eXtensible Markup Language (XML)

What is XML?

- XML stands for eXtensible Markup Language
- XML is a markup language much like HTML
- XML was designed to store and transport data
- XML was designed to be self-descriptive
- XML is a W3C Recommendation

XML Does Not DO Anything

Maybe it is a little hard to understand, but XML does not DO anything.

This note is a note to Tove from Jani, stored as XML:

```
<note>
  <to>Mahi</to>
  <from>Diya</from>
  <heading>Reminder</heading>
  <body>Don't forget to send email!</body>
</note>
```

The XML above is quite self-descriptive:

- It has sender information
- It has receiver information
- It has a heading
- It has a message body

But still, the XML above does not DO anything. XML is just information wrapped in tags.

The Difference Between XML and HTML

XML and HTML were designed with different goals:

- XML was designed to carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are

XML Simplifies Things

- XML simplifies data sharing
- XML simplifies data transport
- XML simplifies platform changes
- XML simplifies data availability

Many computer systems contain data in incompatible formats. Exchanging data between incompatible systems (or upgraded systems) is a time-consuming task for web developers. Large amounts of data must be converted, and incompatible data is often lost.

XML stores data in plain text format. This provides a software- and hardware-independent way of storing, transporting, and sharing data.

XML also makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

With XML, data can be available to all kinds of "reading machines" like people, computers, voice machines, news feeds, etc.

XML Separates Data from HTML

When displaying data in HTML, you should not have to edit the HTML file when the data changes.

With XML, the data can be stored in separate XML files.

With a few lines of JavaScript code, you can read an XML file and update the data content of any HTML page.

```
<?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" cover="paperback">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>

</bookstore>
```

Transaction Data

Thousands of XML formats exist, in many different industries, to describe day-to-day data transactions:

- Stocks and Shares
- Financial transactions
- Medical data
- Mathematical data
- Scientific measurements
- News information
- Weather services

Example: XML News

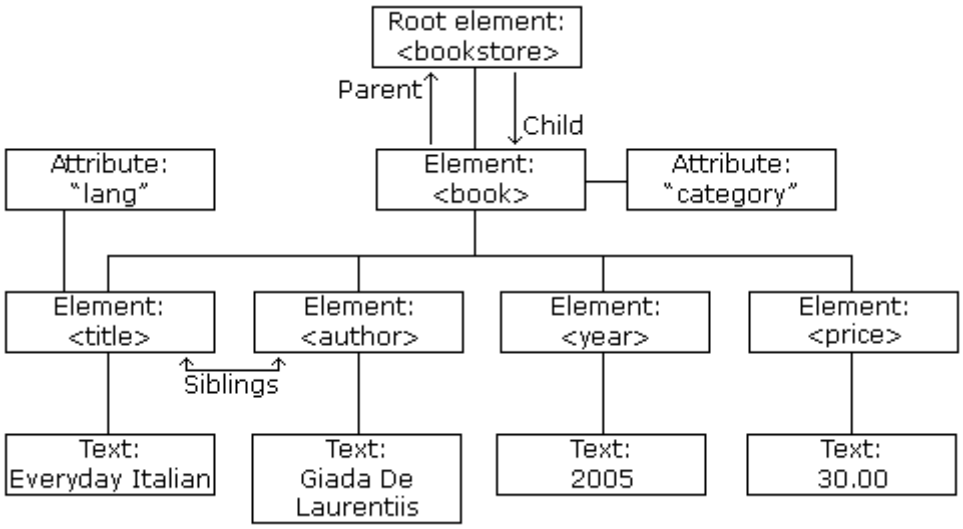
```
<?xml version="1.0" encoding="UTF-8"?>
<nitf>
  <head>
    <title>Colombia Earthquake</title>
  </head>
  <body>
    <headline>
      <hl1>143 Dead in Colombia Earthquake</hl1>
    </headline>
    <byline>
      <bytag>By Jared Kotler, Associated Press Writer</bytag>
    </byline>
    <dateline>
```

```
      <location>Bogota, Colombia</location>
      <date>Monday January 25 1999 7:28 ET</date>
    </dateline>
  </body>
</nitf>
```

Example: XML Weather Service

```
<?xml version="1.0" encoding="UTF-8"?>
<current_observation>
<credit>NOAA's National Weather Service</credit>
<credit_URL>http://weather.gov/</credit_URL>

<image>
  <url>http://weather.gov/images/xml_logo.gif</url>
  <title>NOAA's National Weather Service</title>
  <link>http://weather.gov</link>
</image>

<location>New York/John F. Kennedy Intl Airport, NY</location>
<station_id>KJFK</station_id>
<latitude>40.66</latitude>
<longitude>-73.78</longitude>
<observation_time_rfc822>Mon, 11 Feb 2008 06:51:00 -0500 EST
</observation_time_rfc822>

<weather>A Few Clouds</weather>
<temp_f>11</temp_f>
<temp_c>-12</temp_c>
<relative_humidity>36</relative_humidity>
<wind_dir>West</wind_dir>
<wind_degrees>280</wind_degrees>
<wind_mph>18.4</wind_mph>
<wind_gust_mph>29</wind_gust_mph>
<pressure_mb>1023.6</pressure_mb>
<pressure_in>30.23</pressure_in>
<dewpoint_f>-11</dewpoint_f>
<dewpoint_c>-24</dewpoint_c>
<windchill_f>-7</windchill_f>
<windchill_c>-22</windchill_c>
<visibility_mi>10.00</visibility_mi>
<icon_url_base>http://weather.gov/weather/images/fcicons/</icon_url_bas
e>
<icon_url_name>nfew.jpg</icon_url_name>
<disclaimer_url>http://weather.gov/disclaimer.html</disclaimer_url>
<copyright_url>http://weather.gov/disclaimer.html</copyright_url>

</current_observation>
```

The XML Tree Structure



An Example XML Document

The image above represents books in this XML:

```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">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

XML Elements

An XML document contains XML Elements.

What is an XML Element?

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

```
<price>29.99</price>
```

An element can contain:

- text
- attributes
- other elements
- or a mix of the above

```
<bookstore>
  <book category="children">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

In the example above:

<title>, <author>, <year>, and <price> have **text content** because they contain text (like 29.99).

<bookstore> and <book> have **element contents**, because they contain elements.

<book> has an **attribute** (category="children").

Empty XML Elements

An element with no content is said to be empty.

In XML, you can indicate an empty element like this:

```
<element></element>
```

You can also use a so called self-closing tag:

```
<element />
```

The two forms produce identical results in XML software (Readers, Parsers, Browsers).

XML Naming Rules

XML elements must follow these naming rules:

- 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

Any name can be used, no words are reserved (except xml).

Best Naming Practices

Create descriptive names, like this: <person>, <firstname>, <lastname>.

Create short and simple names, like this: <book_title> not like this: <the_title_of_the_book>.

Avoid "-". If you name something "first-name", some software may think you want to subtract "name" from "first".

Avoid ".". If you name something "first.name", some software may think that "name" is a property of the object "first".

Avoid ":". Colons are reserved for namespaces (more later).

Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software doesn't support them!

Naming Conventions

Some commonly used naming conventions for XML elements:

Style	Example	Description
Lower case	<firstname>	All letters lower case
Upper case	<FIRSTNAME>	All letters upper case
Snake case	<first_name>	Underscore separates words (commonly used in SQL databases)
Pascal case	<FirstName>	Uppercase first letter in each word (commonly used by C programmers)
Camel case	<firstName>	Uppercase first letter in each word except the first (commonly used in JavaScript)

XML Attributes

XML elements can have attributes, just like HTML.

Attributes are designed to contain data related to a specific element.

XML Attributes Must be Quoted

Attribute values must always be quoted. Either single or double quotes can be used.

For a person's gender, the <person> element can be written like this:

```
<person gender="female">
```

or like this:

```
<person gender='female'>
```

If the attribute value itself contains double quotes you can use single quotes, like in this example:

```
<gangster name='George "Shotgun" Ziegler'>
```

or you can use character entities:

```
<gangster name="George &quot;Shotgun&quot; Ziegler">
```

XML Elements vs. Attributes

Take a look at these two examples:

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

In the first example, gender is an attribute. In the last example, gender is an element. Both examples provide the same information.

There are no rules about when to use attributes or when to use elements in XML.

XML DTD

An XML document with correct syntax is called "Well Formed".

An XML document validated against a DTD is both "Well Formed" and "Valid".

What is a 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:

```
<?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>
```

The DOCTYPE declaration above contains a reference to a DTD file. The content of the DTD file is shown and explained below.

XML DTD

The purpose of a DTD is to define the structure and the legal elements and attributes of an XML document:

Note.dtd:

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

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"

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

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

XML Schema is an XML-based alternative to DTD:

```
<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>
```

The Schema above is interpreted like this:

- <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

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

XQuery

What is XQuery?

- XQuery is to XML what SQL is to databases.
- XQuery is designed to query XML data.

XQuery Example

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

What is FLWOR?

FLWOR (pronounced "flower") is an acronym for "For, Let, Where, Order by, Return".

- **For** - selects a sequence of nodes
- **Let** - binds a sequence to a variable
- **Where** - filters the nodes
- **Order by** - sorts the nodes
- **Return** - what to return (gets evaluated once for every node)

The XML Example Document

We will use the "books.xml" document in the examples below (same XML file as in the previous chapter).

How to Select Nodes From "books.xml" With FLWOR

Look at the following path expression:

```
doc("books.xml")/bookstore/book[price>30]/title
```

The expression above will select all the title elements under the book elements that are under the bookstore element that have a price element with a value that is higher than 30.

The following FLWOR expression will select exactly the same as the path expression above:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
return $x/title
```

The result will be:

```
<title lang="en">XQuery Kick Start</title>
<title lang="en">Learning XML</title>
```

With FLWOR you can sort the result:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

The **for** clause selects all book elements under the bookstore element into a variable called $x.

The **where** clause selects only book elements with a price element with a value greater than 30.

The **order by** clause defines the sort-order. Will be sort by the title element.

The **return** clause specifies what should be returned. Here it returns the title elements.

The result of the XQuery expression above will be:

```
<title lang="en">Learning XML</title>
<title lang="en">XQuery Kick Start</title>
```

Present the Result In an HTML List

Look at the following XQuery FLWOR expression:

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

The expression above will select all the title elements under the book elements that are under the bookstore element, and return the title elements in alphabetical order.

Now we want to list all the book-titles in our bookstore in an HTML list. We add <ul> and <li> tags to the FLWOR expression:

```
<ul>
{
for $x in doc("books.xml")/bookstore/book/title
order by $x
return <li>{$x}</li>
}
</ul>
```

The result of the above will be:

```
<ul>
<li><title lang="en">Everyday Italian</title></li>
<li><title lang="en">Harry Potter</title></li>
<li><title lang="en">Learning XML</title></li>
<li><title lang="en">XQuery Kick Start</title></li>
</ul>
```

Now we want to eliminate the title element, and show only the data inside the title element:

```
<ul>
{
for $x in doc("books.xml")/bookstore/book/title
order by $x
return <li>{data($x)}</li>
}
</ul>
```

The result will be (an HTML list):

```
<ul>
<li>Everyday Italian</li>
<li>Harry Potter</li>
<li>Learning XML</li>
<li>XQuery Kick Start</li>
</ul>
```

XQuery Selecting and Filtering

The XML Example Document

We will use the "books.xml" document in the examples below (same XML file as in the previous chapters).

Selecting and Filtering Elements

As we have seen in the previous chapters, we are selecting and filtering elements with either a Path expression or with a FLWOR expression.

Look at the following FLWOR expression:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

- for - (optional) binds a variable to each item returned by the in expression
- let - (optional)
- where - (optional) specifies a criteria
- order by - (optional) specifies the sort-order of the result
- return - specifies what to return in the result

The for Clause

The for clause binds a variable to each item returned by the in expression. The for clause results in iteration. There can be multiple for clauses in the same FLWOR expression.

To loop a specific number of times in a for clause, you may use the **to** keyword:

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

Result:

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

The **at** keyword can be used to count the iteration:

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

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 in expression in the for clause. Use comma to separate each in expression:

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

Result:

```
<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>
```

The let Clause

The let clause allows variable assignments and it avoids repeating the same expression many times. The let clause does not result in iteration.

```
let $x := (1 to 5)
return <test>{$x}</test>
```

Result:

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

The where Clause

The where clause is used to specify one or more criteria for the result:

```
where $x/price>30 and $x/price<100
```

The order by Clause

The order by clause is used to specify the sort order of the result. Here we want to order the result by category and title:

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

Result:

```
<title lang="en">Harry Potter</title>
<title lang="en">Everyday Italian</title>
<title lang="en">Learning XML</title>
<title lang="en">XQuery Kick Start</title>
```

The return Clause

The return clause specifies what is to be returned.

```
for $x in doc("books.xml")/bookstore/book
return $x/title
```

Result:

```
<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>
```

Working with XML Data in SQL Server

We will see now how we can work with XML in SQL Server. We will see how to convert tables in SQL into XML, how to load XML documents into SQL Server and how to create SQL tables from XML documents.

Let's first generate some dummy data. We will use this data to create XML documents. Execute the following script:

```
CREATE DATABASE Showroom
 Use Showroom
CREATE TABLE Car
(
  CarId int identity(1,1) primary key,
  Name varchar(100),
  Make varchar(100),
  Model int ,
  Price int ,
  Type varchar(20)
)
insert into Car( Name, Make,  Model , Price, Type)
VALUES ('Corrolla','Toyota',2015, 20000,'Sedan'),
('Civic','Honda',2018, 25000,'Sedan'),
('Passo','Toyota',2012, 18000,'Hatchback'),
('Land Cruiser','Toyota',2017, 40000,'SUV'),
('Corrolla','Toyota',2011, 17000,'Sedan'),
('Vitz','Toyota',2014, 15000,'Hatchback'),
('Accord','Honda',2018, 28000,'Sedan'),
('7500','BMW',2015, 50000,'Sedan'),
('Parado','Toyota',2011, 25000,'SUV'),
('C200','Mercedez',2010, 26000,'Sedan'),
('Corrolla','Toyota',2014, 19000,'Sedan'),
('Civic','Honda',2015, 20000,'Sedan')
```

In the script above, we created a Showroom database with one table Car. The Car table has five attributes: CarId, Name, Make, Model, Price, and Type. Next, we added 12 dummy records in the Car table.

Converting into XML from SQL tables

The simplest way to convert data from SQL tables into XML format is to use the FOR XML AUTO and FOR XML PATH clauses.

FOR XML AUTO in SQL SERVER

The FOR XML AUTO clause converts each column in the SQL table into an attribute in the corresponding XML document.

Execute the following script:

```
USE Showroom
SELECT * FROM Car
FOR XML AUTO
```

In the console output you will see the following:

Click on the link and you will see the following document in a new query window of SQL Server management studio:

```
<Car CarId="1" Name="Corrolla" Make="Toyota" Model="2015" Price="20000" Type="Sedan" />
<Car CarId="2" Name="Civic" Make="Honda" Model="2018" Price="25000" Type="Sedan" />
<Car CarId="3" Name="Passo" Make="Toyota" Model="2012" Price="18000" Type="Hatchback" />
<Car CarId="4" Name="Land Cruiser" Make="Toyota" Model="2017" Price="40000" Type="SUV" />
<Car CarId="5" Name="Corrolla" Make="Toyota" Model="2011" Price="17000" Type="Sedan" />
<Car CarId="6" Name="Vitz" Make="Toyota" Model="2014" Price="15000" Type="Hatchback" />
<Car CarId="7" Name="Accord" Make="Honda" Model="2018" Price="28000" Type="Sedan" />
<Car CarId="8" Name="7500" Make="BMW" Model="2015" Price="50000" Type="Sedan" />
<Car CarId="9" Name="Parado" Make="Toyota" Model="2011" Price="25000" Type="SUV" />
<Car CarId="10" Name="C200" Make="Mercedez" Model="2010" Price="26000" Type="Sedan" />
<Car CarId="11" Name="Corrolla" Make="Toyota" Model="2014" Price="19000" Type="Sedan" />
<Car CarId="12" Name="Civic" Make="Honda" Model="2015" Price="20000" Type="Sedan" />
```

You can see that for each record an element Car has been created in the XML document, and for each column, an attribute with the same name has been added to each element in the XML document.

FOR XML PATH in SQL SERVER

The FOR XML AUTO class creates an XML document where each column is an attribute. On the other hand, the FOR XML PATH will create an XML document where each record is an element and each column is a nested element for a particular record. Let's see this in action:

```
USE Showroom

SELECT * FROM Car

FOR XML PATH
```

A snapshot of the output is as follows:

```
<row>
  <CarId>1</CarId>
  <Name>Corrolla</Name>
  <Make>Toyota</Make>
  <Model>2015</Model>
  <Price>20000</Price>
  <Type>Sedan</Type>
</row>
<row>
  <CarId>2</CarId>
  <Name>Civic</Name>
  <Make>Honda</Make>
  <Model>2018</Model>
  <Price>25000</Price>
  <Type>Sedan</Type>
</row>
<row>
  <CarId>3</CarId>
  <Name>Passo</Name>
  <Make>Toyota</Make>
  <Model>2012</Model>
  <Price>18000</Price>
  <Type>Hatchback</Type>
</row>
<row>
  <CarId>4</CarId>
  <Name>Land Cruiser</Name>
  <Make>Toyota</Make>
  <Model>2017</Model>
  <Price>40000</Price>
  <Type>SUV</Type>
</row>
```

In the output, you will see a total of 12 elements (the screenshot shows only the first 4). You can see that each column name has been converted to an element. However, there is one problem; by default, the parent element name is “row”. We can change that using the following query:

```
USE Showroom
SELECT * FROM Car
FOR XML PATH ('Car')
```

```
<Car>
  <CarId>1</CarId>
  <Name>Corrolla</Name>
  <Make>Toyota</Make>
  <Model>2015</Model>
  <Price>20000</Price>
  <Type>Sedan</Type>
</Car>
<Car>
  <CarId>2</CarId>
  <Name>Civic</Name>
  <Make>Honda</Make>
  <Model>2018</Model>
  <Price>25000</Price>
  <Type>Sedan</Type>
</Car>
<Car>
  <CarId>3</CarId>
  <Name>Passo</Name>
  <Make>Toyota</Make>
  <Model>2012</Model>
  <Price>18000</Price>
  <Type>Hatchback</Type>
</Car>
<Car>
  <CarId>4</CarId>
  <Name>Land Cruiser</Name>
  <Make>Toyota</Make>
  <Model>2017</Model>
  <Price>40000</Price>
  <Type>SUV</Type>
</Car>
```

In the output, you can see Car as the parent element for each sub-element. However, the document is not well-formed as there is no root element in the document. To add a root element, we need to execute the following script:

```
USE Showroom
SELECT * FROM Car
FOR XML PATH ('Car'), ROOT('Cars')
```

In the output, you should see “Cars” as the root element as shown below:

```
<Cars>
  <Car>
    <CarId>1</CarId>
    <Name>Corrolla</Name>
    <Make>Toyota</Make>
    <Model>2015</Model>
    <Price>20000</Price>
    <Type>Sedan</Type>
  </Car>
  <Car>
    <CarId>2</CarId>
    <Name>Civic</Name>
    <Make>Honda</Make>
    <Model>2018</Model>
    <Price>25000</Price>
    <Type>Sedan</Type>
  </Car>
  <Car>
    <CarId>3</CarId>
    <Name>Passo</Name>
    <Make>Toyota</Make>
    <Model>2012</Model>
    <Price>18000</Price>
    <Type>Hatchback</Type>
  </Car>
  <Car>
    <CarId>4</CarId>
    <Name>Land Cruiser</Name>
    <Make>Toyota</Make>
    <Model>2017</Model>
    <Price>40000</Price>
    <Type>SUV</Type>
  </Car>
```

Now suppose you want that CarId should be the attribute of the Car element rather than an element. You can do so with the following script:

```
USE Showroom
SELECT  CarId as [@CarID],
   Name  AS [CarInfo/Name],
   Make [CarInfo/Make],
   Model [CarInfo/Model],
   Price,
   Type
FROM Car
FOR XML PATH ('Car'), ROOT('Cars')
```

The output looks like this:

```
<Cars>
  <Car CarID="1">
    <CarInfo>
      <Name>Corrolla</Name>
      <Make>Toyota</Make>
      <Model>2015</Model>
    </CarInfo>
    <Price>20000</Price>
    <Type>Sedan</Type>
  </Car>
  <Car CarID="2">
    <CarInfo>
      <Name>Civic</Name>
      <Make>Honda</Make>
      <Model>2018</Model>
    </CarInfo>
    <Price>25000</Price>
    <Type>Sedan</Type>
  </Car>
  <Car CarID="3">
    <CarInfo>
      <Name>Passo</Name>
      <Make>Toyota</Make>
      <Model>2012</Model>
    </CarInfo>
    <Price>18000</Price>
    <Type>Hatchback</Type>
  </Car>
  <Car CarID="4">
    <CarInfo>
      <Name>Land Cruiser</Name
      <Make>Toyota</Make>
      <Model>2017</Model>
    </CarInfo>
    <Price>40000</Price>
    <Type>SUV</Type>
  </Car>
```

You can see now that CarId has become an attribute of the Car element.

We can add further nesting levels to an XML document. For instance, if we want Name, Make and Model elements to be nested inside another element CarInfo we can do so with the following script:

```
USE Showroom
SELECT  CarId as [@CarID],
   Name  AS [CarInfo/Name],
   Make [CarInfo/Make],
   Model [CarInfo/Model],
   Price,
   Type
FROM Car
FOR XML PATH ('Car'), ROOT('Cars')
```

In the output, you will see a new element CarInfo that encloses the Name, Make and Model elements as shown below:

```
<Cars>
  <Car CarID="1">
    <CarInfo>
      <Name>Corrolla</Name>
      <Make>Toyota</Make>
      <Model>2015</Model>
    </CarInfo>
    <Price>20000</Price>
    <Type>Sedan</Type>
  </Car>
  <Car CarID="2">
    <CarInfo>
      <Name>Civic</Name>
      <Make>Honda</Make>
      <Model>2018</Model>
    </CarInfo>
    <Price>25000</Price>
    <Type>Sedan</Type>
  </Car>
  <Car CarID="3">
    <CarInfo>
      <Name>Passo</Name>
      <Make>Toyota</Make>
      <Model>2012</Model>
    </CarInfo>
    <Price>18000</Price>
    <Type>Hatchback</Type>
  </Car>
  <Car CarID="4">
    <CarInfo>
      <Name>Land Cruiser</Name>
      <Make>Toyota</Make>
      <Model>2017</Model>
    </CarInfo>
    <Price>40000</Price>
    <Type>SUV</Type>
  </Car>
```

Finally, if you want to convert the elements Name and Make into an attribute of element CarInfo, you can do so with the following script:

```
USE Showroom
SELECT  CarId as [@CarID],
   Name  AS [CarInfo/@Name],
   Make [CarInfo/@Make],
   Model [CarInfo/Model],
   Price,
   Type
FROM Car
FOR XML PATH ('Car'), ROOT('Cars')
```

The output looks like this:

```
<Cars>
  <Car CarID="1">
    <CarInfo Name="Corrolla" Make="Toyota">
      <Model>2015</Model>
    </CarInfo>
    <Price>20000</Price>
    <Type>Sedan</Type>
  </Car>
  <Car CarID="2">
    <CarInfo Name="Civic" Make="Honda">
      <Model>2018</Model>
    </CarInfo>
    <Price>25000</Price>
    <Type>Sedan</Type>
  </Car>
  <Car CarID="3">
    <CarInfo Name="Passo" Make="Toyota">
      <Model>2012</Model>
    </CarInfo>
    <Price>18000</Price>
    <Type>Hatchback</Type>
  </Car>
  <Car CarID="4">
    <CarInfo Name="Land Cruiser" Make="Toyota">
      <Model>2017</Model>
    </CarInfo>
    <Price>40000</Price>
    <Type>SUV</Type>
  </Car>
```

Save the above XML document with the name Cars.xml. In the next section, we will load this XML script into the SQL Server and will see how to create a table from the XML Document.

Creating a SQL table from an XML document

In the previous section, we saw how to create an XML document from the SQL table. In this section, we will see how to do the reverse i.e. we will create a table in SQL using XML documents.

The document we will use is the document that we created in the last section. One node of the document looks like this:

```
<Cars>
  <Car CarID="1">
    <CarInfo Name="Corrolla" Make="Toyota">
      <Model>2015</Model>
    </CarInfo>
    <Price>20000</Price>
    <Type>Sedan</Type>
  </Car>
```

Creating SQL table using XML attributes

Let's first see how we can create an SQL table using attributes. Suppose we want to create a table with two columns that contain the values from the Name and Make attributes from the CarInfo element. We can do so using the following script:

```
DECLARE @cars xml
SELECT @cars = C
FROM OPENROWSET (BULK 'D:\Cars.xml', SINGLE_BLOB) AS Cars(C)
SELECT @cars
DECLARE @hdoc int
EXEC sp_xml_preparedocument @hdoc OUTPUT, @cars
SELECT *
FROM OPENXML (@hdoc, '/Cars/Car/CarInfo' , 1)
WITH(
  Name VARCHAR(100),
  Make VARCHAR(100)
  )
EXEC sp_xml_removedocument @hdoc
```

In the script above we declare an XML type variable @cars. The variable stores the result returned by the OPENROWSET function which retrieves XML data in binary format. Next using the SELECT @Cars statement we print the contents of the XML file. At this point in time, the XML document is loaded into the memory.

Next, we create a handle for the XML document. To read the attributes and elements of the XML document, we need to attach the handle with the XML document. The sp_xml_preparedocument performs this task. It takes the handle and the document variable as parameters and creates an association between them.

Next, we use the OPENXML function to read the contents of the XML document. The OPENXML function takes three parameters: the handle to the XML document, the path of the node for which we want to retrieve the attributes or elements and the mode. The mode value of 1 returns the attributes only. Next, inside the WITH clause, we need to define the name and type of the attributes that you want returned. In our case the CarInfo element has two attributes Name, and Make, therefore we retrieve both.

As a final step, we execute the sp_xml_removedocument stored procedure to remove the XML document from the memory. In the output you will see values from the Name and Make attributes of the CarInfo element as shown below:

Results Messages

	(No column name)
1	<Cars><Car CarID="1"><CarInfo Name="Corrolla" Ma...

	Name	Make
1	Corrolla	Toyota
2	Civic	Honda
3	Passo	Toyota
4	Land Cruiser	Toyota
5	Corrolla	Toyota
6	Vitz	Toyota
7	Accord	Honda
8	7500	BMW
9	Parado	Toyota
10	C200	Merc...
11	Corrolla	Toyota
12	Civic	Honda

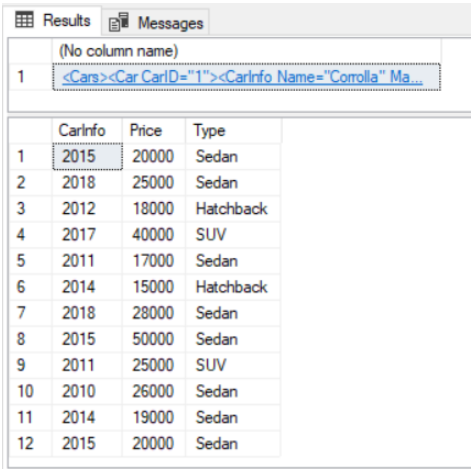
Creating a SQL table using XML elements

To create a SQL table using XML elements, all you have to do is to change the mode value of the OPENXML function to 2 and change the name of the attributes to the name of the element you want to retrieve.

Suppose we want to retrieve the values for the nested CarInfo, Price and Type elements of the parent Car element, we can use the following script:

```
DECLARE @cars xml
 SELECT @cars = C
FROM OPENROWSET (BULK 'D:\Cars.xml', SINGLE_BLOB) AS Cars(C)
 SELECT @cars
 DECLARE @hdoc int
 EXEC sp_xml_preparedocument @hdoc OUTPUT, @cars
SELECT *
FROM OPENXML (@hdoc, '/Cars/Car' , 2)
WITH(
   CarInfo INT,
   Price INT,
   Type VARCHAR(100)
   )
EXEC sp_xml_removedocument @hdoc
```

Output of the script above looks like this:



Results | Messages

	(No column name)
1	<Cars><Car CarID="1"><CarInfo Name="Corrolla" Ma...

	CarInfo	Price	Type
1	2015	20000	Sedan
2	2018	25000	Sedan
3	2012	18000	Hatchback
4	2017	40000	SUV
5	2011	17000	Sedan
6	2014	15000	Hatchback
7	2018	28000	Sedan
8	2015	50000	Sedan
9	2011	25000	SUV
10	2010	26000	Sedan
11	2014	19000	Sedan
12	2015	20000	Sedan