Unit-3

The Client Tier

Unit 3 The Client Tier **10 Hrs.**

Representing Content; Introduction to XML; Elements and Attributes; Rules for Writing XML; Namespaces; Schema: Simple Types and Complex Types, XSD Attributes, Default and Fixed Values, Facets, Use of Patterns, Order Indicators(All, Choice, Sequences), Occurrence Indicators (Maxoccurs, Minoccurs), DTD: Internal Declaration, Private External Declaration, Public External Declaration, Defining Elements and Attributes; XSL/XSLT; Xpath; Xquery; SAX; DOM , Creating XML Parser.

Introduction to XML

XML stands for eXtensible Markup Language. It is a markup language that is designed to store, transport, and describe data in a structured format. XML is both human-readable and machine-readable, making it a versatile choice for representing various types of data. It does not have a predefined set of tags like HTML; instead, we can create our own tags to define the structure of our data.

XML uses a hierarchical structure of nested elements to organize and represent data. Each element consists of a start tag, content, and an end tag. Elements can also contain attributes that provide additional information about the element.

XML uses tags to mark up elements within a document. These tags define the structure and meaning of the data. Unlike HTML, which has predefined tags for specific purposes (like <p> for paragraphs), XML allows to create own tags based on the context of data.

Features of XML:

a. Markup Language: XML uses tags to mark up elements within a document. These tags define the structure and meaning of the data. Unlike HTML, which has predefined tags for specific purposes (like <p> for paragraphs), XML allows to create our own tags based on the context.

b. Hierarchy: XML documents have a hierarchical structure, similar to a tree. Elements can contain other elements, forming parent-child relationships. This hierarchy is one of the main reasons XML is so effective at representing structured data.
c. Elements and Attributes: Elements are the building blocks of an XML document. They consist of opening and closing tags, encasing the content they define. Elements can also have attributes, which are key-value pairs that provide additional information about the element.
d. Text Content: Elements can contain text content, which can be data, information, or descriptions. Text content is placed between the opening and closing tags of an element.
e. Well-Formedness and Validity: An XML document must be well-formed to be considered syntactically correct. This means it adheres to the rules of XML syntax, such as properly nested elements and correctly closed tags. Additionally, an XML document can be validated against a Document Type Definition (DTD) or an XML Schema to ensure it conforms to a specific structure and content.
f. Namespaces: XML namespaces allow to avoid naming conflicts when using XML from different sources or domains. They enable to differentiate elements and attributes that might have the same names but come from different contexts.
g. CDATA: CDATA (Character Data) sections are used to include blocks of text that should not be parsed as XML. This is useful when the content contains characters that are normally interpreted as XML markup.

XML is widely used for a variety of purposes, including:

a. Data Interchange: XML is used to exchange data between different systems and platforms. It provides a common format that can be understood by different programming languages and applications.

b. Configuration Files: Many software applications use XML for configuration files. These files define various settings and parameters that the application uses to function properly.

c. Web Services: XML is commonly used as the format for data exchanged between web services. It allows different systems to communicate and share data seamlessly.

d. Document Markup: While not as common as HTML, XML can also be used for marking up documents, especially when a custom structure is required.

e. Database Import/Export: XML can be used to export and import data from databases. It provides a standardized way to represent database records and structures.

Example:

```
<person>
    <name>John Doe</name>
    <age>30</age>
```

```
    <email>john@example.com</email>
</person>
```

In this example, <person>, <name>, <age>, and <email> are elements. The text content within the elements represents the data. Attributes can also be added to elements, providing additional information.

Overall, XML's flexibility, readability, and compatibility have made it a fundamental technology for data representation and interchange in a wide range of applications.

No.	HTML	XML
1)	HTML is used to display data and focuses on how data looks.	XML is a software and hardware independent tool used to transport and store data. It focuses on what data is.
2)	HTML is a markup language itself.	XML provides a framework to define markup languages.
3)	HTML is not case sensitive.	XML is case sensitive.
4)	HTML is a presentation language.	XML is neither a presentation language nor a programming language.
5)	HTML has its own predefined tags.	You can define tags according to your need.
6)	In HTML, it is not necessary to use a closing tag.	XML makes it mandatory to use a closing tag.
7)	HTML is static because it is used to display data.	XML is dynamic because it is used to transport data.
8)	HTML does not preserve whitespaces.	XML preserve whitespaces.

Elements and Attributes

Elements

Elements are the fundamental building blocks of an XML document. They define the structure and content of the data being represented. Each element consists of an opening tag, optional content, and a closing tag. The opening tag and closing tag must match and be properly nested. Elements can be hierarchical, allowing to create a structured representation of data.

Syntax:

```
<elementName>content</elementName>
```

- <elementName>: This is the opening tag of the element.
- </elementName>: This is the closing tag of the element.
- content: This is the content within the element. It can be text, other elements, or a combination of both.

Example:

```
<note>
  <date>2008-01-10</date>
  <to>Ram</to>
  <from>Sita</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Rules of XML Elements:

- Element names are case-sensitive.
- Element names must start with a letter or underscore.
- Element names cannot start with the letters xml (or XML, or Xml, etc).
- Element names can contain letters, digits, hyphens, underscores, and periods.
- Element names cannot contain spaces.

Attributes

Attributes provide additional information about an element. They are typically used to give extra details or metadata to an element. Attributes are always specified within the opening tag of an element. An attribute consists of a name and a value, separated by an equals sign (=) and enclosed in double or single quotes.

Syntax:

```
<elementName attributeName="attributeValue">content</elementName>
```

Example:

```
<product id="123" name="Smartphone" price="599.99">
    <description>Flagship Phone</description>
</product>
```

In this example, the <product> element has three attributes: id, name, and price. These attributes provide additional information about the product.

The choice between using attributes and elements to store data depends on the nature of the information. If the data forms an integral part of the content's structure, use elements. If the data is optional, metadata, or doesn't require further structure, use attributes.

Properties of Attributes:

- Uniqueness: Attributes within the same element must have unique names. An element cannot have two attributes with the same name.
- Value Encapsulation: Attribute values are typically enclosed in double or single quotes, like "attributeValue" or 'attributeValue'.
- Used to provide additional information about an element (metadata).
- Used to offer configuration details for software or systems.
- Used to convey options or settings for a specific element.

Rules for Writing XML

- Case Sensitivity: XML is case-sensitive. Tags and attribute names must use consistent casing throughout the document.

- Root Element: Every XML document must have a single root element that encapsulates all other elements.

- Nesting: Elements must be properly nested within each other. Opening and closing tags must match in order and hierarchy.

- Self-Closing Tags: Elements without content can be represented using self-closing tags, like <element />.

- Attribute Quoting: Attribute values must be enclosed in single or double quotes.

- Comments: Comments can be included using <!-- ... -->. Comments cannot be nested and cannot contain the string "--".

- Reserved Characters: Reserved characters like <, >, &, ", and ' cannot be used as-is in content. Use their respective character entities.

- Whitespace: Leading and trailing whitespace within elements' content is typically ignored. Whitespace within tags is preserved.

  Example:

  ```
  < element  attribute="value"  >  This is content      </element >
  ```

  Here, whitespace inside tag name is preserved. The whitespace inside the content is ignored but whitespace before and after content are ignored.

- Namespace Declarations: When using namespaces, define them in the opening tag of the element using the xmlns attribute.

- Consistency: Use consistent formatting, indentation, and naming conventions to improve code readability.

- Validation: Validate the XML against a schema (DTD or XML Schema) to ensure it adheres to a specified structure.

- Well-Formedness: Ensure the XML is well-formed, meaning it adheres to the basic syntax rules of XML.

- Order of Elements: The order of elements within their parent element is significant. Changing the order can change the meaning of the document.

- Attribute Order: The order of attributes within an opening tag is not significant.

- Empty Elements: For elements without content, use self-closing tags like <emptyElement />.

Namespaces

XML namespaces are a powerful mechanism used to avoid naming conflicts and ensure the uniqueness of element and attribute names within an XML document. They are particularly important when integrating XML content from different sources or domains.

In XML, an XML namespace is a way to uniquely identify elements and attributes to avoid naming collisions. This becomes crucial when multiple parties contribute to an XML document, and each party defines elements or attributes with the same names. By using namespaces, we can differentiate between these elements and attributes, even if they have the same names.

A namespace is defined using the xmlns attribute in the opening tag of an element. This attribute assigns a Uniform Resource Identifier (URI) to the namespace. The URI doesn't necessarily need to point to an actual resource; it serves as a unique identifier.

Consider the following two pieces of XML:

```
<table>
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.

Name conflicts in XML can easily be avoided using a name prefix. This XML carries information about an HTML table, and a piece of furniture:

```
<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

In the example above, there will be no conflict because the two <table> elements have different names.

When using prefixes in XML, a namespace for the prefix must be defined. The namespace can be defined by an xmlns attribute in the start tag of an element. The namespace declaration has the following syntax. xmlns:prefix="URI".

```
<root>

<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>
```

```
  <f:table xmlns:f="https://www.w3schools.com/furniture">
    <f:name>African Coffee Table</f:name>
    <f:width>80</f:width>
    <f:length>120</f:length>
  </f:table>

</root>
```

The xmlns attribute in the first <table> element gives the h: prefix a qualified namespace. The xmlns attribute in the second <table> element gives the f: prefix a qualified namespace. When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace. Namespaces can also be declared in the XML root element:

```
<root xmlns:h="http://www.w3.org/TR/html4/"
xmlns:f="https://www.w3schools.com/furniture">

<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

The namespace URI is not used by the parser to look up information. The purpose of using an URI is to give the namespace a unique name.

XML namespaces are typically defined using Uniform Resource Identifiers (URIs) and are indicated using namespace prefixes.

A namespace prefix is a short string (usually one to three characters) that is used as an abbreviation for the namespace URI. It is combined with a colon to create a qualified name for elements or attributes in XML documents. The qualified name consists of the namespace prefix and the local name, separated by a colon.

For example, consider an XML document that uses two namespaces: one for books and another for authors. The prefixes "bk" and "auth" could be used to distinguish between elements from the two namespaces:

```
<library xmlns:bk="http://example.com/books"
xmlns:auth="http://example.com/authors">
  <bk:book>
    <bk:title>XML Basics</bk:title>
    <auth:author>John Doe</auth:author>
  </bk:book>
</library>
```

In this example, the prefixes "bk" and "auth" are used to associate elements with their respective namespaces.

Features of XML Namespace

XML namespaces are used to avoid naming conflicts in XML documents, especially when multiple vocabularies or schemas are combined within a single document. They provide a way to uniquely identify elements and attributes from different sources.

Here are the key features of XML namespaces:

- Declaration: XML namespaces are declared using namespace declarations in the root element of an XML document or in the opening tag of an element. These declarations associate a namespace prefix with a namespace URI.

```
<rootElement xmlns:prefix="namespaceURI">
  <!-- Elements and attributes within this scope can use the
  prefix to reference the namespace -->
</rootElement>
```

- Uniqueness: XML namespaces ensure that element and attribute names are globally unique by associating them with a namespace URI. This prevents naming conflicts when different XML vocabularies are combined.

- Namespace URI: Each XML namespace is identified by a Uniform Resource Identifier (URI). This URI can be a URL or any other valid URI, and it serves as a unique identifier for the namespace.

- Prefixing: XML namespaces allow to use short prefix strings to associate elements and attributes with specific namespaces. This makes XML documents more readable and manageable, especially when dealing with multiple namespaces.

```
<library xmlns:bk="http://example.com/books"
xmlns:auth="http://example.com/authors">
  <bk:book>
    <bk:title>XML Basics</bk:title>
    <auth:author>John Doe</auth:author>
```

```
  </bk:book>
</library>
```

- Inheritance: XML namespaces can be inherited by child elements. If an element declares a namespace, its child elements can use the same namespace without explicitly declaring it again. Above XML can also be written as follows:

```
<library xmlns:bk="http://example.com/books">
  <book>
    <title>XML Basics</title>
  </book>
  <author>John Doe</author>
</library>
```

- XML Schema Integration: XML namespaces are commonly used when working with XML Schema (XSD) to define complex data structures.

- Interoperability: XML namespaces facilitate interoperability by allowing different organizations and standards to define their own elements and attributes within separate namespaces, while still allowing them to be used together in a single document.

- Namespace in Attributes: Namespaces can be applied to both elements and attributes in an XML document. This allows for precise identification and differentiation of data within the document.

```
<library xmlns:bk="http://example.com/books"
xmlns:auth="http://example.com/authors">
  <book bk:isbn="123456789">
    <title>XML Basics</title>
    <auth:author bk:role="writer">John Doe</auth:author>
  </book>
</library>
```

Schema/ XML Schema Definition (XSD)

XML Schema is a specification used for defining the structure, content, and data types of XML documents. XML Schema serves as a blueprint or contract that documents must be obeyed, to ensure validity of XML document.

XML Schema provides a range of built-in data types (e.g., string, integer, date, boolean) that we can use to specify the data type of element content and attributes. Additionally, we can create your custom data types by extending or restricting existing types.

XML Schema allows you to validate XML documents against the defined schema. Validation checks if an XML document conforms to the rules and constraints specified in the schema. If validation fails, errors are reported, indicating where the document deviates from the schema.

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".

An XML Schema describes the structure of an XML document. XML Schema is commonly known as XML Schema Definition (XSD). It is used to describe and validate the structure and the content of XML data. XSDs check the validity of structure and vocabulary of an XML document against the grammatical rules of the appropriate XML language. XML schema defines the elements, attributes and data types. Schema element supports Namespaces. It is similar to a database schema that describes the data in a database.

Properties of XML Schema

- XML Schema allows to define the structure of XML documents by specifying elements, attributes, and their relationships, providing a clear blueprint for document organization.
- It supports a wide range of built-in data types (e.g., string, integer, date, boolean) that can be used to specify the data type of element content and attributes.
- Custom data types can also be created by extending or restricting existing types.
- XML Schema enables the validation of XML documents against the defined schema.
- XML Schema supports namespaces.
- Simple types are used for defining the data type and constraints of element content. They can impose constraints like minimum and maximum length, pattern matching, and enumeration of values.
- XML Schema allows the definition of complex types, which describe the content model of elements.
- XML Schema supports mechanisms for importing and including external schema definitions.
- XML Schema itself is expressed in XML, making it human-readable and easy to manipulate using XML tools.

Well Formed and Valid XML

Well Formed XML	Valid XML
Well-formed XML refers to an XML document that adheres to the basic syntax rules and structural requirements of XML.	Valid XML not only meets the well-formedness criteria but also conforms to a specific Document Type Definition (DTD) or XML Schema Definition (XSD).
Well formed XML is independent to any external specification.	XML document is validated against some external specification.
A well formed XML may or may not be valid.	A valid XML is always Well Formed.

Check of well-formedness involves checking for the basic rules and syntax of writing XML.	Validation involves checking a document for well-formedness as well as conformance to a schema.

Simple Type and Complex Type

XSD Type definitions are used to create new simple data type or complex data type. A type definition that is used as a base for creating new definitions is known as the base type definition. A simple Type or complex Type can be either named or anonymous. A named simple Type or complex Type is always defined globally. We can define a custom simple type or complex type and refer it in schema document as type of an element.

a) Simple Type:

The simple Type allows to have text-based elements. It contains less attributes, child elements, and cannot be left empty. Simple type element is used only in the context of the text. Some of the predefined simple types are: xs:integer, xs:boolean, xs:string, xs:date etc. XML schema has several built-in data types. A simple Type type is derived from an XML schema built-in data type.

In XML Schema, you can define simple types using the <simpleType> element. Simple types are used to specify the data type and constraints for elements or attributes in an XML document.

Features of simpleType:

- Simple types are used to define the data type of individual elements or attributes within an XML document.
- They specify the format and constraints for the data contained within an element or attribute.
- Common simple data types include:
  string: Used for text data.
  integer: Used for whole numbers.
  boolean: Used for true/false values.
  date/time types: Used for date and time values.
  decimal: Used for decimal numbers.
- We can also define custom simple types by restricting or extending existing simple types. For example, we can create a custom type called "positiveInteger" that restricts the integer type to only allow positive numbers.

Here is the basic syntax for defining a simple type in an XML Schema:

Syntax:

```
<xs:simpleType name="TypeName">
    <!-- Define restrictions or enumeration values here -->
</xs:simpleType>
```

Example: let us create a XML schema that contains a simple type “positiveInteger”.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="positiveInteger">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='dob'>
    <xs:restriction base='date'>
      <xs:minInclusive value='2060/01/01'>
    </xs:restriction>
  </xs:simpleType>


  <xs:element name="age" type="positiveInteger">
  <xs:element name="birthdate" type="dob">
  <xs:element name="name" type="string">

</xs:schema>
```

Where,

- <xs:simpleType>: This is the root element for defining a simple type. Here, xs is the namespace prefix.
- name="TypeName": This is an optional attribute that specifies the name of the simple type. we can choose a name for our type, and it should be unique within the scope of your schema.
- Following are the different possible built in simple types:

  xs:string: Represents textual data.
  xs:boolean: Represents boolean values (true or false).
  xs:decimal: Represents numbers with a decimal point.
  xs:integer: Represents whole numbers without a decimal point.
  xs:date: Represents date values in the format "YYYY-MM-DD."
  xs:time: Represents time values in the format "hh:mm:ss."
  xs:float: Represents single-precision floating-point numbers.
  xs:double: Represents double-precision floating-point numbers.

b) Complex Type:

In XML Schema, a complex type is used to define the structure and content model of XML elements that can contain other elements and attributes. Complex types describe the composition of elements within a complex element and can have a variety of characteristics, including sequence, choice, and all.

Complex elements are those that can contain other elements and attributes, as opposed to simple elements, which contain only text or atomic data.

Complex types can include;

- Sequence: Specifies that child elements must appear in a specific order.
- Choice: Specifies that one of several child elements must appear.
- All: Specifies that child elements can appear in any order.
- Attributes: Defines attributes that the complex element can have.

In summary, complex types in XML Schema provide a powerful mechanism for defining the structure and content rules for XML elements that contain other elements and attributes. They allow you to create structured and hierarchical data models for your XML documents, making it easier to validate and work with complex XML data.

Syntax:

```
<xs:complexType name="TypeName">
    <!-- Define the structure and content model here -->
</xs:complexType>
```

Example: Let us create a schema for complex type "person".

```
<xs:complexType name="Person">
  <xs:sequence>
     <xs:element name="FirstName" type="xs:string"/>
     <xs:element name="LastName" type="xs:string"/>
     <xs:element name="Address" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:element name='student' type='Person'>
<xs:element name="teacher" type="Person">
```

Following XML is valid according to above schema.

```
<student>
     <FirstName>Ram</FirstName>
     <LastName>Adhikar</LastName>
     <Address>Pokhara</Address>
</student>
</teacher>Hari Prasad</teacher>  ➔ Invalid
```

Simple Type	Complex Type
Simple types are used to define the constraints and allowed values for the **content** of XML elements.	Complex types are used to define the **structure** and content model of XML elements that can have nested elements and attributes.
They are primarily **used for elements that contain text data** (e.g., strings, numbers, dates) and don't have nested elements or attributes.	They are **used for elements that contain other** elements, have attributes or have complex structures.
They **include constraints** such as data types (e.g., string, integer, date) and facets (e.g., length, minInclusive, maxExclusive) that further restrict the allowed values.	Complex types **define the structure** of elements, including child elements and their occurrence constraints. They can specify • whether an element can have attributes and define attribute constraints. • Allowed child for an element and their order or occurrence etc.
Simple type definition may use attributes like name, type etc. but the corresponding XML elements have no child and attributes.	Complex types **can define attributes** for elements, including their data types and constraints.

Example:
`<xs:element name="age" type="xs:integer"/>`
	Example:
	`<xs:element name="person">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element name="name"type="xs:string"/>`
	`<xs:element name="age" type="xs:integer"/>`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`

Example: XSD that defines the book element.

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

XML document validated by the above XSD is as follows:

```
<book>
  <title>Web Technology</title>
  <athor>Ramesh</author>
  <price>500</price>
</book>
```

Example: XSD that defines the class element.

```
<xs:schema xmlns:xs="http://www.w3.org/2003/xmlschema">
  <xs:element name="class">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="student" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="firstname" type="xs:string"/>
              <xs:element name="lastname" type="xs:string"/>
              <xs:element name="age" type="xs:int"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Following is a XML document that is valid according to above XSD:

```
<class>
  <student>
    <firstname>Ramesh</firstname>
    <lastname>Thapa</lastname>
    <age>20</age>
  </student>
  <student>
    <firstname>Suresh</firstname>
    <lastname>Thapa</lastname>
    <age>20</age>
  </student>

</class>
```

Rules of Defining XSD

- Use <xs:schema xmlns:xs="URI"> as first element.

- We can use the <xs:enumeration> element within a custom simple type to specify a list of allowed values for the element.
- Simple type elements are typically used as part of complex types. You can define complex types with the <xs:complexType> element and include simple type elements within them.
- Use any one of the following syntaxes to define simple type element.

Defining normal simple type	Defining custom simple type
<xs:element name="tag_name" type="simple_type"> **Example:** <xs:element name='fname' type='string>	<xs:simpleType name="tag_name"> <xs:restriction base="data_type"> <!-- constraints --> </xs:restrictions> </xs:simpleType>
• It doesn't define new simple type, rather it uses simple types such as integer, float, double, string, character etc. • It is not possible to define restrictions on content in this way.	• Basic data_type (int, char etc.) are used to define new custom data type. • The custom simple type defined in this way can be used further by other complex types as the value of type attribute. • We can specify different types of restrictions on the content in this way.

Example: XSD defining a simple type element <temperature> with restrictions.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Define a simple type named "Temperature" with restrictions -->

  <xs:simpleType name="Temperature">
    <xs:restriction base="xs:decimal">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
      <xs:fractionDigits value="2"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:element name='fever' type='Temperature'>
</xs:schema>
```

Valid XML as per above XSD:

```
<fever>-5</fever>     ->  invalid because -5
<fever>99</fever>     -> valid
```

Example: XSD defining a complex type element <weather>.

```
<!-- Use the "Temperature" simple type in an element -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="weather">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="temperature" type="Temperature"/>
        <!-- Other elements can be included in the complex type -->
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Valid XML as per above XSD:

```
<weather>
  <temperature>20</temperature>
</weather>
```

XSD Attributes (Default and Fixed Values)

In XML Schema Definition (XSD), attributes are used to define additional information about an element. Attributes provide metadata or properties associated with an XML element, and they are declared within the XML schema to specify the structure and constraints of an XML document.

We use the following rules to define an attribute in an XML Schema:

a. **Declaration:** To define an attribute in an XML schema, we use the <xs:attribute> element. This element is typically placed within the complex type or simple type definition that corresponds to the element to which the attribute is associated.

Syntax:

```
<xs:attribute
  name = "attributeName"
  type = "dataType"
  default = "defaultValue"
  fixed = "fixedValue"
  use = ("optional" | "required" | "prohibited")
  ...
</xs:attribute>
```

We can define the attributes using following different keywords:

Keyword	Meaning
default	To define a default fallback value for an attribute.
Fixed	To define the fixed value for an attribute.
Use	Optional: the attribute may or may not present Required: the attribute must present Prohibited: the attribute cannot present

Example: Consider an XSD for validating a complex book element.

```
<xs:element name="book">
  <xs:complexType>
    <xs:attribute name="isbn" type="xs:string" required/>
    <xs:attribute name= "language" type= "xs:string" default= "english">
    <xs:attribute name= "version" type= "xs:string" fixed = "1.0">
    <!-- Other elements and attributes within the 'book' element -->
  </xs:complexType>
</xs:element>
```

XML as per above schema:

```
<book isbn="123abc456" langage="Nepali" version="1.0"> Web Technology Book </book>
```

Facets:

In XML Schema Definition (XSD), facets are used to further constrain or specify the allowable values for a simple type. Simple types in XSD represent data types for elements and attributes that contain text content. Facets provide a way to define rules or constraints on these simple types. There are several facets available in XSD, each serving a specific purpose.

Here are some commonly used facets:

a. **Enumeration (xs:enumeration):** This facet allows to specify a list of allowed values for a *simple type*. The value of the element or attribute must match one of the values listed in the enumeration.

Example:

```
<xs:simpleType name="Color">
  <xs:restriction base="xs:string">
    <xs:enumeration value="red" />
    <xs:enumeration value="green" />
    <xs:enumeration value="blue" />
  </xs:restriction>
</xs:simpleType>

<xs:element name="dress" type="Color">
```

XML document valid as per above XSD:

<dress>Red</dress> it would be invalid if yellow

b. **Min Inclusive (xs:minInclusive) and Max Inclusive (xs:maxInclusive):** These facets set the minimum and maximum values that a numeric simple type can have. The values themselves are included in the allowed range.

Example:

```
<xs:simpleType name="PositiveInteger">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1" />
    <xs:maxInclusive value=”100”/>
  </xs:restriction>
</xs:simpleType>
```

c. **Length (xs:length), Min Length (xs:minLength), and Max Length (xs:maxLength):** These facets specify constraints on the length of a string value. length sets the exact length, minLength sets the minimum length, and maxLength sets the maximum length.

Example:

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

d. **Total Digits (xs:totalDigits) and Fraction Digits (xs:fractionDigits):** These facets are used with decimal and double types to specify the total number of digits and the number of decimal places allowed in a value.

Example:

```
<xs:simpleType name="Decimal">
  <xs:restriction base="xs:decimal">
    <xs:totalDigits value="8" />
    <xs:fractionDigits value="2" />
  </xs:restriction>
</xs:simpleType>
```

Use of Patterns

Pattern is a type of facet defined by <xs:pattern> element. In XML Schema Definition (XSD), it is a powerful tool for specifying regular expression patterns that values of a string type (like xs:string) must conform to. Patterns allow to define complex rules for the format and structure of string data.

Example:

```
<xs:element name="email">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-Z0-9._%+-]+@[a-zA-Z]+\.[a-zA-Z]{3}" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The value can be defined using the regular expression pattern.

Following are the major points to be remembered to define the regular expression as pattern:

Pattern	Meaning
[a-z]	Any single lowercase character.
[a-zA-Z]	Any single alphabet character.
[a-zA-Z0-9]	Any single alphanumeric character.
[a-z][A-Z][0-9]	A string of three characters. First being lowercase, second being uppercase and third being digit.
.	Any single character, including lowercase, uppercase, digit, special symbol like !@#$% etc.
[a-zA-Z]+	One or more alphabet.
.*	Zero or more character. (true for any string).
[a-zA-Z0-9]+	One or more occurrence of alphanumeric characters.
[0-9]*	Zero or more occurrence of a digit.
[a]	Occurrence of 'a' at any position.
[a-zA-Z]{2,4}	2 to 4 occurrences of a character.
[a-zA-Z]{3}	Exactly 3 occurrence of a character.
\. Or \+	Presence of characters such as . and +etc. are preceded with escape sequence.
^[a-zA-Z]	String that starts with a lowercase or uppercase alphabet.
(\.com)$	String that ends with word ".com" at the end.
[\w]	A special character (character except alphabet, underscore and digit).

Q. Define XSD for an element <password> with the following pattern:

- There must be at least 8 characters.
- Must start with alphabet.
- There must be at least 1 lowercase, 1 uppercase and one special symbol.

Ans→

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="password">
    <xs:simpleType>
```

```
      <xs:restriction base="xs:string">
        <xs:pattern value="^([A-Za-z])(.{7})([A-Z])+([\W]+)"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

Order Indicators (All, Choice, Sequence)

In XML Schema, order indicators are used to specify the order in which child elements can appear within a complex type definition. The three primary order indicators are: `all, Choice and Sequence.`

a. <xs:all>

- The <xs:all> indicator specifies that the child elements **can appear in any order** within the parent element.
- All child elements listed within <xs:all> **must occur exactly once or not at all**.
- It is commonly used when we want to specify that a set of child elements are allowed to appear in any order, and each child element must appear only once.

Example:

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element name="age" type="xs:int"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

b. <xs:choice>

- The <xs:choice> indicator specifies that **exactly one of the child elements within it can appear in the parent element**.
- It is used when we want to define a choice between different child elements, and only one of them can occur.

Example

```
<xs:element name="shape">
  <xs:complexType>
    <xs:choice>
      <xs:element name="circle" type="xs:string"/>
      <xs:element name="rectangle" type="xs:string"/>
      <xs:element name="triangle" type="xs:string"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

c. <xs:sequence>

- The <xs:sequence> indicator specifies that child elements must appear in a specific order within the parent element.
- It is commonly used when you want to define a strict sequence of child elements.

Example

```
<xs:element name="address">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="zip" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Occurrence Indicators (Maxoccurs, Minoccurs)

In XML Schema, occurrence indicators (maxOccurs and minOccurs) are used to specify the minimum and maximum number of times an element can appear within a complex type definition. These indicators help define the cardinality or occurrence constraints for elements.

a. minOccurs:

- minOccurs specifies the **minimum number of times an element must appear** within its parent element.
- It can have a value of **0 or a positive integer** (1, 2, 3, etc.).
- If minOccurs is set to 0, it means the element is optional, and it may or may not appear.
- If minOccurs is set to a positive integer, it means the element must appear at least that many times.

Example:

```
<xs:element name="students">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="student" type="xs:string" minOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

b. maxOccurs:

- maxOccurs specifies the maximum number of times an element can appear within its parent element.
- It can have a value of 1, a positive integer (2, 3, etc.), or the special value unbounded, indicating that there is no upper limit.
- If maxOccurs is set to 1, it means the element can appear at most once (it's optional).
- If maxOccurs is set to a positive integer, it means the element can appear at most that many times.

- If maxOccurs is set to unbounded, it means the element can appear an unlimited number of times.

Example:

```
<xs:element name="items">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="item" type="xs:string" maxOccurs="3"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

DTD

DTD stands for Document Type Definition. A DTD defines the structure and the legal elements and attributes of an XML document. The XML Document Type Declaration, commonly known as DTD, is a way to describe XML language accurately. DTDs check vocabulary and validity of the structure of XML documents against grammatical rules of appropriate XML language. An XML document with correct syntax is called "Well Formed". An XML document validated against a DTD is both "Well Formed" and "Valid".

Advantages:

- DTDs provide a formal way to **define the structure of an XML** document.
- DTDs enable **validation of XML documents**. XML parsers can check whether a given XML document conforms to the specified DTD.
- It helps in **interoperability**. That means, when multiple parties agree on a DTD, they can exchange XML data with confidence that it will be interpreted correctly.
- DTDs serve as **documentation for the XML structure**. They provide a clear and formal description of the elements and attributes that can be used.
- DTDs are supported by older **XML systems**, making them a suitable choice when dealing with legacy applications.
- DTDs can be used to define **security policies for XML** data.
- DTDs are **valuable when transforming or migrating data** between different systems.

An XML document can contain a DTD, which can either be embedded within the document itself (known as an internal DTD) or stored in a separate file (an external DTD). The DTD are divided into the following types.

a. **Internal DTD:**

Internal DTD is declared within the XML file. It is enclosed in the <!DOCTYPE> tag. The DOCTYPE tag must be the first tag in the XML file. The DOCTYPE tag contains the name of the DTD and the location of the DTD within the XML file.

It is enclosed in square brackets [] and appears before the document's root element. An internal DTD is defined within the same file as the XML content it describes.

Syntax: If DTD is declared inside XML document itself, it must be wrapped within the DOCTYPE definition with the following syntax.

```
<!DOCTYPE root_element [
      element declarations
]>
```

Further, the element declarations can follow following syntax:

```
<!ELEMENT element_name (content)>
```

Possible values for content may be:

- EMPTY: `<!ELEMENT br EMPTY>`
- CDATA: `<!ELEMENT email (CDATA)>`
- PCDATA: `<!ELEMENT person (#PCDATA)>`
- ANY: `<!ELEMENT body ANY>`
- Not having content but sequence of child:

```
<!ELEMENT book (title, author, price)>
```

Example:

```
<?xml version="1.0"?>
<!DOCTYPE bookstore [
  <!ELEMENT bookstore (book+)>
  <!ELEMENT book (title, author+, price)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>

<bookstore>
  <book>
    <title>Sample Book</title>
    <author>John Doe</author>
    <price>25.00</price>
  </book>
  <!-- More book elements... -->
</bookstore>
```

Example:

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note (to, from, heading, body)>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT from (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body (#PCDATA)>
]>

<note>
  <to>Ramesh</to>
  <from>Suresh</from>
  <heading>Hello</heading>
  <body>How are you?</body>
</note>
```

We use different symbols in DTD as shown in the following table:

Symbol	Meaning	Example	Explanation
+	One or more occurrence of element.	<!ELEMENT body (h1+)>	h1 element can occur one or more times in body element.
*	Zero or more occurrence of element.	<!ELEMENT div (p*)>	div element can have zero or more paragraphs.

b. External DTD

In an external DTD, the DTD declarations are placed in a separate, standalone DTD file and are referenced from the XML document using the DOCTYPE declaration. This allows for the reuse of DTD definitions across multiple XML documents, promoting modularity and consistency.

Private External DTD: In a private external DTD, the DTD file is referenced using a relative or absolute file path within the DOCTYPE declaration. It's called "private" because it's specific to a particular XML document, and the DTD file's location is known only to the creator of the XML document. Private external DTDs are typically used for XML documents that are not intended for widespread distribution or for cases where the DTD is tightly coupled with the XML content.

Example: books.xml

```
<?xml version="1.0"?>
<!DOCTYPE bookstore SYSTEM "books.dtd">

<bookstore>
  <book>
    <title>Sample Book</title>
    <author>John Doe</author>
```

```
    <price>25.00</price>
  </book>
  <!-- More book elements... -->
</bookstore>
```

Books.dtd

```
<!DOCTYPE bookstore[
      <!ELEMENT bookstore (book+)>
      <!ELEMENT book (title, author, price)>
      <!ELEMENT title (#PCDATA)>
      <!ELEMENT author (#PCDATA)>
      <!ELEMENT price (#PCDATA)> // you can write NUMBER data
type.
]>
```

c. Public External DTD:

Public DTD refers to a DTD that is defined separately from the XML document and can be shared and reused across multiple XML documents. Public DTDs are often hosted on public servers and identified by a formal public identifier (PID) or a system identifier (SID). To reference a public external DTD in an XML document, you typically use a <!DOCTYPE> declaration that includes the public identifier or system identifier.
Example:

```
<?xml version="1.0"?>
<!DOCTYPE library PUBLIC "library.dtd">
<library>
  <!-- XML content here -->
</library>
```

> **Note:** SYSTEM is used for DTDs available in the local file system or on a local network. PUBLIC is used for DTDs that are publicly available on the internet and are identified using a formal public identifier and a URL.

Defining Elements and Attributes

a. Defining Element: To define elements in a Document Type Definition (DTD), you use the <!ELEMENT> declaration. The <!ELEMENT> declaration specifies the structure and content model of an element in an XML document. Here's the basic syntax for defining elements in a DTD:

```
<!ELEMENT element_name content_model>
```

- element_name: This is the name of the element you are defining.

- content_model: This specifies what the element's content can contain. There are several content models we can use to define the structure of elements:

Different content modes and description
EMPTY: The element is empty and cannot contain any content.
ANY: The element can contain any content (including elements and text) and is not restricted by the DTD.
(#PCDATA): The element can contain parsed character data (text).
(child_element_name): The element can contain one occurrence of the specified child element.
(child_element_name1, child_element_name2, ...): The element can contain one occurrence of each of the specified child elements in the specified order.
(child_element_name1 | child_element_name2): The element can contain either the first child element or the second child element.

Example:

```
<!ELEMENT book EMPTY> <!-- An empty "book" element -->

<!ELEMENT title (#PCDATA)> <!-- A "title" element containing text -->

<!ELEMENT author (first_name, last_name)>
<!-- An "author" element containing "first_name" and "last_name" child
elements -->

<!ELEMENT person (name | address)>
<!-- A "person" element containing either "name" or "address" as child
elements -->
```

b. Defining Attributes: To define attributes in a Document Type Definition (DTD), we use the <!ATTLIST> declaration. The <!ATTLIST> declaration specifies the attributes that are allowed for a particular element and provides information about each attribute's name, data type, default value, and other characteristics.

Here's the basic syntax for defining attributes in a DTD:

```
<!ATTLIST element_name attr_name attr_data_type default_value>
```

- element_name: This is the name of the element to which the attributes are being defined.
- attribute_name: This is the name of the attribute you are defining for the element.
- attribute_data_type: This specifies the data type of the attribute, indicating the kind of data that is allowed for this attribute. Common attribute data types include CDATA (character data), ID (unique identifier), ENUM (enumerated list of values), and others.
- default_value: This is the default value for the attribute, and it can be one of the following:

  #REQUIRED: The attribute must always be provided for the element.
  #IMPLIED: The attribute is optional and doesn't need to be present.
  "default_value": Specifies a default value for the attribute.

Example:

```
<!ELEMENT book (title, author)>
<!ATTLIST book
      isbn CDATA #REQUIRED
      published CDATA #IMPLIED
>
```

In the above example:

- book is defined as an element containing title and author elements.
- The <!ATTLIST> declaration for the book element specifies two attributes:
- isbn is of type CDATA (character data) and is required attribute.
- published is also of type CDATA, but it is not mandatory (#IMPLIED).

Example:

```
<!DOCTYPE cars [
  <!ELEMENT cars (car+)>
  <!ELEMENT car (make, model, year)>
  <!ATTLIST car
      vin CDATA #REQUIRED
```

```
    color CDATA "Unknown"
  >
  <!ELEMENT make (#PCDATA)>
  <!ELEMENT model (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
]>
```

Note: #PCDATA and CDATA are two different data types used in Document Type Definitions (DTDs) and XML schemas to define how the content of elements should be treated.

- PCDATA parses special characters and requires them to be escaped. We are not supposed to include <, >, &, # etc. as content.
- CDATA treats special characters as literal characters and does not require escaping. We can include <, >, &, # etc. as content.

Following XML document is according to the above DTD.

```
<cars>
 <car vin="ABC123XYZ">
   <make>Toyota</make>
   <model>Camry</model>
   <year>2022</year>
 </car>
 <car vin="DEF456UVW" color="Red">
   <make>Ford</make>
   <model>Mustang</model>
   <year>2023</year>
 </car>
</cars>
```

Q. Consider the following XML file. Write DTD for the given XML.

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

Ans→ **DTD**

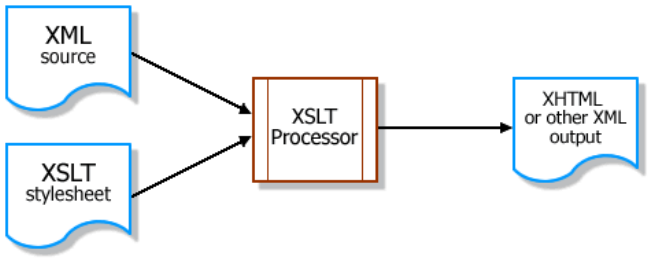
```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

XSLT

XSL (Extensible Stylesheet Language) is a family of languages used for transforming and styling XML documents. The primary members of the XSL family is XSLT.

XSLT stands for Extensible Stylesheet Language Transformations. It is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element.

A common way to describe the transformation process is to say that XSLT transforms an XML source-tree into an XML result-tree.



XSLT uses XPath to find information in an XML document. XPath is used to navigate through elements and attributes in XML documents.

In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

Features:

- It is applicable to transform the XML document from one form to another.
- XPath is integrated for navigating and querying XML documents within XSLT.
- XSLT uses a declarative syntax, specifying what the output should look like rather than how to create it.
- Supports complex transformations, including grouping, sorting, and conditional processing.
- Allows the use of variables and parameters for dynamic transformations.
- XSLT is a W3C standard, ensuring compatibility and consistency.

XPATH

XPath stands for XML Path Language. It uses "path like" syntax to identify and navigate nodes in an XML document. XPath contains over 200 built-in functions and is **a major component in the XSLT** standard. It is a **W3C recommended** technology to be used with XML.

XPath uses path expressions to select nodes or node-sets in an XML document. These path expressions look very much like the path expressions we use with **traditional computer file systems**. Today XPath expressions **can also be used in** JavaScript, Java, XML Schema, PHP, Python, C and C++, and lots of other languages. **XPATH 3.0 is the latest version** of XPATH standard released on 2014.

XML documents are treated as trees of nodes. The XPATH uses seven different types of nodes:

- Root node
- element node
- attribute node
- text node
- namespace node
- processing-instruction node
- comment node

Consider the following XML document:

```
<?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>

  <book>
    <title lang="en">Web Technology</title>
    <author>G. Dhungana</author>
    <year>2015</year>
    <price>500</price>
  </book>

</bookstore>
```

XPath uses path expressions to select nodes or node-sets in an XML document. The most important path expressions as listed below.

Expression	Description
nodename	Selects all nodes with the name "nodename"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

For example, for the above xml document,

Path Expression	Result
bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore Note: If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element.
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
//title | //price	Selects all the title AND price elements in the document

Advantages of Using XPATH:

- XPath provides a concise and easy-to-understand syntax for addressing elements and attributes in XML documents.
- XPath is versatile and can be used in various XML-related technologies.
- XPath is supported by many programming languages, libraries, and tools.
- XPath processors are optimized for performance even when dealing with large datasets.

Overall, XPath is a fundamental tool for working with XML data, providing a wide range of capabilities for selecting, querying, and manipulating XML documents. Its simplicity, flexibility, and broad adoption make it a valuable asset in XML-related development and data processing tasks.

Xquery:

XQuery is a powerful and expressive query language designed for querying and transforming XML (Extensible Markup Language) data. It was developed by the World Wide Web Consortium (W3C) and is part of the family of XML-related technologies.

Features:

- Xquery is a query language designed to query XML data.
- It uses Xpath for navigation.
- It offers a rich set of built-in functions and operators.
- It allows creating and modifying XML documents.
- It is W3C recommendation for broad adoption.
- It is suitable to use for web service interactions.
- It is suitable for large XML datasets.

Example: Consider the following XML document (books.xml).

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

<bookstore>

<book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>

<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>

<book category="WEB">
```

```
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>

<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>

</bookstore>
```

Now, consider the following table showing different Xquery expressions and their results.

Xquery Expressions	Result

doc("books.xml")/bookstore/book/title	`<title lang="en">Everyday Italian</title>`
	`<title lang="en">Harry Potter</title>`
	`<title lang="en">XQuery Kick Start</title>`
	`<title lang="en">Learning XML</title>`

doc("books.xml")/bookstore/book[price<30]	`<book category="CHILDREN">`
	`  <title lang="en">Harry Potter</title>`
	`  <author>J K. Rowling</author>`
	`  <year>2005</year>`
	`  <price>29.99</price>`
	`</book>`

doc("books.xml")/bookstore/book[price<30]/title	`<title lang="en">Harry Potter</title>`

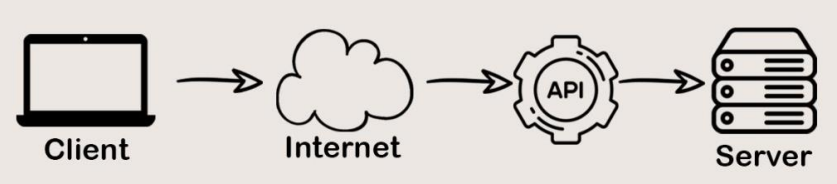
SAX:

SAX stands for "Simple API for XML" or "SAX API." It is a widely used event-driven, and serial access API for parsing and processing XML documents. SAX is an efficient and lightweight way to process XML data

because it doesn't require loading the entire XML document into memory, making it suitable for working with large XML files or streams.

Developers can use SAX to read XML documents sequentially, handling elements, attributes, and content as they are encountered during parsing.

SAX (Simple API for XML) is typically used in programming languages like Java to parse and process XML documents. Not only java, most of the other popular programming languages like python, php etc. also provide different classes and functions to work with SAX.



Features:

- It is use to parse XML document efficiently.
- Since, SAX doesn't require loading the entire XML document into memory. With SAX, we can efficiently parse and process large XML files and streams.
- We can extract specific information or data elements from XML documents.
- We can use SAX to perform XML validation, including well-formedness checks and schema validation using XML Schema Definition (XSD).
- It allows selectively parse specific part of a XML document.

DOM:

DOM is an acronym stands for Document Object Model. It defines a standard way to access and manipulate documents. The Document Object Model (DOM) is a programming API for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

As a W3C specification, one important objective for the Document Object Model is to provide a standard programming interface that can be used in a wide variety of environments and applications. The Document Object Model can be used with many programming language.

XML DOM defines a standard way to access and manipulate XML documents.

Features:

- is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content.
- We can access all elements through the DOM tree.

- The HTML DOM defines a standard way for accessing and manipulating HTML documents by javascript.
- It presents an HTML document as a tree-structure.
- We can modify or delete their content and also create new elements.

Consider a XML document as follows:

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

  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>

  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>

</bookstore>
```

Following JavaScript code can be used to access the above document to get the value of the first title elemnt.

```
txt = xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
```

Creating XML Parser:

An XML parser is a software component or library that is used to process and interpret XML (Extensible Markup Language) documents. XML parsers are responsible for reading XML data, validating its structure, and making it available for further processing by applications or systems. XML parsers play a crucial role in working with XML data, as they ensure that XML documents are correctly understood and processed by software.

Creating an XML parser typically involves using a programming language and a library or module that provides XML parsing capabilities.

Here is an example of creating XML parser in PHP.

```php
<?php
$xml_data = '
<bookstore>
  <book>
    <title lang="en">Himalayan Ninja</title>
    <author>Ramesh Gopal Varma</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <!-- Add more XML content here -->
</bookstore>
';

// Create a SimpleXML object by loading the XML data

$xml = simplexml_load_string($xml_data);

// Check if the parsing was successful
if ($xml === false) {
    echo "Failed to parse XML.\n";
    exit;
}

// Access and process XML elements
foreach ($xml->book as $book) {
     $title = $book->title;
     $author = $book->author;
     $year = $book->year;
     $price = $book->price;

     echo "Title: $title\n";
     echo "Author: $author\n";
     echo "Year: $year\n";
     echo "Price: $price\n";
     echo "----------------------\n";
}
?>
```

Output:

```
Title: Himalayan Ninja
Author: Ramesh Gopal Varma
Year: 2005
Price: 30.00
----------------------
```

XML parsers are essential components in various software applications and systems, including web services, data interchange formats, configuration files, content management systems, and more. They

facilitate the interoperable exchange of structured data in a standardized and well-defined format, making XML a fundamental technology for data representation and exchange in many domains.