Text Encoding and Semantic Representation

Introduction | XML

marilena.daquino2@unibo.it | https://github.com/marilenadaquino/tesr_dhdk

Outline

About

The project

Introduction to XML

Hands-on

In this course

You will learn how to…

- **Encode** a text using a markup language (XML)
- Use a **metadata schema** to annotate literary texts (TEI)
- Extract and annotate **real-world entities** from texts (NER), APIs, etc. and create structured data
- Generate web documents (HTML) from XML/TEI documents
- Transform semantic data into **graphs** (RDF) according to an **ontology** (OWL)

Example

Imagine you have a web project

The digital edition of Dante's works.

On the website you want to browse his texts and the comments made by scholars.

You also want an index of people, places, and texts he quoted / mentioned in his work.

https://dama.dantenetwork.it
https://dantesources.dantenetwork.it/en/

Example

Transcription

To create a digital edition, you will probably work on digital images of your texts (jpg, png) and you will transcribe the text in digital format (txt)


```
Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
ché la diritta via era smarrita.

Ahi quanto a dir qual era è cosa dura
esta selva selvaggia e aspra e forte
che nel pensier rinova la paura!
```

Example

Encoding (XML/TEI)

You want a **convention** (TEI) to annotate important content in your text (e.g. linguistic aspects, person names)
…and you need a **syntax** (XML) to do it in a way that is understandable by both humans and machines.

```
<lg type="canto">
    <l>
      <LM lemma="il" catg="rdms">Nel</LM>
      <LM lemma="in mezzo di" catg="eilaksl">mezzo</LM>
      <LM lemma="il" catg="rdms">del</LM>
      <LM lemma="cammino" catg="sm2ms">cammin</LM>
      <LM lemma="di" catg="epskg">di</LM>
      <LM lemma="nostro" catg="as1fs">nostra</LM>
      <LM lemma="vita" catg="sf1fs">vita</LM>
    </l>
    <l>
      <LM lemma="mi" catg="pf1sypr">mi</LM>
      <LM lemma="ritrovare"
catg="vta1ip1s">ritrovai</LM>
      <LM lemma="per" catg="epskpl">per</LM>
      <LM lemma="una" catg="rifs">una</LM>
      <LM lemma="selva" catg="sf1fs">selva</LM>
      <LM lemma="oscuro" catg="a1fs">oscura</LM>
    </l>
```

Example

Annotate

Some entities appear in other documents on the web, that include information you haven't (e.g. Wikipedia biographies). You want an **identifier** shared between you and others, and include a hook (annotation) in your text for future reuse.

```
<l>
    <LM lemma="allora" catg="b"> Allor</LM>
    <LM lemma="vedere" catg="vta2irs1">vid'</LM>
    <LM lemma="io" catg="pp1slso">io</LM>
    <LM lemma="meravigliare" catg="vi1fp">maravigliar</LM>
    <LM xml:id="https://www.wikidata.org/wiki/Q1398"
        lemma="Virgilio"
        catg="np">Virgilio</LM>
 </l>
```

Example

Encoding (HTML)

You need a text that is interpretable by a **browser**, to show it to your users. This document (HTML) is not the same as the one where you annotate content.

Here the focus is on **presentational** aspects - the content is the same, but different markups of it can exist.

```html
<p>
  Nel mezzo del cammin di nostra vita<br>
  mi ritrovai per una selva oscura,<br>
  ché la diritta via era smarrita.<br>
<br>
  Ahi quanto a dir qual era è cosa dura<br>
  esta selva selvaggia e aspra e forte<br>
  che nel pensier rinova la paura!
</p>
```

Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
ché la diritta via era smarrita.

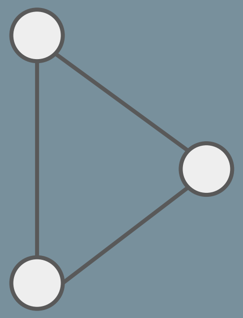
Ahi quanto a dir qual era è cosa dura
esta selva selvaggia e aspra e forte
che nel pensier rinova la paura!

Example

Semantics (RDF)

There are concepts that are not explicit in the text (e.g. relations between people and places). You separate **semantic content** from the text.

You first need to **extract** entities (e.g. persons, places, books) and relations between entities, and store them separately from the text, in RDF files (as we did for the HTML version of our text).

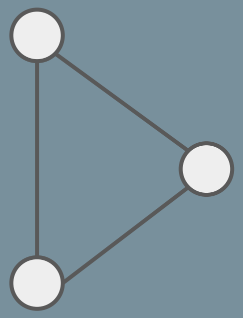


Example

Semantics (OWL)

You encode this information in a different (optimised) format than your full-text, e.g. **graph data**, and you may include extra information. You need to **organise** this knowledge in a way that is readable by the machine, and understandable by humans, i.e. an ontology (OWL).

You can then reuse this information in your web application, e.g. to create indexes, and others can access it to integrate it in their applications.



Calendar

Topics

Course Introduction + Introduction to XML
Introduction to TEI
Advanced usages of TEI
XML to HTML
Knowledge extraction (NER, API, scraping)

Text analysis
Semantic Web (RDF)
Semantic Web (OWL, SPARQL, data integration)
.* to RDF
Workshop + wrap up

Project

The second module

The exam consists in a **project presentation**.

The project is shared with the second module of this course and the exam covers requirements of both modules. Specs of the other module (in pills) are:

1. A **collection** of at least ten items related to a topic
2. An **ontology** and the documentation of the design phases (theoretical model, conceptual model, ontology)
3. A **RDF dataset** organised according to the ontology you designed
4. A **website** presenting the ontology design project

Project

Assignments of this module

One of the collected items must a text, for which you must provide

1. A XML/TEI document (a sample if too long) (included in the website)
2. A XML to HTML transformation (python) and a HTML document (included in the website)
3. A XML/TEI to RDF transformation (python) and a RDF dataset (included in the website)

Metadata of all items must be transformed to RDF (according to your ontology) **with python**

1. A .* to RDF transformation (python) and a RDF dataset for each item (included in the website)

Exam

Preparation checklist

Create a website with the following pages/sections:

- Ontology design documentation (see specs of the second module)
- Presentation of the items (see specs of the second module)
- **The encoded text** (include HTML, XML/TEI, and **script for transformation** XML to HTML, [optional] text analysis results)
- RDF dataset with metadata of all items (see specs of the second module) and **script for transformation** .* to RDF

Disclaimer

Overlaps with other courses

This course includes concepts and technologies addressed in other 1st year subjects.
It was originally meant to be held at the end of the first year, after you had most of the classes, so as to summarise the knowledge acquired into a unifying overview.
However…

Python programming

Basic concepts of python programming:

1. For Loop, if-else statements, Variable assignment, function declaration
2. Data structures: lists, tuples, dictionaries
3. File formats: csv, json, rdf (and its syntaxes)
4. Data access: read and write files

We will top them up with:

1. Data manipulation: different data structures (text, XML, RDF, json, csv)
2. Text analysis python libraries

Computational thinking and programming

Data Science

Text retrieval, analysis, and mining

Digital textuality

You should be learning:

1. HTML to format text and show content to users on the web
2. CSS and Javascript to style and interact with web content
3. Problems related to textuality in digital formats

But there is not only one way to encode documents

1. XML/TEI to annotate and exchange meaningful content between applications (not browsers)
2. XSLT: transform XML to HTML

Knowledge graphs

Basic concepts of Semantic Web technologies

1. Languages and frameworks: RDF and OWL
2. Ontology design and development
3. Knowledge extraction from structured data
4. Standards (content, metadata, ontologies) and good practices

We will top them up with:

1. Data transformation: from unstructured and semi-structured data to graphs with Python

Knowledge Organisation in libraries and archives

Knowledge Representation and extraction

XML

In pills

```
<email>
  <sender>me@mail.org</sender>
  <receiver>you@mail.org</receiver>
  <message>
    <title>Hello there!</title>
    <content>Enjoying your first class of TESR?</content>
  </message>
</email>
```

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.

It is not a programming language! It does not tell a machine “what to do” but only “how to categorise” strings according to human-defined labels (the machine does not understand the meaning of labels though)

XML

Meta-markup

It is 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 (what aspects, concepts, relations).

You must top up XML with a **schema**.
There are many schemas that one can use.

```xml
<email>
  <sender>me@mail.org</sender>
  <receiver>you@mail.org</receiver>
  <message>
    <title>Hello there!</title>
    <content>Enjoying your first class of TESR?</content>
  </message>
</email>
```

Schemas address elements, attributes, hierarchy, order, and allowed repetitions of elements.

```html
<body>
  <p>me@mail.org</p>
  <p>you@mail.org</p>
  <section>
    <h1>Hello there!</h1>
    <p>Enjoying your first class of TESR?</p>
  </section>
</body>
```

HTML shares the same syntax used by XML, but has its own vocabulary

XML

Elements

XML building blocks are **elements**: each element is composed of two **tags**, i.e. strings enclosed in <> brackets.

```xml
<email>
  <sender>me@mail.org</sender>
  <receiver>you@mail.org</receiver>
  <message>
    <title>Hello there!</title>
    <content>Enjoying your first class of TESR?</content>
  </message>
</email>
```

The **closing tag** includes a slash / before the element name.

XML

Elements

An element can include text (which is called **value** of the element)...

```
<email>
  <sender>me@mail.org</sender>
  <receiver>you@mail.org</receiver>
  <message>
    <title>Hello there!</title>
    <content>Enjoying your first class of TESR?</content>
  </message>
</email>
```

XML

Elements

… or it can include other **elements** (email includes sender, receiver, and message; message includes title and content)

```xml
<email>
  <sender>me@mail.org</sender>
  <receiver>you@mail.org</receiver>
  <message>
    <title>Hello there!</title>
    <content>Enjoying your first class of TESR?</content>
  </message>
</email>
```

we will see that an element can include **both** text and elements

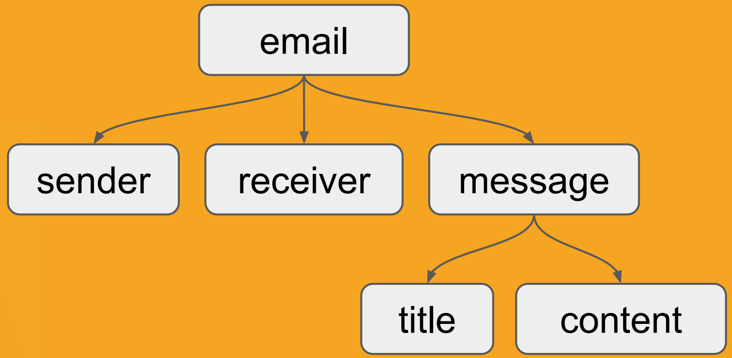
XML

Hierarchy

An XML document can be seen as a **tree**, i.e. a hierarchical structure of elements, also called **nodes**.

```
<email>
  <sender>me@mail.org</sender>
  <receiver>you@mail.org</receiver>
  <message>
    <title>Hello there!</title>
    <content>Enjoying your first class of TESR?</content>
  </message>
</email>
```

There is always a **root node** (email, in this case) that includes all elements and texts



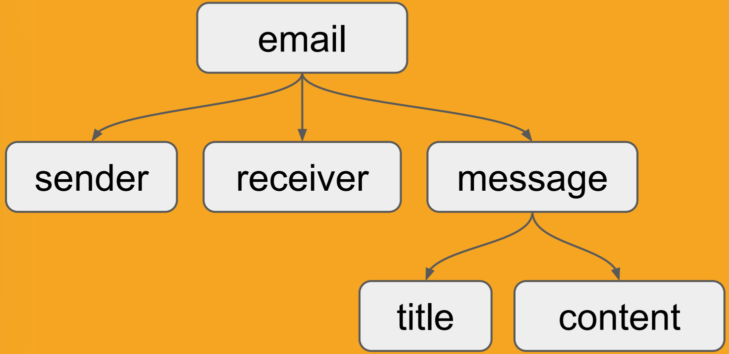
Children nodes can include more nodes or text (which is not part of the hierarchy), e.g. message has two children, and can have **siblings**, e.g. message, sender, and receiver are siblings

XML

Indentation

Notice that in a XML document, the hierarchy is graphically represented with **indented blocks**. It is not mandatory, but it helps readability.

```xml
<email>
  <sender>me@mail.org</sender>
  <receiver>you@mail.org</receiver>
  <message>
    <title>Hello there!</title>
    <content>Enjoying your first class of TESR?</content>
  </message>
</email>
```



XML

Nesting

Moreover, the hierarchy is respected by the **nesting** of elements. **The root element must include all children elements.**

```xml
<email>
  <sender></sender>
  <receiver></receiver>
  <message></message>
</email>
```

correct

```xml
<email>
  <sender></sender>
</email>
  <receiver></receiver>
  <message></message>
```

wrong

XML

Nesting

Each child element must be completely included in the parent element, to avoid **overlap**. Same applies for sibling elements.

```xml
<email>
  <sender></sender>
  <receiver></receiver>
  <message>
    <title></title>
    <content></content>
  </message>
</email>
```

correct

```xml
<email>
  <sender></sender>
  <receiver></receiver>
  <message>
    <title><content></title></content>
  </message>
</email>
```

wrong

XML

Milestones and comments

Milestones are empty elements, that act as placeholders.

Empty elements can be written in an abbreviated form:

- The opening tag ends with /
- There is no closing tag

They can be very useful to avoid overlap

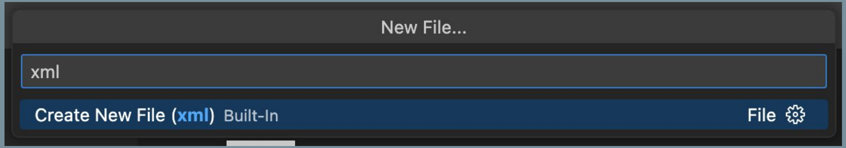
```
<email>
  <sender>me</sender>
  <receiver>you</receiver>
  <message>
    <title>hello world</title>
    <content>Do you like XML?</content>
    <sign/>
    <!-- A comment for my future self-->
  </message>
</email>
```

Comments are strings that are not processed by parsers, and includes notes for humans.

Exercise

Create your first XML file

- Open Visual Studio Code
- Menu: File > New File
- Type xml and press enter



Create an XML file to describe the list of subjects of the first year at DHDK (at least three). For each course include the title, the name of the professor, and the number of CFU.
You shall invent element names.

XML

Attributes

Attributes can be used to replace elements and record information in a more **compact** way.

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).

```xml
<email sender="me" receiver="you">
  <message signature="yes">
    <title>hello world</title>
    <content>
        <p id="1" >Do you like XML?</p>
        <p id="2">Why wouldn't you..</p>
    </content>
  </message>
</email>
```

XML

Attributes

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).

```xml
<email sender="me" receiver="you">
  <message signature="yes">
    <title>hello world</title>
    <content>
        <p id="1" >Do you like XML?</p>
        <p id="2">Why wouldn't you..</p>
    </content>
  </message>
</email>
```

XML

Attributes

In other cases **multiple values** can appear in the same attribute. In such cases white spaces separate different values.

No punctuation signs should be used to separate values in an attribute value (e.g. “, ; .”).

```
<email sender="me" receiver="you yourColleague yourTeacher">
  ...
</email>
```

XML

Namespaces

So far we have seen invented element and attribute names.

To facilitate **exchange** of XML files between people and machines, we should rather use terms from existing **schemas**, that is, documents where element and attribute names are defined and rules of usage are formally described.

We include the special attribute **@xmlns** in the root element. The value is the **URL** where the schema/vocabulary is available.

```xml
<email xmlns="http://example.org/myschema">
  <sender></sender>
  <receiver></receiver>
  <message></message>
</email>
```

XML

Namespaces

We can use as many schemas as we like.

We must specify the **namespace** (the URL) and a **prefix** for each shemas (we are allowed to have a default namespace without any prefix).

Elements and attributes belonging to a certain schema always appear with their prefix.

```xml
<email xmlns="http://example.org/myschema"
       xmlns:two="http://example.org/anotherschema">
  <sender two:name="me"></sender>
  <receiver two:name="you"></receiver>
  <message>
    <two:title>Hello world</two:title>
    <body>Getting challenging?</body>
  </message>
</email>
```

XML

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 **well-formed**.

```
<xs:element name="email">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="sender" type="xs:string"/>
      <xs:element name="receiver" type="xs:string"/>
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

XML

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 **well-formed**.

```
<!DOCTYPE email
[
<!ELEMENT email (sender,receiver,message)>
<!ELEMENT sender (#PCDATA)>
<!ELEMENT receiver (#PCDATA)>
...
]>
```

```
<!DOCTYPE email SYSTEM "Email.dtd">
<email>
  <sender></sender>
  <receiver></receiver>
  <message></message>
</email>
```

XML

Prolog

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rootElement>
  <child>Text and <innerchild>inner children</innerchild></child>
  <child></child>
</rootElement>
```

Every XML document starts with a **prolog**, i.e. a set of instructions that will be used by XML parsers.

There is only an opening tag, enclosed in <??>, which includes:

- The version of XML
- The method for encoding characters (Unicode Transformation Format, 8 bit)

A XML file w/ correct syntax is called **valid.** You can use a XML Validator.

XML

Case sensitivity

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

```
<email></EMAIL> <!-- wrong -->
```

Exercise

Revise your XML file

- Add the prologue
- Add at least two namespaces
- Replace some elements with attributes

Save your file in a folder dedicated to the course.

Work at home

Practice markup

Given the following tree, create a XML document.
This time, do not add the prologue!

Fill elements with some text when appropriate.

Tip: open the xml file of both the exercise and the homework in a browser (e.g. Chrome)