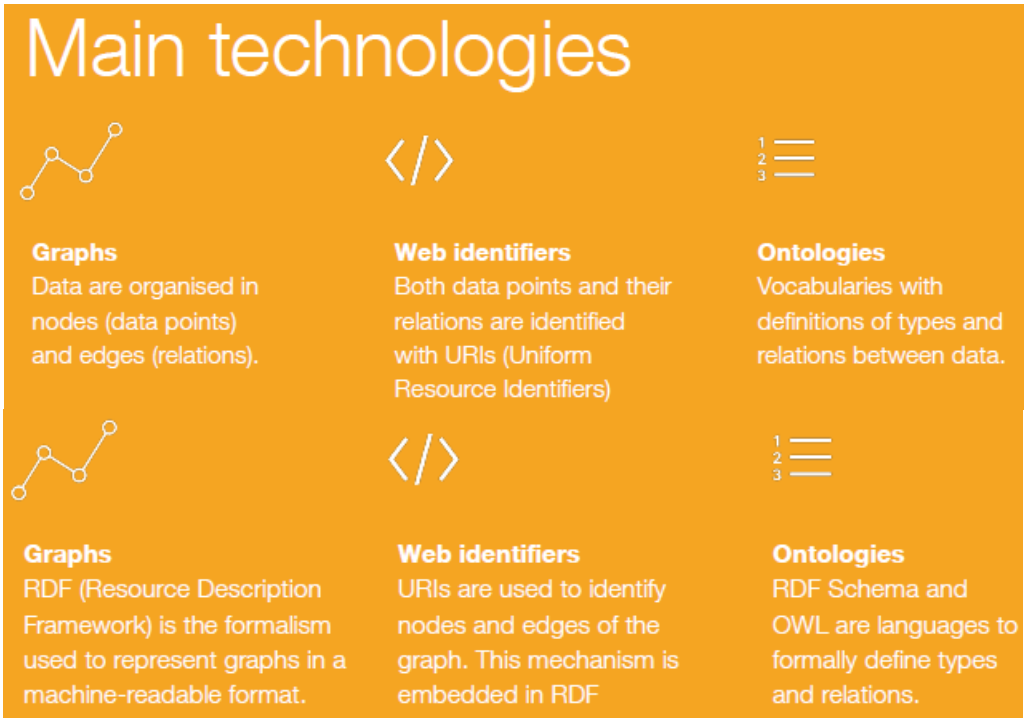
The current web is based on interlinked **HTML** (HyperText Markup Language) documents, served on the web via the **HTTP** (HyperText Transfer Protocol) protocol and retrievable via their **URL** (Uniform Resource Locator). Such documents include a wealth of **unstructured** information that the **human reader** is usually required to parse manually to understand its content and the links to other documents. While **structured** databases exist, these are isolated from similar data and the relations between data points are not understandable by machines. The idea of Semantic Web is to make information available on the web **understandable by machines**, so as to facilitate every-day information retrieval tasks (i.e. search) and make results more accurate.

Objectives:

1. Make structured and semi-structured data available on the web according to standard **formats** (e.g. XML) to facilitate exchange and mashup

2. Make available not only data but also metadata about **relations** between data available on the web

3. Describe the **semantics** associated to data and relations with a formalism that is understandable by Machines



URIs, similarly to URLs (locations) and URNs (names), identify resources. However, **URLs** are meant to provide the address of the digital document without retaining any information about its content. **URNs** instead, are identifiers associated to the content of a resource (e.g. the ISBN of a book represents a specific edition of that content), but do not tell us where to find the resource. **URIs** both identify conceptual or real-world entities (e.g. people, places) and tell us where to find data about that concept. Notably the data associated to the entity is provided according to the RDF model. Notice that the URI identifying a concept **does not identify a HTML page** describing the entity.

- If the location of a HTML file describing the entity changes, the URI does not change.

- Many HTML pages can describe the same entity identified by the same URI.

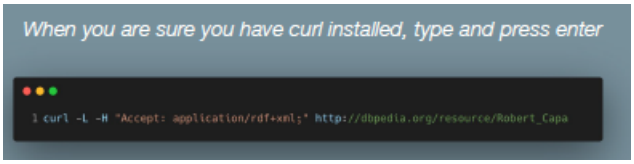
- A HTML page can include information about many entities, hence being linked to many

URIs.

CONTENT NEGOTIATION: However, when associating a URI to a concept, a mechanism called **content negotiation** is in place. That is, according to who requests the URI, data in a different format is returned, such as a web page, a XML dump of data, or other formats. E.g. https://dbpedia.org/resource/Robert_Capa returns a web page (built on top of the graph data) if requested by a browser (i.e. a user) Notice it redirects to https://dbpedia.org/page/Robert_Capa The same URI returns the RDF data if requested by a machine. For instance, you can request the XML version of data via command-line, which redirects and downloads data available at: https://dbpedia.org/data/Robert_Capa.xml

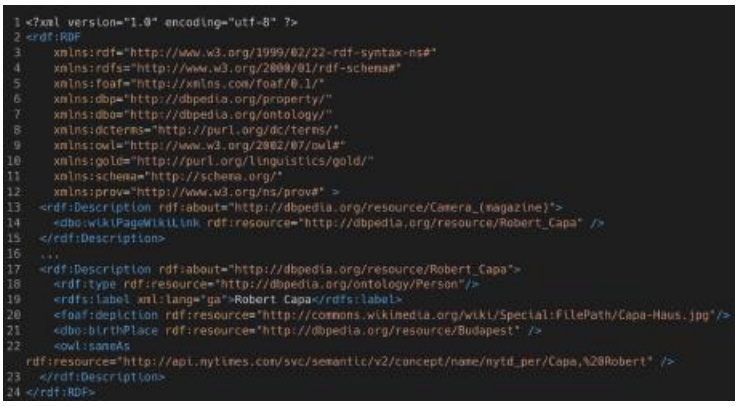
HOW CAN YOU REQUEST DATA VIA URI?

Open the terminal and check if you have the command curl. Type:



- (Windows) curl -V

Or install: https://curl.se/download.html Or download RDF/XML (also on our repo)



It's an XML file

- Prologue

- Root element

- Namespaces

- 2 <rdf:Description>

- @rdf:about

- @rdf:resource

- @xml:lang and element value

- <owl:sameAs>

Let's draw a network graph where our nodes are named with the values of @rdf:about and @rdf:resource and the edges are named with the tag name of elements children of rdf:Description.

- Download yEd editor

- Create a new document

- Select from the right column the graphic library Graffoo

- Add some individuals (pink circles): Robert Capa, Budapest, magazine…

Use the value of @rdf:about and @rdf:resource to name the circles

- Add relations (blue) edges between circles and name them as the element name where @rdf:resource appear

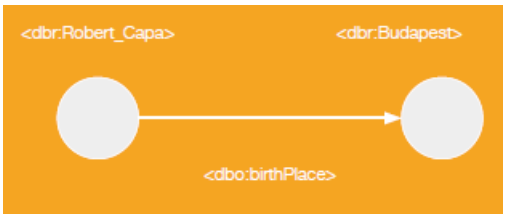
- Add a string (green block): use the value of rdfs:label element

- Add a class (yellow block): call it with the value of rdf:type/@rdf:resource

We have studied XML and we have seen that it can be interpreted as a tree, i.e. a hierarchical structure of parent/children elements, which can be seen also as a uni-directed graph. RDF allows you to create bidirected graphs (there is not just one hierarchy).

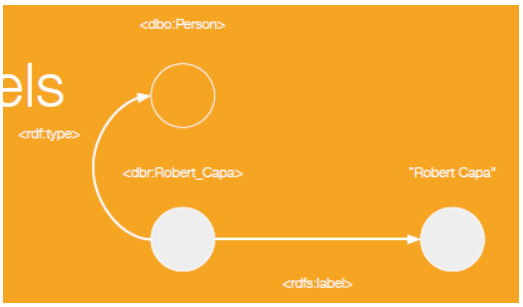
But if XML is our syntax, what is RDF?

RDF Resource Description Framework (RDF) is a model that allows you to represent knowledge in the form of graphs. The pillars of a graph are statements, called triples, made of <subject> <predicate> <object> NB. In the RDF/XML file we have seen <rdf:Description> identifies a subject (@rdf:about includes its URI), its children elements are the predicates, and @rdf:resource (or the value of the child element) are the objects.



The basic structure <subject> <predicate> <object> is composed of **2-3 URIs**, each representing a building block of the triple. While @rdf:about and @rdf:resource already mention the URI, it's worth noting that also the predicate (i.e. the relation) can be represented by a URI. In fact, the URI of predicates is made of the namespace URI + the tag name. To make the graph and the data more readable, full URIs can indeed be replaced by the ns prefix + name.

Sometimes the object is not a resource, i.e. a URI, but a string, also called **Literal**. Several data types exist to characterise the string. Such types are borrowed by XML Schema (xsd) data types, and include integers, dates, etc.



TYPES AND LABELS: A special predicate is **rdf:type**, which allows you to specify one or more types categorizing entities. Such types are called **classes**. Another special predicate is **rdfs:label**, which associates a human-readable string to URIs.

RDF SYNTAXES

```xml
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:dbp="http://dbpedia.org/property/"
    xmlns:dbo="http://dbpedia.org/ontology/"
    xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:gold="http://purl.org/linguistics/gold/"
    xmlns:schema="http://schema.org/"
    xmlns:prov="http://www.w3.org/ns/prov#" >
  <rdf:Description rdf:about="http://dbpedia.org/resource/Robert_Capa">
    <rdf:type rdf:resource="http://dbpedia.org/ontology/Person"/>
    <rdfs:label xml:lang="ga">Robert Capa</rdfs:label>
    <dbo:birthPlace rdf:resource="http://dbpedia.org/resource/Budapest" />
  </rdf:Description>
</rdf:RDF>
```

(data.xml)

As aforementioned, **rdf is not a syntax**. Instead, several syntaxes can be used to represent RDF data, as long as the triplet format is respected. After **XML**, the simplest notation is **N-Triples**, where s,p, and o URIs are written in full and triples are separated by full stop.

```
<http://dbpedia.org/resource/Robert_Capa> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/ontology/Person> .
<http://dbpedia.org/resource/Robert_Capa> <http://www.w3.org/2000/01/rdf-schema#label> "Robert Capa"@ga .
<http://dbpedia.org/resource/Robert_Capa> <http://dbpedia.org/ontology/birthPlace>
  <http://dbpedia.org/resource/Budapest> .
```

data.ntriples

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix dbo: <http://dbpedia.org/ontology/> .

dbr:Robert_Capa a dbo:Person ;
  rdfs:label "Robert Capa"@ga ;
  dbo:birthPlace dbr:Budapest .
```

A compact version of N-triples is **Turtle**, where full URIs are replaced by their namespaces prefixes and the last part of the URI.

```
[
  {"@id":"http://dbpedia.org/ontology/Person"},
  {"@id":"http://dbpedia.org/resource/Budapest"},
  {"@id":"http://dbpedia.org/resource/Robert_Capa",
   "@type":["http://dbpedia.org/ontology/Person"],
   "http://www.w3.org/2000/01/rdf-schema#label":[
     {"@value":"Robert Capa",
      "@language":"ga"}
   ],
   "http://dbpedia.org/ontology/birthPlace"[
     {"@id":"http://dbpedia.org/resource/Budapest"}
   ]
  }
]

```

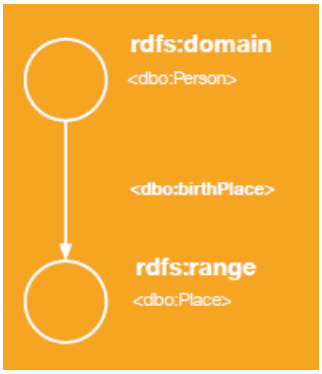
A more recent and convenient syntax is **JSON-LD**. Based on JSON, a format for exchanging data on the web that is "replacing" XML in plenty of contexts, it can be extended to represent triples.

RDF FORMATS: While RDF is not a syntax, .rdf can be used as a **format**. This format can be used regardless of the syntax used in the document, and will be appropriately interpreted by a parser. Rdf data can be converted losslessly from format to format, e.g. using converters. Similarly, RDF visualisers exist and accept any of the standard syntaxes, e.g. RDF Grapher, isSemantic.

ONTOLOGIES: Similar to XML schemas, **predicates and classes are defined in external vocabularies called ontologies**, which include terms, the hierarchy, **their usage and restrictions** (**domain and range**). **Ontologies are documents** (external to data) **available online for reuse**. Their role is to facilitate semantic interoperability, that is, the more people describe their data using the same ontologies, the easier will be to seamlessly integrate different data sources. Content-wise, an ontology describes concepts and properties that characterize a certain domain (e.g. Arts, libraries) or a certain task (e.g. provenance). As we have seen, multiple ontologies (namespaces) can be used at the same time to describe data.

OWL: The definitions and the constraints are written with a dedicated language, called **Web ontology language** (**OWL**). Notably an ontology is written using the RDF formalism (subj +pred+obj). Like RDF, OWL is a formalism to represent knowledge, and several syntaxes can be used to describe an ontology.

CLASSES: Are categories describing identified by URIs. In ontologies it is possible to formalise a hierarchy of concepts via the property **rdfs:subClassOf**. A basic principle is inheritance, or **subsumption**: an entity that belongs to a subclass also belong to the superclass.



PROPERTIES: They represent predicates, that is, **relations** between entities that belong to classes. The class of subjects is called **domain** of the property, and the class of the object is called **range**.

There are two types of properties:

- **Object properties:** connect entities identified with a URI

- **Data properties:** connect a URI to a Literal

Also properties can be part of a hierarchy, represented via the predicate **rdfs:subPropertyOf**. Notice in the example that the super property has a different namespace. Indeed, terms belonging to different ontologies can be integrated in the same ontology.

ONTOLOGY ALIGNMENT: Despite ideally ontologies should be complementary (content-wise), they may present overlap of terms. When the semantics is similar but not a full match, r**dfs:subClassOf/subPropertyOf** are used between terms. When there is a full overlap, special predicates are provided by the OWL language, namely:

- **owl:equivalentClass** (between classes)

- **owl:equivalentProperty** (between properties)

- **owl:sameAs** (between individuals). Notably this predicate is found mostly in datasets rather than

in ontologies.

[Se vuoi, vedi un tutorial su protegè]