XML

Extensible Markup Language

Data Model: Tree

Node Types: Document, `<Element>`, `Attribute=""`, `Text`, `<?Processing Instruction>`, `<!-- Comment -->`

```
<?xml version="1.0" encoding="UTF-8"?>
<products>
  <product id="17">
    <description>chocolate bar</description>
    <price>0.89</price>                <!-- Todo: Increase price-->
  </product>
  <product id="29">
    <description>dishwasher tabs</description>
    <price currency="EUR">3.99</price>
    <sold_out/>
  </product>
</products>
```

The XML document starts with a prolog which contains the XML version and more.

XML Elements and Attributes

<Elements>

- start with an opening tag, end with a closing tag
  `<price>``0.89`**`</price>`**
  abbreviation for empty elements: **`<sold_out/>`**

- have a *set* of attributes
  `<price` **`currency="EUR"`**`>0.89</price>`

- have contents (children):
  - text: `0.89`
  - elements (*sequence* of nested elements)
  - mixed content (text and elements)

`<sold_out/>` is an abbreviation for `<sold_out></sold_out>`
The attributes of an element are unordered, the children of an element have an order.

Well-formedness, validity, DTD

When a document follows all rules of XML (e.g., each opening tag has to be closed), it is **well-formed**.

When a document matches a given schema (e.g., a DTD or an XML Schema (XSD)), it is **valid**.

DTD (Document Type Definition)

```
<!ELEMENT products (product*)>
<!ELEMENT product (description,price,sold_out?)>
<!ATTLIST product id CDATA #REQUIRED>
<!ELEMENT description (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ATTLIST price currency CDATA #IMPLIED>
<!ELEMENT sold_out EMPTY>
```

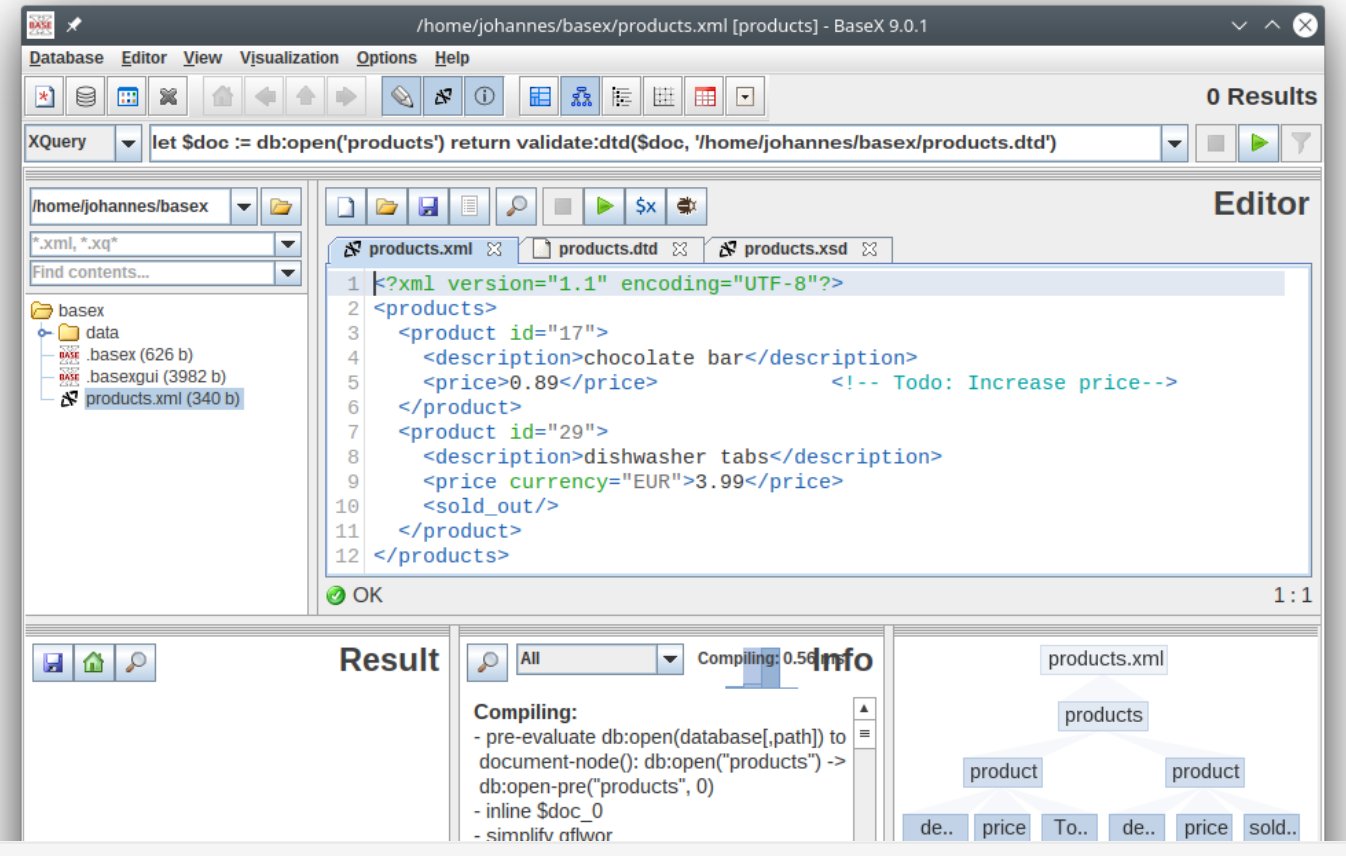
ELEMENT defines which elements are allowed to exist in the XML document and which content they can have. | is an or-choice, ? stands for optional, * arbitrary many times, + at least once. #PCDATA stands for text content, CDATA is text content for attributes.

XML Schema (XSD)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="products">
<xs:complexType>
  <xs:sequence>
    <xs:element name="product" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="description" type="xs:string"/>
          <xs:element name="price">
            <xs:complexType><xs:simpleContent>
                <xs:extension base="xs:decimal">
                  <xs:attribute name="currency" type="xs:string"/>
                </xs:extension>
            </xs:simpleContent></xs:complexType>
          </xs:element>
          <xs:element name="sold_out" minOccurs="0"><xs:complexType/></xs
        </xs:sequence>
        <xs:attribute name="id" type="xs:integer" use="required"/>
```

XSD (XML Schema Definition) is a more powerful language for defining a schema than DTD. In contrast to DTD, an XSD is a well-formed and valid XML document. XML schema supports data types, namespaces (e.g. `xs:`), user-defined data types, keys and key references, and much more.

BaseX (XML DBMS)



After creating a database (Database → New) by selecting an XML file, this XML database can be queried using XPath or XQuery (see next slides). The XQuery example in this screenshot shows how to validate an XML document to a DTD. With `validate:xsd`, the same can be done for an XML Schema. Mind that it is required to drop and re-create the database, when the XML file is changed.

XPath

Query language to traverse an XML document tree step by step.

Data Model: List of Nodes (Elements, Attributes, Texts, ...)

- **Path Expression**: `/step/step/step/...`
- **step**: `axis::node-test[predicate]`
- **axis**: `child` (default), `descendant-or-self` (`//`), `parent` (`..`), ...
- **node-test**: name test, `*`, `text()`, ..
- **predicate**, e.g. `[@id="17" or description="cat food"]`

```
//product[price>1]/description/text()
```

≙

```
/descendant-or-self::product[child::price/child::text()>1]
 /child::description/child::text()
```

The axis defines the traversal direction for each step, the default axis is traversing to the child nodes of the current node. `//` finds the current node, its children, grand-children, and so on. The example expression on this slide finds all descriptions of products that cost more than 1 EUR (or USD, ...). `@id` stands for the attribute `id` and `description` for the child element `description`. The result on an XPath expression is a list of nodes. For the given example, the result is a list of one text node `dishwasher tabs`.

XQuery

FLWOR Expressions

```
for $forvar in <Expr>
let $letvar := <Expr>
where <BoolExpr>
order by <Expr> ascending|descending
return <Expr>
```

Descriptions of products that cost more than 1 EUR (or USD,...)

```
let $doc := db:open('products')
for $p in $doc//product
where $p/price > 1
return $p/description/text()
```

`for` and `let` can be used multiple times in any order. `for` iterates over the nodes found by the given expression, and binds one node in each iteration to the given variable. `let` just binds the whole result to the variable. An optional `where` and `order by` clause can be used in the same way as in SQL, the `return` clause defines how to build the result.

XQuery

```
<expensive_products>
{
  let $doc := db:open('products')
  for $p in $doc//product
  let $price := switch($p/price/@currency)
                case "EUR" return $p/price*1.19
                default return $p/price
  where $price > 0.5
  order by $p/description
  return <product price="{fn:round($price,2)}">
      { $p/description/text() }</product>
}
</expensive_products>
```

Result:

```
<expensive_products>
  <product price="0.89">chocolate bar</product>
  <product price="4.75">dishwasher tabs</product>
</expensive_products>
```

This query finds all products that cost more than 0.50 USD and creates a new XML document as its output.

JSON

JavaScript Object Notation

Data Types:

- string
- number
- boolean
- array
- object
- null

```
{
  "product_id": 17,
  "description": "chocolate bar",
  "price": 0.89,
  "categories": ["food", "sweets"],
  "manufacturer": {
    "company": "Monsterfood",
    "country": "USA"
  },
  "available": true,
  "size": null
}
```

JSONiq

```
jsoniq version "1.0";
[
  let $products :=
  [
    { "product_id": 17, "description": "chocolate bar",
      "price": 0.89 },
    { "product_id": 29, "description": "dishwasher tabs",
      "price": 3.99 }
  ]
  for $p in $products()
  where $p.price gt 0.5
  order by $p.description
  return $p.description
]
```

Try out at http://try.zorba.io/

There are many query languages for querying JSON data. The language shown here is called JSONiq and uses FLWOR expressions similar to XQuery.

JSONPath

```
$[?(@.price>1)].description
```

`$`	context element (e.g. root object or array)
`.attr`	access attribute in an object
`[n]`	access the n-th element in an array
`*`	access all attributes in an object or all elements in an array
`..attr`	recursively descendant search of an attribute
`?(predicate))`	filter expression
`@`	the current node (used in filter expressions)

Try out at https://jsonpath.curiousconcept.com/

In the given example, we access an array of product objects. Within the `[brackets]`, we access those elements of the array that fulfill the filter predicate `@.price>1`. For the object that are found, we access the description attribute. The result is an array of descriptions of products that have a price greater than 1.

JSON Schema

```
{
  "$id": "https://example.com/product.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Product",
  "type": "object",
  "required": [ "description" ],
  "properties": {
    "description": {
      "type": "string",
      "description": "A short description of the product"
    },
    "price": {
      "type": "number", "multipleOf": 0.01, "minimum": 0,
      "description": "The product price in USD",
    }
  }
}
```

Try out at https://www.jsonschemavalidator.net/

Similar to DTD, and XSD, JSON schema defines whether a JSON document is valid or not. In this example, each document must have the `description` field. In the `properties` declaration, validity criteria for the attributes are defined. A JSON schema is both for restricting JSON data to follow the schema, and as metadata that helps understanding JSON data.

BSON

Binary JSON

Storage format for more efficient scan-speed than JSON strings

```
{ "_id": 17, "description": "chocolate bar" }
```

length	type	field	value	type	field	length	value	EOD
45	16	_id\0	17	2	description\0	14	chocolate bar\0	\0
4	1	4	4	1	12	4	14	1
Bytes								

BSON was first used by the NoSQL database MongoDB. There, JSON documents are not stored as strings but in a form as shown on this slide. This allows a much faster search. The length field in the very beginning is useful to skip the whole document to directly jump to the next one. The same holds for string fields. The length indicator for string fields is useful to directly jump to the next attribute.

Summary

- Structured Data,
  semi-structured data,
  unstructured data
- CSV
- YAML
- XML: DTD, XSD, XPath, XQuery
- JSON: JSONiq, JSONPath, JSON Schema, BSON