XML

DTD

- Faire un arbre et checker que tout vas bien. **Ordre important**
- La balise après le DOCTYPE doit indiquer la balise à la racine du document.
- ~~(#PCDATA, balise1, balise2)~~ FAUX => (#PCDATA | balise1 | balise2) OK
- + = [1-n]
- * = [0-n]
- ? = [0-1]
- On peut utiliser `EMPTY` pour un élément vide, mais en général avec des attributs

```
<!ELEMENT menu EMPTY>
<!ATTLIST menu
    nom CDATA #REQUIRED
    prix CDATA #REQUIRED
>
```

Ca ressemble à ça une DTD pour un elm restaurant

```
<!ELEMENT base (restaurant|ville)*>
<!ELEMENT restaurant (fermeture?, menu, menu+)>
<!ATTLIST restaurant
    nom CDATA #REQUIRED
    etoile (0|1|2|3) #IMPLIED
    ville IDREF #REQUIRED
>
```

Attribut

```
<!ATTLIST Balise
    nom type mode
    nom type mode
>
```

Liste des types :

- **CDATA** : la valeur de l 'attribut est une chaîne de caractères
- **ID** : identificateur d 'élément, **IDREF(S)** : renvoi vers un (des) ID. Ici c'est ez, **tous les ID sont uniques** donc on idref juste vers un ID existant.
- NMTOKEN(S) : un ou des noms symboliques (sans blanc)
- `(value1 | value2 | value3)` liste
- ENTITY(IES) : entités externes non XML Liste des mode :
- Une valeurs par default
- `#FIXED` une constante
- `#REQUIRED` Attribut requis
- `#IMPLIED` Attribut facultatif
-

XSchema

Généralité

- Ordre des séquences importants
- Cardinalité précise
- On a des type simple et des type complexes, qu'on imbrique dans des balise `<xs:element>`

Balise element

```
<xs:element name="restaurant" minOccurs="0" maxOccurs="unbounded">
    <xs:complexeType>..... </xs:complexType>
</xs:element>
```

Type simple

- `xs:string`
- `xs:decimal`
- `xs:integer`
- `xs:boolean`
- `xs:date`
- `xs:time`

```
<xs:element name="start_date" type="xs:date"/>
```

Type complexe

- empty elements
- elements that contain only other elements
- elements that contain only text
- elements that contain both other elements and text

A l'intérieur d'un `<xs:complexType>` on a toujours soit :

- Un `<xs:choice name="" type="">`
- Un `<xs:sequence name="" type="">`
- => Qui contienne des `<xs:element>`.

Et ensuite des `<xs:attribute name="" type="">`

Exemple

Complex Example + **héritage**

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="professor" type="fullpersoninfo">

<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Empty Element

```
<xs:complexType name="prodtype">
  <xs:attribute name="prodid" type="xs:positiveInteger"/>
</xs:complexType>
```

Text only + attribute

```
<xs:complexType name="shoetype">
  <xs:simpleContent>
    <xs:extension base="xs:integer">
      <xs:attribute name="country" type="xs:string" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Integer only + Attirbute + attribute restriction

```
<xs:complexType>
  <xs:simpleContent>
    <xs:extension base="xs:integer">
      <xs:attribute name="unit">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:length value="3" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Restriction

Sur le nombre d'élément max

On utilise `minOccurs="0" maxOccurs="unbounded"` dans les `xs:element`, `xs:choice`, `xs:sequence`.

Unicité

On créer une clé pour garantir l'unicité d'un élement.

```
<xs:element name='clients'>
    <xs:complexType>
    <xs:sequence>
        <xs:element name='client' minOccurs='0' maxOccurs='unbounded'>
            <xs:complexType name='ClType'>
                <xs:sequence>
                    <xs:element name='nom' type='xs:string' />
                    <xs:element name='prenom' type='xs:string' />
                    <xs:element name='dateNaissance' type='xs:string' />
                </xs:sequence>
                <xs:attribute name='clientID' type='xs:integer' />
            </xs:complexType>
        </xs:element>
    </xs:sequence>
    </xs:complexType>

    <!-- /!\ Placement important /!\ -->
    <xs:key name="clientUnique">
        <xs:selector xpath="client"/>
        <xs:field xpath="@clientId"/>
    </xs:key>
</xs:element>
```

Sur le contenu

Il faut recreer un type simple qui contient une restriction

```
<xs:simpleType name="costInt">
    <xs:restriction base="xs:integer">
        <xs:minInclusive value="10"></xs:minInclusive>
    </xs:restriction>
</xs:simpleType>
```

- `xs:minInclusive`

- xs:maxInclusive
- xs:minExclusive
- xs:maxExclusive
- xs:enumeration
- xs:pattern
- xs:whiteSpace
- xs:length
- xs:minLength
- xs:maxLength
- xs:totalDigits
- xs:fractionDigits

Min-max

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Enumeration

```
<xs:element name="car" type="carType"/>

<xs:simpleType name="carType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Audi"/>
    <xs:enumeration value="Golf"/>
    <xs:enumeration value="BMW"/>
  </xs:restriction>
</xs:simpleType>
```

Ref

- Les key et keyref la racine du parent commun. @ lorsqu'on pointe sur un attribut

```
<!-- Ville : Key -->
<xs:key name="idVille">
    <xs:selector xpath="ville"></xs:selector>
    <xs:field xpath="@nom"></xs:field>
</xs:key>

<!-- Restaurant Ref ville -->
<xs:keyref refer="idVille" name="refVille">
    <xs:selector xpath="restaurant"></xs:selector>
    <xs:field xpath="@ville"></xs:field>
</xs:keyref>
```

```
            {$t}
            {$a}
        </result>
}</result>
```

Equivalent à

```
<results>{
  for $b in //book
  return (
    for $a in $b/author
      return (
        <result>
         {$b/title}
         {$a}
        </result>
      )
  )
}</results>
```

For loop

The For Clause To loop a specific number of times in a for clause, you may use the to keyword.

This

```
for $x in (1 to 5)
return <test>{$x}</test>
```

Returns

```
<test>1</test>
<test>2</test>
<test>3</test>
<test>4</test>
<test>5</test>
```

To count the iteration use the at keyword

This

```
for $x at $i in doc("books.xml")/bookstore/book/title
return <book>{$i}. {data($x)}</book>
```

Returns

XPath

- // au début pour chercher parmis tous l'arbre DOM
- / pour faire un chemin dans l'arbre DOM
- .. pour accédé au parent, attention un attribue est un fils dans l'arbre DOM
- `element[condition and condition or nor(condition)]`
- `A!=N` est vrai si la valeur textuelle d'un des noeuds est différente de la valeur textuelle de A.
- fonction **contains()** : `//restaurant[contains(menu/@nom, @ville)]` mais si il y a plusieurs menu il est pas sûr du comportement de contain donc pour être sur on fait `//restaurant[menu[contain(@nom, ../@ville)]]`
- fonction **count(menu)**
- Les foreign key : `//ville[@nom = //restaurant/@ville[count(menu) >= 4]]` ou `//restaurant[@ville = //ville[count(plusBeauMonument) = 0]/@nom`
- **Position**,
  - Plus safe d'utiliser à partir de `descendant::menu[5]` plutôt que `//=descendant-or-self::` :
    - `descendant::menu[5]` 5ème menu du document : **pas de contexte self**
    - `descendant-or-self::menu[5]` Le 5ème menu de chaque restaurant : **avec contexte self**
  - `last()` pour le dernier element mais attention c'est particulier aussi
  - `preceding-sibling`, ect ... voir image pour les voisins
- `//ville[@nom = //restaurant/@ville[count(./menu) >= 4]]`

XQuery

Classiquement : **for — let — where - order by - return**

```
<results>{
  for $r in //restaurant
  where $r/@etoile = 2
  return (

  )
}</results>
```

Imbriquer les boucles

```
<result>{
    for $b in //book, $t in $b/title, $a in $b/author
    return
        <result>
```

```
<book>1. Everyday Italian</book>
<book>2. Harry Potter</book>
<book>3. XQuery Kick Start</book>
<book>4. Learning XML</book>
```

it is also allowed with more than one expression in the for clause. Use comma to separate each in expression.

This

```
for $x in (10,20), $y in (100,200)
return <test>x={$x} and y={$y}</test>
```

Returns

```
<test>x=10 and y=100</test>
<test>x=10 and y=200</test>
<test>x=20 and y=100</test>
<test>x=20 and y=200</test>
```

Jointure

```
<books-with-prices>{
  for $b in /bib/book,
      $r in  /reviews/entry
  where $b/title=$r/title
  return <book-with-prices>
      {$b/title}
      <price-review>{$r/price/text()}</price-review>
      <price-bib>{$b/price/text()}</price-bib>
 </book-with-prices>
}
</books-with-prices>
```

Order by

Classique

```
for $x in doc("books.xml")/bookstore/book
order by $x/@category, $x/title
return $x/title
```

Expression conditionnel

```
<books>{
  for $x in //book
```

```
  return
    <book>
      { $x/title }
      est {
        if ($x/@year > 1999)
        then "récent"
        else "ancient"
      }
    </book>
}
</books>
```

Some et Exist DIAPO 32

Autre truc maybe utile

Opérateurs séquences:

- union: union
- différence : except
- intersection : intersect
- Distinct-value() : concat les str de l'objet pour faire un id DIAPO 38 Fonctions : count(), last(), first(), contains()...

RDF

- Ensemble de triplet type : Sujet x Prédicat x Object.
- Peux être sous la forme d'un graph
- Syntaxe Turtle = factoriser les triplet

Factorisation

On factorise comme ça :

- s p1 o1 + s p2 o2 -> s p1 o1 ; p2 o2
- s p o1 + s p o2 -> s p o1, o2

Développement

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .
<#green-goblin>
rel:enemyOf <#spiderman> ;
a foaf:Person ; # in the context of the Marvel universe
foaf:name "Green Goblin" .
<#spiderman>
rel:enemyOf <#green-goblin> ;
a foaf:Person ;
foaf:name "Spiderman", "Человек-паук"@ru .
```

Donne :

Sujet	Prédicat/propriété	Objet
<#green-goblin>	rel:enemyOf	<#spiderman>
<#green-goblin>	a	foof:Person
<#green-goblin>	foaf:name	"Green goblin"
<#spiderman>	rel:enemyOf	<#green-goblin>
<#spiderman>	a	foof:Person
<#spiderman>	foaf:name	"Spiderman"
<#spiderman>	foaf:name	"dsgosqg"@run

SPARQL

```
SELECT $variable
WHERE {MOTIF}
ORDER | LIMIT | OFFSET
```

Tout tourne autour des motifs

- `.` le point est un "et"
- `UNION` l'union des deux cotés, c'est comme un "OU", opérateur prioritaire sur le `.`
- `OPTIONAL ...` "et éventuellement un ...
- `FILTER (! bound(?l))` retire les lignes avec des cases vides
- `FILTER (Condition)` pour mettre des conditions sur les variables

Exemple

Les personnes qui ont étudié dans une université différente que cette de leur père ou leur mère

```
SELECT
{
    ?p :studiedAt ?u.
    ?p :hasFather :f.
    ?f :studiedAt :u1.
    ?p :hasMother :m.
    :m :studiedAt :u2.
    FILTER(?u != ?u1 && ?u != ?u2)
}
```

Les villes citées dans la DB

```
SELECT DISTINCT $v
{
    {?x :livesIn ?v} UNION
    {?y :locatedAt ?v} UNION
    {?v a :city}
}
```

Imprimer le TD en vrais de vrais

N1QL/JSON

- `ARRAY_LENGTH()` Taille d'une liste imbriqué
- **Pas de sous requête**
- **Penser au `INTERSECT`, `EXECPT` (DIFFERENCE en SQL)**

ANY & EVERY

- `ANY (...c IN continent...) SATISFIES (...condition...) END`
- `EVERY ... SATISFIES ... END`
- Condition dans les listes : donc s'utilise sur des listes uniquement !
- Pas de `ANY` ou `EVERY` à la racine d'un doc => penser aux jointures : intersect, except, ...
- Ces deux trucs en faite duplique chaque résultat comme avec un produit cartésiens pour ensuite filtrer ligne par ligne

Exemple

Les noms des pays qui se trouve *exactement* sur deux continents et où la couverture sur l'un des continents dépasse 50%

```
SELECT name
FROM country
WHERE
    ARRAY_LENGTH(continents) = 2
    AND
    ANY c IN continent SATISFIES (continents.percentage > 50) END
```

Les noms des pays dont toutes les frontières sont supérieures à 100km

```
SELECT name
FROM country
WHERE EVERY n IN neighbors SATISFIES n.length > 100
```

UNNEST

- Comme un `table()` de SQL3, quand on doit plonger dans un liste de sous-objet
- Assez space 😕

Exemple

Les nom des continents sans doublons

```
SELECT DISTINCT c.continent
FROM Country UNNEST continents AS c
```

Compréhension de liste

- Pour recréer une liste rapidement en résultat
- `ARRAY ... FOR ... IN ... END`

Les nom des pays se trouvant sur plus d'un continent avec la liste des noms de leurs continents et le nombre de leurs voisins

```
SELECT name, ARRAY c.continent FOR c ON continents END, ARRAY_LENGTH(neighbots) A
FROM Country
WHERE ARRAY_LENGTH(continents) > 1
```

Array Aggregate

- S'utilise avec `GROUP BY`
- Ca transforme en liste

Exemple : Pour les organisation ayant plus de 4 pays, leurs noms, la liste des noms des pays membres ainsi que la somme des populations de ces pays

Org	c.nom	c.population
UN	Country1	1142
	Country2	21432
	...	123123

=> Le ARRAY_AGG transforme la colonne c.nom en une liste. => Le COUNT compte le nombre de ligne par org => Le SUM somme sur c.population

Jointure

- Il faut potentiellement toujours utiliser INNER JOIN et pas que JOIN tout seul pour pas qu'il confonde avec la syntaxe de JOIN ON KEY
- Syntaxe :

```
SELECT name, capital, d.deserts
FROM Country c
  INNER JOIN Deserts m ON (c.name = d.country)
```