CSC-318
Web Technology
(BSc CSIT, TU)

Ganesh Khatri
kh6ganesh@gmail.com

AJAX

- AJAX stands for Asynchronous JavaScript And XML
- AJAX is not a programming language
- AJAX is a technique for accessing web servers from a web page
- AJAX is a developer's dream, because you can:
  - Read data from a web server - after the page has loaded
  - Update a web page without reloading the page
  - Send data to a web server - in the background

XML HttpRequest / AJAX

- All modern browsers have a built-in XMLHttpRequest object to request data from a server.
- The XMLHttpRequest object is a developers dream, because you can:
  - Update a web page without reloading the page
  - Request data from a server - after the page has loaded
  - Receive data from a server - after the page has loaded
  - Send data to a server - in the background

Sending an XMLHttpRequest

- A common JavaScript syntax for using the XMLHttpRequest object looks much like this:

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
       // Typical action to be performed when the document is ready:
       document.getElementById("demo").innerHTML = xhttp.responseText;
    }
};
xhttp.open("GET", "filename", true);
xhttp.send();
```

Sending an XMLHttpRequest

- The first line in the example above creates an XMLHttpRequest object:

```
var xhttp = new XMLHttpRequest();
```

- The onreadystatechange property specifies a function to be executed every time the status of the XMLHttpRequest object changes:

```
xhttp.onreadystatechange = function()
```

- When readyState property is 4 and the status property is 200, the response is ready:

```
if (this.readyState == 4 && this.status == 200)
```

- The responseText property returns the server response as a text string.
- The text string can be used to update a web page:

XML Parser

- All major browsers have a built-in XML parser to access and manipulate XML.
- The XML DOM (Document Object Model) defines the properties and methods for accessing and editing XML.
- However, before an XML document can be accessed, it must be loaded into an XML DOM object.
- All modern browsers have a built-in XML parser that can convert text into an XML DOM object.

Parsing a Text String

- This example parses a text string into an XML DOM object, and extracts the info from it with JavaScript:

```
<script>
  var parser, xmlDoc firstTitle;
  var text = "<bookstore><book>" +
  "<title>Everyday Italian</title>" +
  "<author>Giada De Laurentiis</author>" +
  "<year>2005</year>" +
  "</book></bookstore>";

  parser = new DOMParser();
  xmlDoc = parser.parseFromString(text,"text/xml");
  firstTitle = xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
</script>
```

Parsing a Text String

- Example Explained

```
text = "<bookstore><book>" +
"<title>Everyday Italian</title>" +
"<author>Giada De Laurentiis</author>" +
"<year>2005</year>" +
"</book></bookstore>";
```

- A text string is defined:

```
parser = new DOMParser();
```

- The parser creates a new XML DOM object using the text string:

```
xmlDoc = parser.parseFromString(text,"text/xml");
```

The XMLHttpRequest Object

- The XMLHttpRequest Object has a built in XML Parser.
- The responseText property returns the response as a string.
- The responseXML property returns the response as an XML DOM object.
- If you want to use the response as an XML DOM object, you can use the responseXML property.

The XMLHttpRequest Object

```
function loadXMLDoc() {
  var xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200)
    {
      myFunction(this);
    }
  };
  xmlhttp.open("GET", "cd_catalog.xml", true);
  xmlhttp.send();
}
function myFunction(xml) {
  var x, i, xmlDoc, txt; xmlDoc = xml.responseXML;
  txt = "";
  x = xmlDoc.getElementsByTagName("ARTIST");
  for (i = 0; i< x.length; i++) {
    txt += x[i].childNodes[0].nodeValue + "<br>";
  }
  document.getElementById("demo").innerHTML = txt;
}
```

cd_catalog.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD>
    <TITLE>Hide your heart</TITLE>
    <ARTIST>Bonnie Tyler</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
  </CD>
</CATALOG>
```

XML DTD

- DTD stands for Document Type Definition.
- A DTD defines the structure and the legal elements and attributes of an XML document.

Valid XML Documents

- A "Valid" XML document is "Well Formed", as well as it conforms to the rules of a DTD:
- The DOCTYPE declaration above contains a reference to a DTD file.
- Note.dtd ========➔

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

XML DTD

- The DTD above is interpreted like this:
  - !DOCTYPE note - Defines that the root element of the document is note
  - !ELEMENT note - Defines that the note element must contain the elements: "to, from, heading, body"
  - !ELEMENT to - Defines the to element to be of type "#PCDATA"
  - !ELEMENT from - Defines the from element to be of type "#PCDATA"
  - !ELEMENT heading - Defines the heading element to be of type "#PCDATA"
  - !ELEMENT body - Defines the body element to be of type "#PCDATA"
- Tip: #PCDATA means parseable character data.

Using DTD for Entity Declaration

- A DOCTYPE declaration can also be used to define special characters or strings, used in the document:
- Tip: An entity has three parts: it starts with an ampersand (&), then comes the entity name, and it ends with a semicolon (;).

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE note [
<!ENTITY nbsp " ">
<!ENTITY writer "Writer: Donald Duck.">
<!ENTITY copyright "Copyright: W3Schools.">
]>

<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
<footer>&writer; &copyright;</footer>
</note>
```

When to Use a DTD?

- With a DTD, independent groups of people can agree to use a standard DTD for interchanging data.
- With a DTD, you can verify that the data you receive from the outside world is valid.
- You can also use a DTD to verify your own data

When NOT to Use a DTD?

- XML does not require a DTD.
- When you are experimenting with XML, or when you are working with small XML files, creating DTDs may be a waste of time.
- If you develop applications, wait until the specification is stable before you add a DTD. Otherwise, your software might stop working because of validation errors.

XML Schema Definition (XSD)

- An XML Schema describes the structure of an XML document, just like a DTD.
- An XML document with correct syntax is called "Well Formed".
- An XML document validated against an XML Schema is both "Well Formed" and "Valid"
- XML Schema is an XML-based alternative to DTD:

XML Schema : Example

- <xs:element name="note"> defines the element called "note"
- <xs:complexType> the "note" element is a complex type
- <xs:sequence> the complex type is a sequence of elements
- <xs:element name="to" type="xs:string"> the element "to" is of type string (text)
- <xs:element name="from" type="xs:string"> the element "from" is of type string
- <xs:element name="heading" type="xs:string"> the element "heading" is of type string
- <xs:element name="body" type="xs:string"> the element "body" is of type string

```
<xs:element name="note">

<xs:complexType>
  <xs:sequence>
    <xs:element name="to" type="xs:string"/>
    <xs:element name="from" type="xs:string"/>
    <xs:element name="heading" type="xs:string"/>
    <xs:element name="body" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

</xs:element>
```

XML Schemas are More Powerful than DTD

- XML Schemas are written in XML
- XML Schemas are extensible to additions
- XML Schemas support data types
- XML Schemas support namespaces

- Why Use an XML Schema?
  - With XML Schema, your XML files can carry a description of its own format.
  - With XML Schema, independent groups of people can agree on a standard for interchanging data.
  - With XML Schema, you can verify data

XML Schemas Support Data Types

- One of the greatest strengths of XML Schemas is the support for data types:
  - It is easier to describe document content
  - It is easier to define restrictions on data
  - It is easier to validate the correctness of data
  - It is easier to convert data between different data types

XML Schemas use XML Syntax

- Another great strength about XML Schemas is that they are written in XML:
  - You don't have to learn a new language
  - You can use your XML editor to edit your Schema files
  - You can use your XML parser to parse your Schema files
  - You can manipulate your Schemas with the XML DOM
  - You can transform your Schemas with XSLT

XSD Simple Elements

- A simple element is an XML element that can contain only text. It cannot contain any other elements or attributes.

```
<xs:element name="xxx" type="yyy"/>
```

- where xxx is the name of the element and yyy is the data type of the element.
- XML Schema has a lot of built-in data types. The most common types are:
  - xs:string
  - xs:decimal
  - xs:integer
  - xs:boolean
  - xs:date
  - xs:time

XSD Simple Elements

- Here are some XML elements:

```
<lastname>Refsnes</lastname>
<age>36</age>
<dateborn>1970-03-27</dateborn>
```

- And here are the corresponding simple element definitions:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
```

XSD Complex Elements

- A complex element is an XML element that contains other elements and/or attributes.
- There are four kinds of complex elements:
  - empty elements
  - elements that contain only other elements
  - elements that contain only text
  - elements that contain both other elements and text
- **Note:** Each of these elements may contain attributes as well!

XSD Complex Elements

- Empty Complex Element

```
<product pid="1345"/>
```

- A complex XML element, "employee", which contains only other elements:

```
<employee>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</employee>
```

- A complex XML element, "food", which contains only text:

```
<food type="dessert">Ice cream</food>
```

- A complex XML element, "description", which contains both elements and text:

```
<description>
It happened on <date lang="norwegian">03.03.99</date> ....
</description>
```

XSD Attributes

- Simple elements cannot have attributes.
- If an element has attributes, it is considered to be of a complex type.
- But the attribute itself is always declared as a simple type.
- The syntax for defining an attribute is:

```
<xs:attribute name="xxx" type="yyy"/>
```

- where xxx is the name of the attribute and yyy specifies the data type of the attribute.

XSD Attributes

- XML Schema has a lot of built-in data types. The most common types are:
  - xs:string
  - xs:decimal
  - xs:integer
  - xs:boolean
  - xs:date
  - xs:time

```
<lastname lang="EN">Smith</lastname>
```

```
<xs:attribute name="lang" type="xs:string"/>
```

Default and Fixed Values for Attributes

- Attributes may have a default value OR a fixed value specified.
- A default value is automatically assigned to the attribute when no other value is specified.
- In the following example the default value is "EN":

```
<xs:attribute name="lang" type="xs:string" default="EN"/>
```

- A fixed value is also automatically assigned to the attribute, and you cannot specify another value.
- In the following example the fixed value is "EN":

```
<xs:attribute name="lang" type="xs:string" fixed="EN"/>
```

Optional and Required Attributes

- Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:

```
<xs:attribute name="lang" type="xs:string" use="required"/>
```

XSD Restrictions/Facets

- Restrictions are used to define acceptable values for XML elements or attributes.
- Restrictions on XML elements are called facets.

Restrictions on Values

- The following example defines an element called "age" with a restriction.
- The value of age cannot be lower than 0 or greater than 120:

```
<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>
```

Restrictions on a Set of Values

- To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint.
- The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:

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

Restrictions on a Series of Values

- To limit the content of an XML element to define a series of numbers or letters that can be used, we would use the pattern constraint.
- The example below defines an element called "letter" with a restriction. The only acceptable value is ONE of the LOWERCASE letters from a to z:

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Series of Values

- The next example defines an element called "initials" with a restriction. The only acceptable value is THREE of the UPPERCASE letters from a to z:

```
<xs:element name="initials">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z][A-Z][A-Z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Series of Values

- The next example also defines an element called "initials" with a restriction. The only acceptable value is THREE of the LOWERCASE OR UPPERCASE letters from a to z:

```
<xs:element name="initials">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Series of Values

- The next example defines an element called "choice" with a restriction. The only acceptable value is ONE of the following letters: x, y, OR z:

```
<xs:element name="choice">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[xyz]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Series of Values

- The next example defines an element called "prodid" with a restriction. The only acceptable value is FIVE digits in a sequence, and each digit must be in a range from 0 to 9:

```
<xs:element name="prodid">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on Whitespace Characters

- To specify how whitespace characters should be handled, we would use the whiteSpace constraint.
- This example defines an element called "address" with a restriction. The whiteSpace constraint is set to "preserve", which means that the XML processor WILL NOT remove any white space characters:

```
<xs:element name="address">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on Length

- To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints.
- This example defines an element called "password" with a restriction. The value must be exactly eight characters:

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```