Data Processing with XML and JSON

Data Processing with XML and JSON

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2023

Onlinevarsity

Preface

In today's interconnected and data-driven world, the ability to efficiently handle data is of paramount importance. eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) have become the backbone of data exchange in various domains, from Web development APIs to data storage and configuration files. Understanding how to work with these formats is crucial for developers, data engineers, and anyone dealing with data in digital form.

In this Learner's Guide, we aim to interpret the world of data interchange and processing through XML and JSON.

This book helps you to discover how to use XML, syntax, attributes, and schema. It further explains about DTDs. It also covers XSLT and XPath expressions.

It further describes JSON and data types of JSON. The book shows how to create and parse JSON messages with JavaScript. It introduces advanced JSON features such as Web APIs, JSON HTTP and Files, and Data Storage using JSON.

You will also learn about Cross-Origin Resource Sharing (CORS). This book briefly describes different CORS origins and types. It also explains the kind of CORS requests sent by a browser.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

MANY COURSES ONE PLATFORM

Onlinevarsity

Onlinevarsity App

for Android devices

Download from Google Play Store

Table of Contents

Sessions

Session 1

XML and XML Validations

This session explores the fundamentals of XML and its features. The rudimentary building elements of XML code, schema, and validations are also explained.

Objectives

- Define XML and explain its features
- Illustrate XML Attributes, Comments, Trees, and Validations
- Explain XML DTD with its types and describe working with DTDs

1.1 XML and XML Features

EXtensible Markup Language (XML) is a markup language widely used for storing and transporting data. It is an easy and flexible format that both humans and computers can understand. XML uses tags to represent elements and attributes to provide additional details about those elements. XML offers a uniform representation and exchange of data.

XML is commonly used and supported by diverse programming languages. These languages support parsing, generating, manipulating, and working with XML data.

Some of the programming languages, frameworks, and scripting languages commonly used with XML include Java, C#, Python, JavaScript, PHP, .NET Framework, Ruby, Perl, HTML, and CSS.

Figure 1.1 depicts a typical XML document structure.

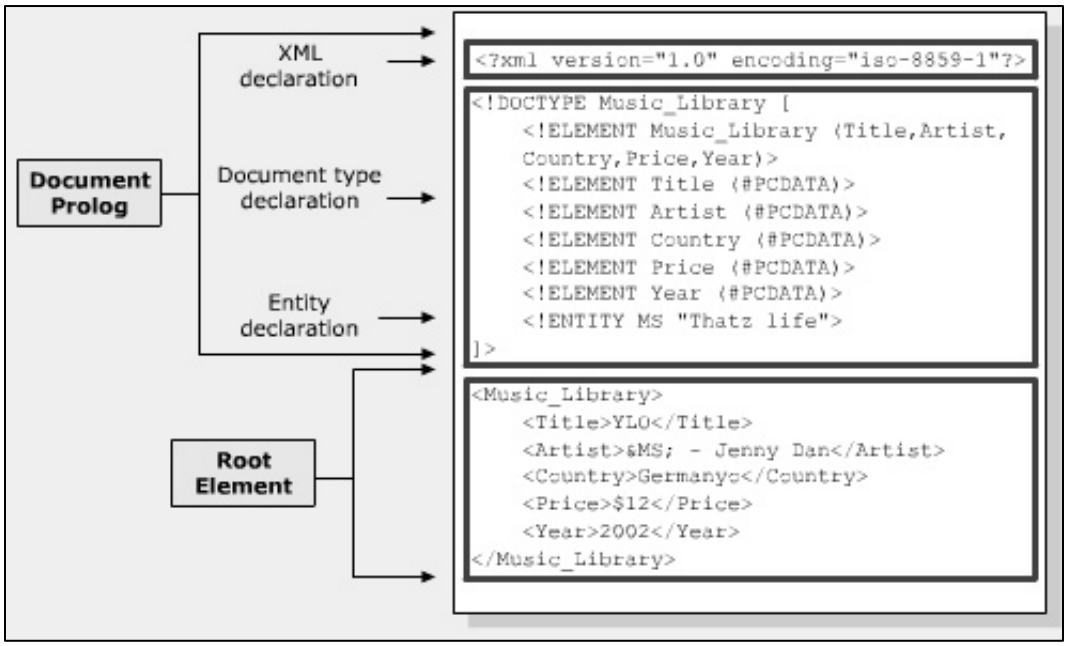


Figure 1.1: XML Document Structure

XML parser gets information about the content in the document with the help of document prolog. Document prolog contains metadata and consists of two parts - XML

Declaration and Document Type Declaration. XML Declaration specifies the version of XML being used. Document Type Declaration defines entities' or attributes' values and checks grammar and vocabulary of markup.

The root element is also called a document element. It must contain all the other elements and content in the document. An XML element has a start tag and end tag.

Data is represented in a systematic form by organizing it into hierarchical order of elements. Programmers can customize the document structure, tags, and characteristics. The systematic form of data is very adaptable to many applications and flexible. XML data can be stored in databases and various apps and applications also support data exchange in XML format.

The hierarchical order of elements can contain text, child elements, or attributes to depict different aspects of the data.

Here are the main ways data is symbolized in XML:

Elements: They are building blocks of XML and represent particular pieces of data. They are enclosed within opening and closing tags. These tags can be custom tags, unlike HTML where tags are standard. For instance:

```
<name>John Mitten</name>
```

In this code, the element <name> denotes the data 'John Mitten'.

Text Content: Elements can contain text content, which describes the actual data values. For example:

```
<age>27</age>
```

In this code, the element <age> contains the text content '27' as the data value.

Attributes: Elements can have attributes that provide further information about the element. Attributes are determined between the opening tag and closing tag and follow the structure of name-value pairs. For example:

```
<book category="fantasy">Alice adventure in wonderland</book>
```

In this code, the element <book> has an attribute called category with the value 'fantasy'. The text content 'Alice adventure in wonderland' depicts the book's title.

Nesting: XML allows for the nesting of elements within other elements, creating a hierarchical system to represent complex data relationships. For example:

```
<employee>
   <name>John Harry</name>
   <age>27</age>
</employee>
```

In this code, the <employee> element contains nested elements <name> and <age>, representing the person's name and age respectively.

Arrays or Replicated Elements: XML can represent arrays or repeated elements by using numerous occurrences of the same element name. For example:

```
<fruits>
   <fruit>Apple</fruit>
   <fruit>Guava</fruit>
   <fruit>Pomegranate</fruit>
</fruits>
```

In this case, the <fruits> element contains multiple occurrences of the <fruit> element, representing a list of fruits.

The capacity of XML to evaluate data against a designated schema is one of its fundamental characteristics. The structure and content of an XML document are specified by a set of rules known as a **schema**. It ensures that the data is accurate and adheres to a specific standard by validating it against the rules.

```
<xs:schema xmlns:xs="http://www.w7.org/2004/XMLSchema">
 <xs:element name="bookname">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="titleofthebook" type="xs:string"/>
    <xs:element name="authorname" type="xs:string"/>
   </xs:sequence>
   <xs:attribute name="genres" type="xs:string"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Within this sample, the schema defines an element <bookname> with child elements <titleofthebook> and <authorname>. The <titleofthebook> and <authorname> elements are of type xs:string and the <bookname> element has an attribute called genres of type xs:string.

XML schemas play a crucial role in ensuring the structure and integrity of XML documents, enabling data exchange and interoperability between systems.

Code Snippet 1 shows an XML document that contains data about books.

XML files have an extension of .xml.

Code Snippet 1: books.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book category="fiction">
<title>The Great Gatsby</title>
```

```
<author>F. Scott Fitzgerald</author>
<year>1925</year>
<price>10.99</price>
</book>
<book category="non-fiction">
<title>Thinking, Fast and Slow</title>
<author>Daniel Kahneman</author>
<year>2011</year>
<price>15.99</price>
</book>
</bookstore>
```

In this code, books.xml file represents information about a bookstore and has listed details of two books. Each book is defined as an element, while its title, author, publishing year, and price are sub-elements. Specifying whether a book is a fiction or non-fiction using the 'category' attribute is possible.

XML documents can be processed and interpreted by software applications or computer programming language using XML parsing techniques. Different programming languages and libraries provide distinct APIs and tools for XML parsing and interpretation, enabling programmers to perform tasks efficiently with XML data and extract the necessary details according to their requirements.

For instance, the title and author of each book would be extracted by a program, which would then display them on a Website or be stored in a database.

XML code can be created using any text editor or Integrated Development Environment (IDE). A number of IDEs today have rich support for code completion, commenting, auto-indenting, and so on. Some popular IDEs include Visual Studio Code, Notepad++, and Apache NetBeans.

NetBeans IDE

NetBeans is a versatile Java IDE that enables the creation of applications using a collection of modular software components known as modules. It operates seamlessly on Windows, macOS, Linux, and Solaris. NetBeans offers a flexible development platform for Java developers. Since Java applications frequently use XML files, NetBeans supports XML.

Why is NetBeans preferred for XML?

NetBeans supports XML natively, so there are no plugins required to be installed or hidden options to be enabled. NetBeans is free, has rich community support, and has several options for code completion, auto-indenting, and so on.

To create an XML file in NetBeans, following are the steps:

1. Open NetBeans.
2. Click **File → New File** as shown in Figure 1.2.

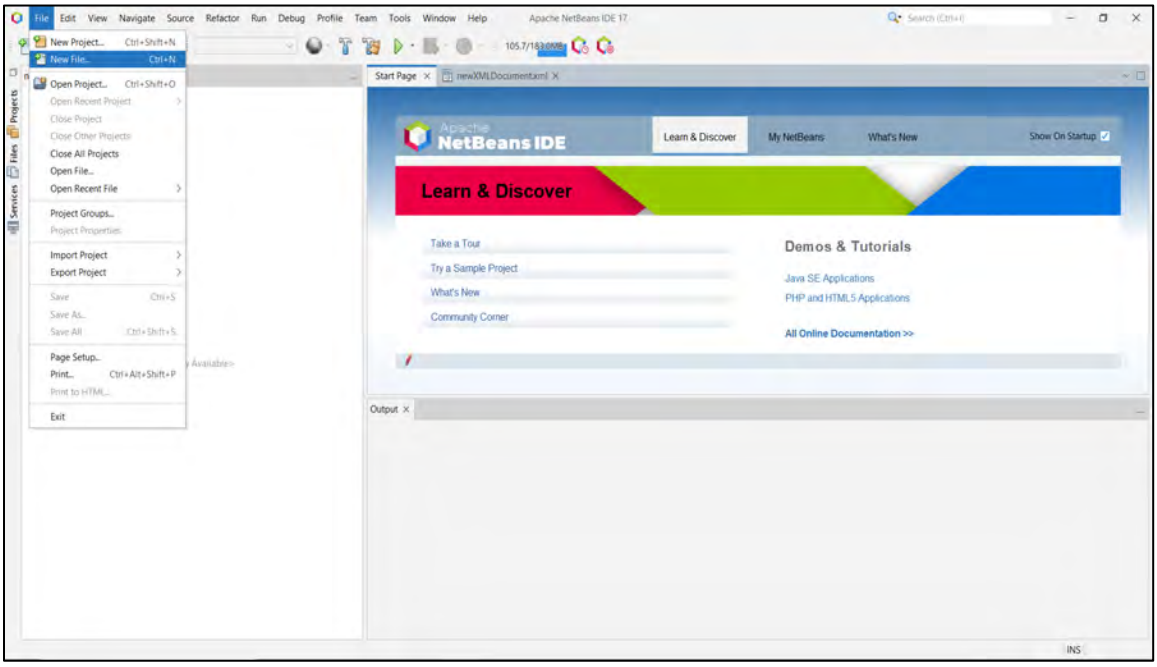


Figure 1.2: Creating a New File in NetBeans IDE

New File dialog box will appear, as shown in Figure 1.3.

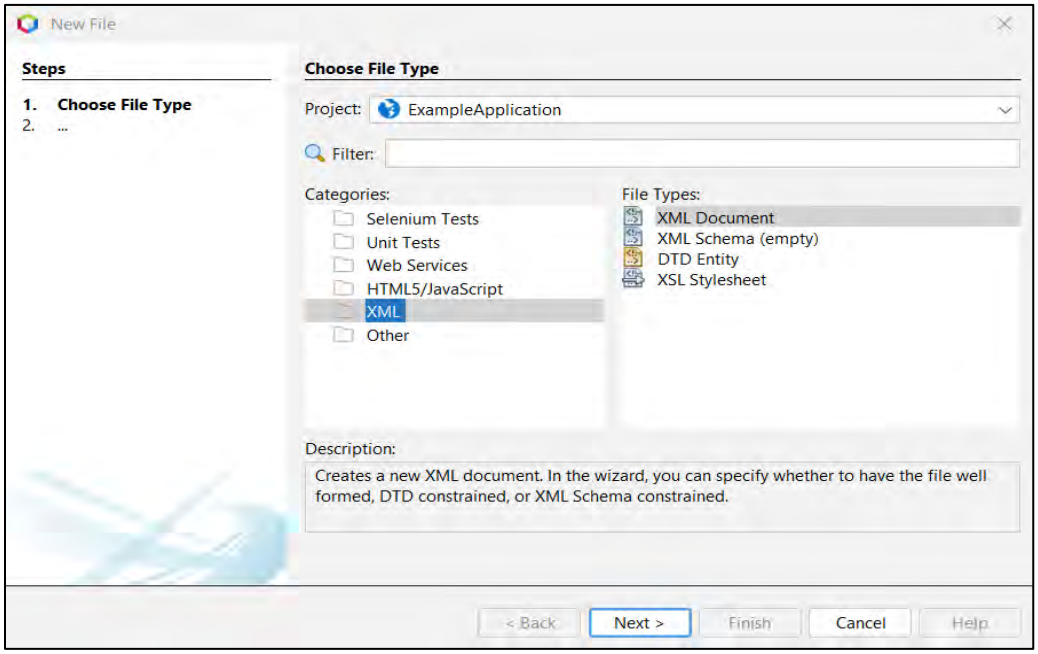


Figure 1.3: Specifying Project Details

3. Select **Categories** as XML and XML Document as shown in Figure 1.3.
4. Click **Next**.
5. Provide the FileName and Folder in which the file should be saved.
6. Select the document type as shown in Figure 1.4.

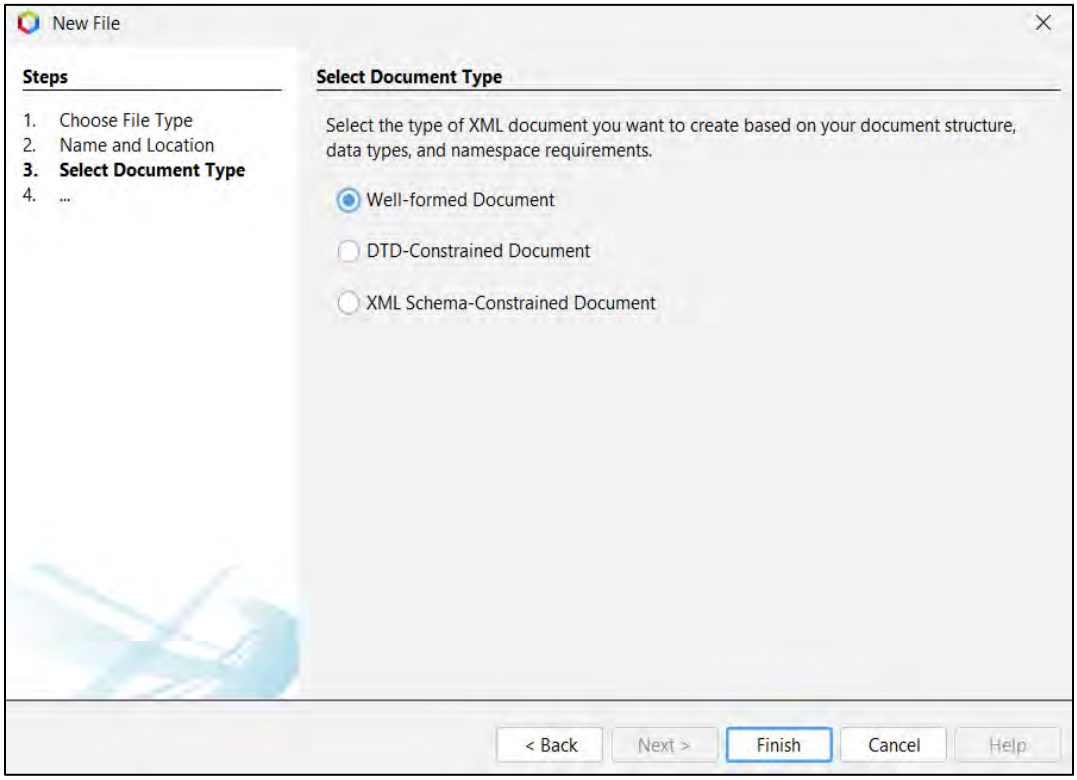


Figure 1.4: Selecting Document Type

7. Click **Finish**. A default document will be created with `<root>` tags.

The programmer can edit this newly created file as per requirements. NetBeans can also be used to access an already existing file and edit it if required.

To open a file, click **File** → **Open File** and then, select the file from the path where it was saved. The XML file opened in NetBeans can be seen as shown in Figure 1.5.

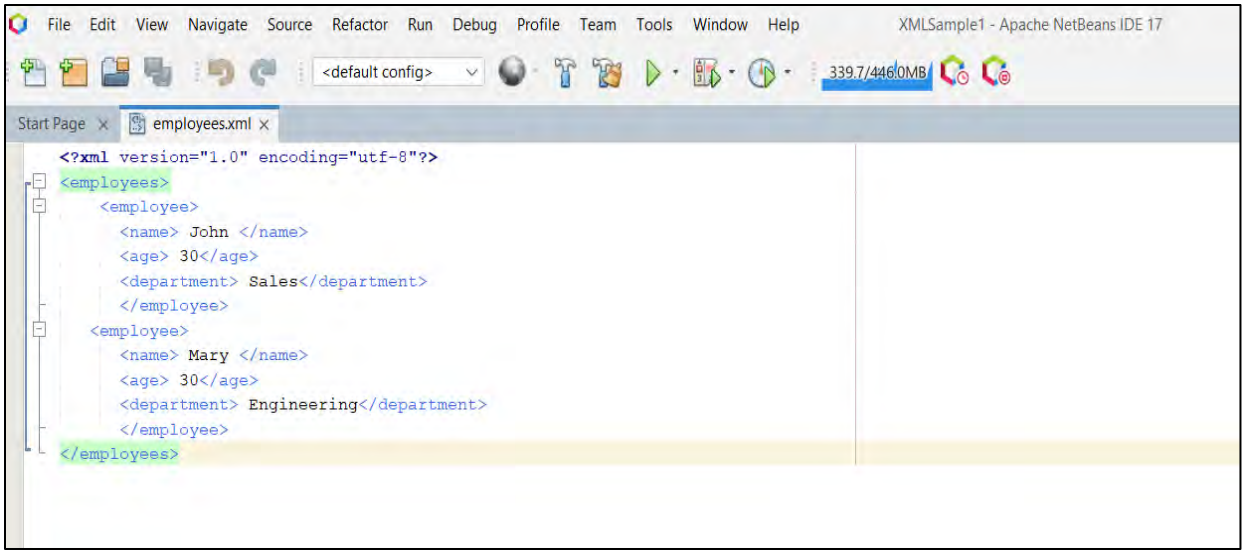


Figure 1.5: Opening an Existing File

An XML file can be viewed in a browser and it will be displayed as shown in Figure 1.6.

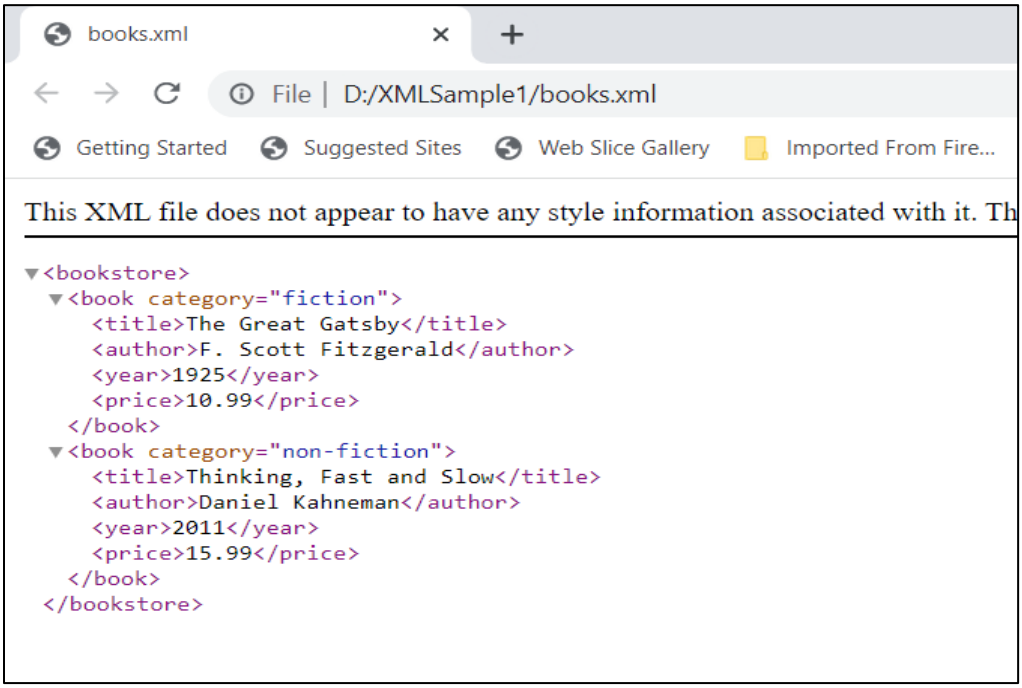


Figure 1.6: XML Output in Browser

In Figure 1.6, the browser output also mentions that there is no style information associated with the XML data. One can set style for XML documents, which will be discussed later. XML has found applications in different domains, such as data exchange, Web services, configuration files, document representation, and more.

Some key features of XML are as follows:

Markup language

- As a markup language, XML uses tags to specify the elements that make up a document. These tags, which are wrapped in angle brackets, are used to denote the data structure.

Platform independent

- Platform independence of XML allows it to work with any operating system and any programming language.

Human-readable

- XML documents are legible by humans, making it simple for programmers to read and comprehend the data's structure.

Extensible

- As XML is flexible, programmers can create custom tags and attributes to represent data in a manner unique to their application.

Supports Unicode

- XML is capable of handling text in any language because it supports Unicode.

Separation of data and presentation

- As XML separates the data from its presentation, the same data can be displayed in a variety of ways to suit the demands of the programmer.

Validation

- With XML, the programmer can verify that data adheres to a certain standard by comparing it to a certain schema.

Easy to parse

- As XML is simple to parse, computers can process it quickly and effectively.

Widely adopted

- Web services, data interchange, and data storage are just a few of the applications that use XML, which has gained widespread adoption.

1.2 XML Technologies

XML has a wide range of applications and is often utilized in conjunction with different technologies, such as Web services, databases, and content management systems.

Following are some of the most popular XML technologies facilitating this:

XML Schema

- An XML document's structure and content are defined using XML Schema. It offers a set of guidelines for validating data against a certain schema.

XSLT

- Extensible Stylesheet Language Transformations (XSLT) is a programming language that converts XML files into HTML or PDF files. It gives programmers access to a potent set of tools for formatting and manipulating XML data.

XPath

- XPath is used to navigate through XML documents and pick out particular nodes. It offers a strong set of capabilities for XML data querying.

XQuery

- Data from XML documents can be retrieved and queried using XQuery. It offers a potent set of tools for XML data analysis and searching.

SOAP

- Over the Internet, structured data can be transferred between several applications using the Simple Object Access Protocol (SOAP). It offers a standard method of application-to-application communication.

RSS

- Content can be published and shared online using Really Simple Syndication (RSS). It offers a uniform method of distributing content, such as blog entries or news items.

Atom

- Atom, which is used to publish and share content online, is comparable to RSS. Compared to RSS, it offers a format that is more expandable and versatile.

Resource Description Framework (RDF)

- RDF is used to describe and exchange metadata about Web resources. It offers a uniform method of describing resources, including Web pages or photos.

1.3 XML Attributes and Comments

Attributes provide added details such as properties, characteristics or metadata about the data represented by the element. Furthermore, these are enclosed within the opening tag of the element, where they define attributes composed of a name and a value, separated by an equal sign. Code Snippet 2 depicts the 'category' element, which is used to specify whether a book is fiction or non-fiction.

Code Snippet 2:

```
<book category="fiction">
 <title>The Great Gatsby</title>
 <author>F. Scott Fitzgerald</author>
 <year>1925</year>
 <price>10.99</price>
</book>
```

Comments in XML are used to add descriptive text within an XML document which serve as notes for programmers. Also, they are used to understand the purpose, structure, or any other relevant information. They are treated as data and ignored by XML parsers and processors. Comments are enclosed in **<! --** and **-->** can span multiple lines too, as shown in Code Snippet 3.

XML comments are beneficial for documentation, collaboration, and understanding the intent behind the XML document. Furthermore, they help to improve the readability and maintainability of XML files, making it easier for programmers or other stakeholders to work with, and understand the data.

Code Snippet 3:

```
<!--
 This is a comment in XML.
 It can span multiple lines and is used to provide additional
information.
-->
<book category="fiction">
 <title>The Great Gatsby</title>
 <author>F. Scott Fitzgerald</author>
 <year>1925</year>
 <price>10.99</price>
</book>
```

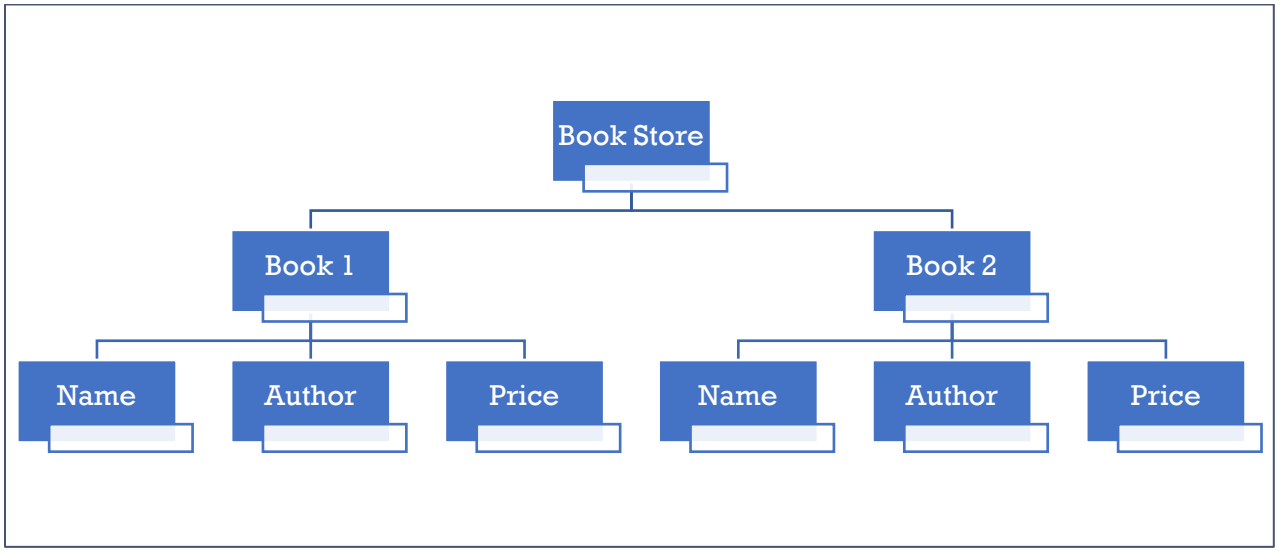
1.4 XML Tree

Each element is a node in the hierarchical tree-like structure that represents the data structure in XML. Root node is the parent element which contains other elements (the

child element) as its immediate content. The tree branches out into child nodes, each with the potential to have its child nodes.

For instance, the XML tree represents a simple fictional structure of bookstore. The root element is <bookstore> and it has two child elements <book>. Each <book> element has child elements such as <title>, <author>, and <price>. The elements and attributes are organized hierarchically, forming the XML tree.

XML tree can be visualized as follows:



Code Snippet 4 illustrates an XML document, which contains details about a corporation.

Code Snippet 4:

```
<company>
 <department name="Sales">
  <employee name="John" age="30"/>
  <employee name="Mary" age="25"/>
 </department>
 <department name="Engineering">
  <employee name="Tom" age="35"/>
  <employee name="Kate" age="28"/>
  <employee name="David" age="40"/>
 </department>
</company>
```

The visual representation of this data is given in Figure 1.7.

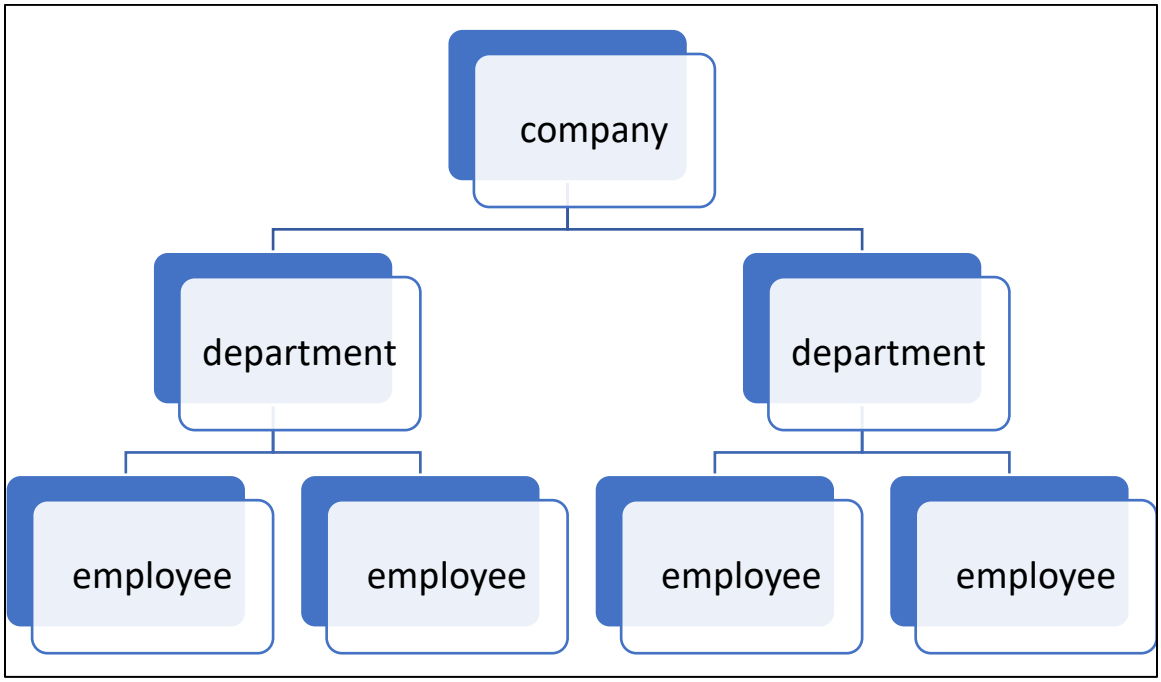


Figure 1.7: XML Tree

On account of this tree-like structure, the data is simple to traverse and manipulate using programming languages and tools that support XML.

1.5 XML Validations

XML validation checks an XML document against a predefined set of rules or constraints to ensure its conformance to a certain XML schema or Document Type Definition (DTD). Additionally, it aids in guaranteeing the proper structure and adherence to the expected format of the XML document, while also achieving any designated data requirements.

XML validation helps ensure the integrity and correctness of XML documents. It allows for premature detection of errors facilitating better data quality and interoperability in XML-based applications.

Schema validation involves using specific tools or libraries that support preferred schema languages such as DTD or XSD. These tools check if XML documents adhere to the defined schema rules. If not, validation errors are reported.

XML validation is vital in data integration, Web services, configuration files, and other XML-related applications. It helps to confirm XML data's reliability, consistency, and quality, allowing efficient data exchange and processing.

1.6 XML DTD and its Types

The structure and regulations for producing legitimate XML documents are set forth in the XML DTD. A DTD is a file that specifies the elements, properties, and rules for combining them that can exist in an XML document. By defining the element types that appear in the document and each element's characteristics, DTDs determine the structure of an XML document. They also specify any limitations on the content of elements, such as data types or acceptable values and the order in which elements occur. DTDs can be used to validate XML documents by ensuring they follow the DTD's rules which guarantee that the XML document is valid, well-formed, and that applications can handle it effectively.

Utilizing DTDs has several benefits, one of which is that they are often straightforward to comprehend. They are created and maintained using basic text editors and have a clear syntax based on a subset of Standard Generalized Markup Language (SGML). The inability to specify complicated content models and the lack of support for data types outside strings are two drawbacks of DTDs. DTDs are frequently replaced by XML Schema when XML documents are more complicated.

Overall, the XML DTD is a crucial tool for defining the structure and standards of XML documents, ensuring they are valid and well-formed.

A suitably formatted XML document would require adhering to following guidelines:

- An XML declaration must come first.
- A single root element that encloses all other tags in the page must be distinct.
- The appropriate end tag must be used to close each element.
- XML entities for special characters should be used.
- All attribute values must be enclosed in quotations.

Code Snippet 5 illustrates a straightforward DTD which outlines the format of an XML document for the list of books.

Code Snippet 5: bookstore.dtd

```
<!DOCTYPE bookstore [
 <!ELEMENT bookstore (book+)>
 <!ELEMENT book (title, author, year)>
 <!ELEMENT title (#PCDATA)>
 <!ELEMENT author (#PCDATA)>
 <!ELEMENT year (#PCDATA)>
 <!ATTLIST book id ID #REQUIRED>
]>
```

This DTD defines a bookshop element that includes one or more book elements. A title, author, and year element can be found in each book element. According to its definition as just containing Parsed Character Data (PCDATA), the title, author, and year components can each include any type of text.

Additionally, the DTD specifies an ID-typed necessary id attribute for the book element. An element within the document can be uniquely identified using the ID type, a specific type in XML.

Code Snippet 6 exemplifies an XML document that adheres to this DTD.

Code Snippet 6: books-new.xml

```
<?xml version="1.0"?>
<!DOCTYPE bookstore SYSTEM "bookstore.dtd">
<bookstore>
 <book id="123">
  <title>The Hitchhiker's Guide to the Galaxy</title>
  <author>Douglas Adams</author>
  <year>1979</year>
 </book>
 <book id="456">
  <title>The Lord of the Rings</title>
  <author>J.R.R. Tolkien</author>
  <year>1954</year>
 </book>
</bookstore>
```

Two book elements are contained in the bookshop element of this XML file. Each book element has a title, author, and year element and a mandatory ID-type id property. The title, author, and year elements' content are PCDATA; any text can be used in them.

1.6.1 DTD Types

Internal DTD

The DTD is called internal if elements are stated within the XML files. The standalone property in the XML declaration must be set to 'yes' to refer to it as internal DTD. This indicates that the declaration is independent of outside sources.

Syntax

Following is the syntax of internal DTD:

```
<!DOCTYPE root-element [element-declarations]>
```

where, `root-element` is the root element's name and `element-declarations` is the section within which programmer declares the elements.

Example

Code Snippet 7 is a simple example of internal DTD.

Code Snippet 7: internaldtd.xml

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "yes" ?>
<!DOCTYPE address [
  <!ELEMENT address (name,company,phone)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT company (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
]>

<address>
  <name>John Doe</name>
  <company>Stanleys</company>
  <phone>(011) 123-4567</phone>
</address>
```

Following explains the code:

Start Declaration – Begin the XML declaration with following statement:

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

DTD – Immediately after the XML header, the document type declaration follows, commonly called the DOCTYPE.

```
<!DOCTYPE address [
```

The element name in the DOCTYPE declaration begins with an exclamation point (!). The DOCTYPE notifies the parser that this XML document has a DTD attached.

DTD Body – The body of the DTD comes after the DOCTYPE declaration when the Programmer declares elements, attributes, entities, and notations.

```
<!ELEMENT address (name,company,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT company (#PCDATA)>
<!ELEMENT phone_no (#PCDATA)>
```

Several elements are declared here that make up the vocabulary of the <name> document. `<!ELEMENT name (#PCDATA)>` defines the element *name* to be of type `'#PCDATA'`. Here, #PCDATA refers to parseable text data.

End Declaration – Finally, a closing bracket and a closing angle bracket (`]>`) are used to close the DTD's declaration section. The definition is effectively finished and the XML content immediately follows.

Rules

There is only one place in the document where the document type declaration appears: at the beginning, just after the XML header.

The element declarations must begin with an exclamation point, such as DOCTYPE declaration does.

The element type of the root element must match the Name in the document type declaration.

External DTD

An external DTD is stored in a separate file and referenced in an XML document by giving the system attributes, which can either be legitimate.dtd file or a legitimate URL, they are accessed. The standalone property in the XML declaration must be set to 'no' to refer to it as an external DTD. This indicates that the declaration contains data from outside sources.

Syntax

Following is the syntax for external DTD:

```
<!DOCTYPE root-element SYSTEM "file-name">
```

where, *file-name* is the file with *.dtd* extension.

Example

Code Snippet 8 is an example of an XML document that makes use of an external DTD.

Code Snippet 8: address.xml

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no" ?>
<!DOCTYPE address SYSTEM "address.dtd">
```

```
<address>
  <name>John Doe</name>
  <company>Stanleys</company>
  <phone>(011) 123-4567</phone>
</address>
```

The content of the DTD file **address.dtd** is as shown in Code Snippet 9.

Code Snippet 9: address.dtd

```
<!ELEMENT address (name,company,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT company (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
```

Types

Use either system IDs or public identifiers to refer to an external DTD.

System Identifiers

Programmer can indicate the location of an external file containing DTD declarations using a system identifier. Following is the syntax:

```
<!DOCTYPE name SYSTEM "address.dtd" [...]>
```

In the syntax, it specifies the keyword 'SYSTEM' and a URI reference indicating the document's location.

Public Identifiers

Public identifiers offer a way to find DTD resources and are expressed as follows:

```
<!DOCTYPE name PUBLIC "-//Beginning XML//DTD Address
Example//EN">
```

The code depicts that it starts with the keyword PUBLIC and is followed by a unique identifier. Public identifiers identify an entry in a catalog. Although public identifiers can take any shape, Formal Public Identifiers or FPIs are popular. Formal Public Identifiers are specific codes or names used to uniquely identify entities or resources in various domains. These identifiers are typically standardized and widely recognized to ensure consistency and interoperability.

1.7 Working with DTD

Following steps are used to make DTD function:

- Start by establishing a DTD file for the specific XML document.
- Sketch out the document's structure.
- At this point, the programmer can choose whether to construct the DTD using internal or external references.
- Ensure that the file contains all necessary components, entities and characteristics.

DTD is essentially a technique to precisely explain the XML language which specifies an XML document's structure, legal components, and attributes. It is useful for comparing the vocabulary and validity of an XML document's structure to various grammatical conventions of an appropriate XML language.

An XML-DTD can be specified and maintained in two ways: inside the document or in a separate document.

If an XML document has the appropriate syntax, it is well-formed. Moreover, an XML document validated against DTD is valid and well-formed.

DTD can be used to quickly validate an XML document that has been developed.

1.8 XML Schema

The structure and restrictions of XML documents are described in XML Schema which offers a collection of instructions and rules for producing XML documents, including rules for the document's structure, the acceptable data types, and the connections between elements and attributes.

The structure of the target XML document is described by an XML Schema, which is usually defined in a separate XML file. Components, attributes, and data types used in the document are determined by the schema, together with any limitations or guidelines applicable to these elements.

The main benefit of utilizing XML Schema is its ability to validate XML documents. A document's well-formedness, conformance to structure specified in the schema, and validity and consistency of data it includes can all be verified by defining a schema for that particular document.

Sequences, options, and groups are just a few examples of the sophisticated kinds that would be defined using XML Schema. More complicated data models and applications can be constructed easier by using these kinds to determine more complex structures in XML documents.

XML Schema is a crucial tool for working with XML documents, ensuring that documents are well-formed and valid and allowing the creation of more complex data models and applications.

To perform validation on an XML file named 'employees.xml', the description of document's structure and employee data are provided.

Code Snippet 10 defines the structure of the `employees` XML document.

Code Snippet 10: employees.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="employees">
  <xs:complexType>
   <xs: sequence>
    <xs:element name="employee" maxOccurs="unbounded">
     <xs:complexType>
      <xs: sequence>
       <xs:element name="name" type="xs:string"/>
       <xs:element name="age" type="xs:int"/>
       <xs:element name="department" type="xs:string"/>
      </xs: sequence>
     </xs:complexType>
    </xs: element>
   </xs: sequence>
  </xs:complexType>
 </xs: element>
</xs: schema>
```

This schema defines that the `'employees'` element should contain one or more `'employee'` elements, each with a `'name'`, `'age'`, and `'department'` element.

An XML document named employee.xml, which will be validated by the employees.xsd file, is shown in Code Snippet 11.

Code Snippet 11: employee.xml

```
<?xml version="1.0" encoding="utf-8"?>
<employees>
  <employee>
   <name> John </name>
   <age> 30</age>
   <department> Sales</department>
   </employee>
  <employee>
   <name> Mary </name>
   <age> 30</age>
   <department> Engineering</department>
```

```
    </employee>
</employees>
```

1.8.1 Advantages of XSD over DTD

In place of DTD, XSD can be used to specify the structure and rules of XML documents. Following are some benefits of XSD over DTD:

Data type support	XSD offers support for a greater variety of data kinds, including as date/time, numeric, and the user-defined types. This enables XML document validation to be more accurate.
Better namespace support	XSD offers improved namespace support for XML, allowing elements and attributes to be created in various namespaces and facilitating more accurate validation of XML documents.
Support for more complicated content models	XSD supports more sophisticated content models than DTD, allowing for the definition of nested elements and repeating elements with various iterations. This enables XML document validation to be more accurate.
Stronger type checking	XSD offers type checking that is more robust than DTD's, which can aid in identifying issues early in the development cycle.
Better extensibility	XSD supports extensibility more effectively than DTD, which makes it simpler to extend current schemas and develop new schemas that recycle old components.

Overall, XSD has stronger support for data types, namespaces, complicated content models and extensibility than DTD, making it a more powerful and flexible schema language. However, because of its simplicity and usability, DTD is still beneficial for straightforward XML documents and continues to be a popular schema language.

1.9 Summary

- XML is a flexible markup language used for storing and exchanging structured data and validation ensures correctness and adherence of XML documents.
- XML technologies enable structured data representation for various purposes, that enables interoperability, data transformation, and efficient data management across different systems and platforms.
- XML DTD provides resources to define the structure and constraints of XML documents, ensures data integrity, and facilitates interoperability between systems that exchange XML data.
- XML schema ensures that documents are well-formed to create complex models and applications.
- XSD offers support for a greater variety of data types, namespaces, complicated models, and robustness.

1.10 Check Your Progress

1. What does XML stand for?

A.	Extensible Markup Language	C.	Extra Markup Language
B.	Expressive Markup Language	D.	Extended Markup Language

2. Which XML technology is used to transform XML data into different formats, such as HTML or PDF?

A.	XPath	C.	XQuery
B.	XSLT	D.	XSD

3. In DTD, the “element” Keyword is used to:

A.	Define attributes within an XML document	C.	Define complex types for elements
B.	Define data types for elements	D.	Define the root element of an XML document

4. Which technology is commonly used to validate XML documents against an XML schema?

A.	XMLSchema	C.	XQuery
B.	XSLT	D.	XSD

5. XML Schema is used to:

A.	Query and retrieve data from XML documents	C.	Validate the structure and data types of XML documents
B.	Navigate through XML documents and select specific elements or data	D.	Transform XML data into different formats

1.10.1 Answers for Check Your Progress

1.	A
2.	B
3.	D
4.	A
5.	D

Try It Yourself

Task 1: Create a new file and save it with a '.xml' extension (Example 'mydocument.xml') and try to open it via Chrome browser. The file should contain following data:

Sr.No.	Student	Age
1	John	6
2	Kristen	7
3	Alex	8
4	Patrick	9
5	Lukasz	11

Task 2: Make a .dtd and .xsd file for validating the XML file made in Task 1.

Task 3: Implement an XML schema for a user registration form, using enumeration to restrict possible values for gender. Create an XML schema for a menu with food items, using restrictions to limit the range of calorie values.

Session 2

XML Parser and DOM

This session describes an overview of XML parser and explains benefits of using XML Parser and XML DOM. Further, it also identifies the techniques involved in XML Database and XML Namespaces.

Objectives

- Describe XML Parsers
- Outline XML Document Object Model (DOM) and list various types of XML Database
- Explain XML Namespaces

2.1 XML Parser

A software program or library known as an XML parser is used to read, examine, and understand XML documents.

XML parsers are important for processing XML documents in various computer languages and environments. Following are typical tasks carried out by XML parsers:

- **Parsing:** XML parsers read an XML document and check that it adheres to the XML specifications regarding syntax. The XML document must follow the standards of XML syntax, such as having a single root element, properly nested elements, and adequately quoted attribute values, to be considered well-formed.
- **Validation:** A few XML parsers validate against a DTD or an XML schema. While DTDs specify structure and constraints of XML documents in a distinct syntax, XML schemas define structure, data types, and constraints of XML documents.

Validation ensures that the XML document complies with the requirements set forth for its structure and content to ensure the integrity of the data. The DOM developed by the XML parser, is a data structure resembling a tree that depicts the hierarchical organization of the XML text. It offers an XML document format that can be quickly browsed, changed, and queried using programming languages.

DOM-based XML parsers load the complete XML document into memory, enabling random access and data manipulation, but they are not suitable for large XML files due to memory limitations.

Event-driven parsing: Some XML parsers use this method, creating events while the parser scans through the XML document. A programmer can process the XML data by handling triggered events, for instance, when an element or an attribute is encountered. This method does not require loading the entire document into memory, making it memory-efficient and suited for processing huge XML documents sequentially.

Data retrieval is made possible via the Application Programming Interfaces (APIs) that XML parsers offer, which let programmers access and retrieve data from XML documents. For instance, the programmer can use methods and properties to browse

the DOM tree in DOM-based XML parsers to extract element values, attribute values, and other data. Programmers can handle events with event-driven XML parsers to extract data when the parser runs through various XML document sections.

- **Error handling:** XML parsers can find and report XML document errors, such as syntax mistakes, validation mistakes or other problems. They include techniques for handling and reporting failures, such as call-back functions or exception handling.
- **Options for processing:** XML parsers frequently offer options and settings that let the programmer customize the parsing behavior, such as managing whitespace, disregarding comments or configuring validation options. It permits customization and flexibility in the parsing and processing of the XML content.

Web services, data exchange, configuration files, data storage, and many other applications use XML parsers extensively. Parsers come in several varieties. Some of the most popular ones include DOM, Simple API for XML (SAX), Streaming API for XML (StAX), and Java Architecture for XML Binding (JAXB), individually with benefits, and applications.

The best XML parser for a given application would rely on its requirements, including the size of the XML documents, the necessity for validation, and the ideal processing flexibility level. Following are some XML parser examples written in various programming languages:

JAVA:

DOM Parser

- Java provides built-in support for DOM parsing through the `javax.xml`.parsers package. Examples of DOM-based parsers in Java include `DocumentBuilder` and `DocumentBuilderFactory` classes, which allow for parsing and manipulating XML documents as a tree-like structure in memory.

SAX Parser

- Java also provides a SAX parser through the `javax.xml`.parsers package. SAX parsers are event-driven and provide call-back methods triggered as the parser reads through the XML document. Examples of SAX-based parsers in Java include `SAXParser` and `DefaultHandler` classes, which allow for handling events as the parser encounters elements, attributes and other XML components.

StAX Parser

- Java also supports StAX parsing through the `javax.xml.stream` package. StAX parsers are event-driven such as SAX, but they provide a more intuitive and stream-like programming model for processing XML documents. Examples of StAX-based parsers in Java include `XMLStreamReader` and `XMLInputFactory` classes, which allow for streaming reading and processing XML documents.

PYTHON:

ElementTree

- Python's built-in `ElementTree` module provides a simple and efficient DOM-based XML parsing API. It allows for parsing and manipulating XML documents as a tree-like structure in memory and provides methods for navigating and managing XML data.

lxml

- lxml is a popular third-party library for parsing XML documents in Python. It provides both DOM and SAX parsing APIs, allowing tree-based and event-driven parsing. lxml is known for its performance and ease of use and it supports both the ElementTree and XPath syntax for querying and manipulating XML data.

C#:

XmlDocument

- C#/.NET provides the `XmlDocument` class in the `System.Xml` namespace, a DOM-based XML parser that allows for parsing and manipulating XML documents in memory. It provides methods for navigating and modifying the DOM tree and supports XPath for querying XML data.

XmlReader

- C#/.NET also provides the `XmlReader` class in the `System.Xml` namespace, a forward-only, event-driven SAX-like XML parser. It provides call-back methods for handling events as the parser reads through the XML document, making it memory-efficient for processing large XML documents.

2.2 XML DOM

An XML document is represented in memory as a tree-like structure using the programming interface known as XML DOM. It offers a method for interacting and manipulating an XML document's elements, attributes, and data using programming languages.

Individual element, attribute, and text node in an XML document is represented as a node in the XML DOM as a node in a hierarchical tree structure. The nodes are arranged in a parent-child form, with the top-level node serving as the XML document's root element.

The XML DOM makes it possible to access and edit the data included in the XML document by navigating, altering, and querying the nodes in the tree.

A few essential XML DOM concepts are as follows:

Nodes

- Nodes are the XML DOM tree's building blocks. They stand in for the XML document's elements, attributes and text nodes. The DOM API can access and change the node's properties, such as its name, value and attributes.
- There is a parent-child relationship between nodes in the XML DOM tree. All other nodes in an XML document are children or descendants of the root element, represented by the top-level node. Nodes can have children, siblings or both, making navigating and modifying the XML data possible.
- Nodes in the XML DOM tree has properties and methods that provide access to and modify their attributes, values and other attributes. The DOM API, for instance, offers ways to add or delete child nodes, access element content, get and update element properties and more.

Traversal

- The XML DOM offers tools for moving around the tree and between nodes. For instance, the DOM API enables access to a specific node's parent, children and siblings, enabling navigation and manipulation of the XML data.

Modifying XML data

- An XML document's data can be changed using the XML DOM. For instance, it offers ways to add, remove and update the XML document's elements, attributes and text nodes, enabling XML data manipulation.

XML data querying

- The XML DOM offers XML document data querying techniques. As an example, it supports XPath, a potent language for XML data querying that enables searching, filtering and retrieval of specific components or attributes from the XML document.

Several programming languages, including Java, Python, C#, and JavaScript, employ the XML DOM to modify and interact with XML documents. It is suitable for applications that require parsing, validating, manipulating, and creating XML documents because it offers a versatile and powerful approach to accessing, and editing XML data.

2.3 XML Database

A database is a structured collection of data organized for efficient storage, retrieval, and management. It serves as a central repository for storing and managing various types of information, making it easy to access, update, and manipulate data.

Databases are commonly used in various applications and industries, including business, healthcare, finance, e-commerce, and so on.

XML databases are created to manage and store data in the XML format. XML databases provide storage, querying, and processing options for XML data.

There are various XML database types such as:

Native XML databases: These databases were created to store and manage XML data in its original format. In other words, native XML databases are databases that store XML documents and data and can allow manipulation of this data. They offer transaction management, indexing, and querying tailored for XML data. Databases that are native to XML include BaseX and MarkLogic.

XML-enabled databases: They are relational databases that have been enhanced to support XML information. They can provide XML-specific functionality through query languages or APIs and store XML data as text or binary data in relational tables. Databases such as Oracle Database, IBM DB2, and Microsoft SQL Server are examples of those that support XML.

Hybrid databases: These databases combine the features of relational databases and native XML. They offer relational data and SQL queries in addition to being able to store XML data natively. The hybrid databases Sedna and Tamino are two examples.

In applications such as content management systems, data integration, and online services, where XML data is a crucial or significant component of the data storage and processing requirements, XML databases are frequently utilized.

They enable efficient storing, querying, and manipulation of XML data. It can also be applied to the systematic and orderly management of enormous amounts of XML data.

Following is an example of a straightforward native XML database:

Code Snippet 1: books.xml

```
<library>
  <book id="001">
    <title>The Catcher in the Rye</title>
    <author>J.D. Salinger</author>
    <genre>Fiction</genre>
    <price>9.99</price>
  </book>
  <book id="002">
    <title>To Kill a Mockingbird</title>
    <author>Harper Lee</author>
    <genre>Fiction</genre>
    <price>12.99</price>
  </book>
  <book id="003">
    <title>The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
    <genre>Fiction</genre>
    <price>10.99</price>
  </book>
</library>
```

Code Snippet 1 represents a simple library database with three books, every book having an `id` as attribute and elements such as `title`, `author`, `genre`, and `price`.

Code Snippet 2 is a XQuery example that fetches all the books from the native XML database, books.xml.

Code Snippet 2: xquery

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

The results of this XQuery query are returned as XML and include all the books from the books collection stored in database.

To execute XQuery code on the computer, the programmer can use a tool or programming environment that supports XQuery. One popular choice for running XQuery is the BaseX database system, which provides a Graphical User Interface (GUI) for executing XQuery queries and managing XML data. Following are the steps to set up BaseX and execute a XQuery query:

1. Go to the BaseX Website (https://basex.org/) and download the appropriate version of BaseX that is appropriate for Windows.
2. Follow the installation instructions for given on Website to install the BaseX.
3. After installation, launch the BaseX GUI application. It allows programmer to see a graphical interface, interact with XML data, and execute XQuery queries.

The BaseX GUI can be seen as shown in Figure 2.1.

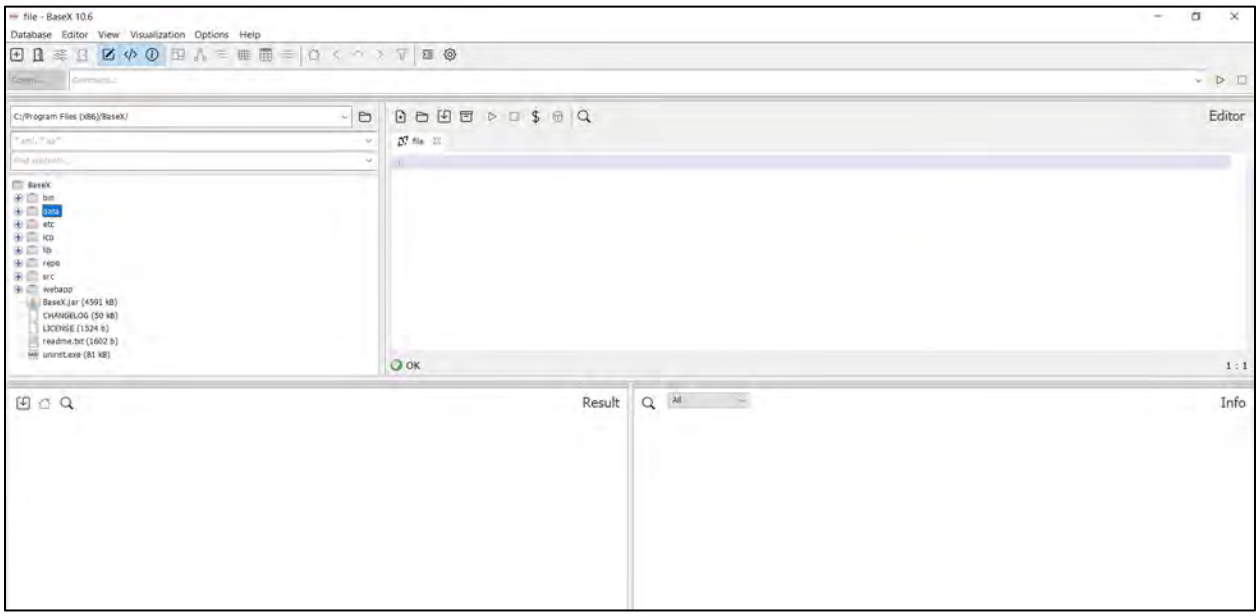


Figure 2.1: BaseX GUI

4. Click **Database** and select **New** to create Database as shown in Figure 2.2.

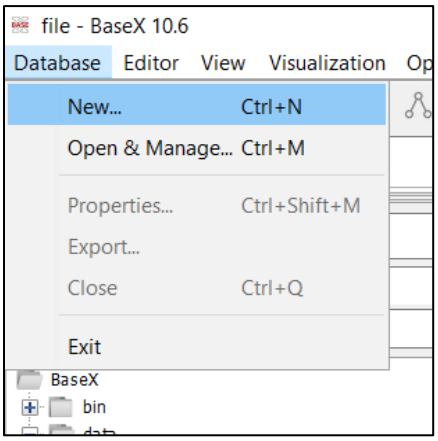


Figure 2.2: New Option

5. In the **Create Database** dialog box, insert the name of the Database to be created and click **OK**, as shown in Figure 2.3. The Database will be created.

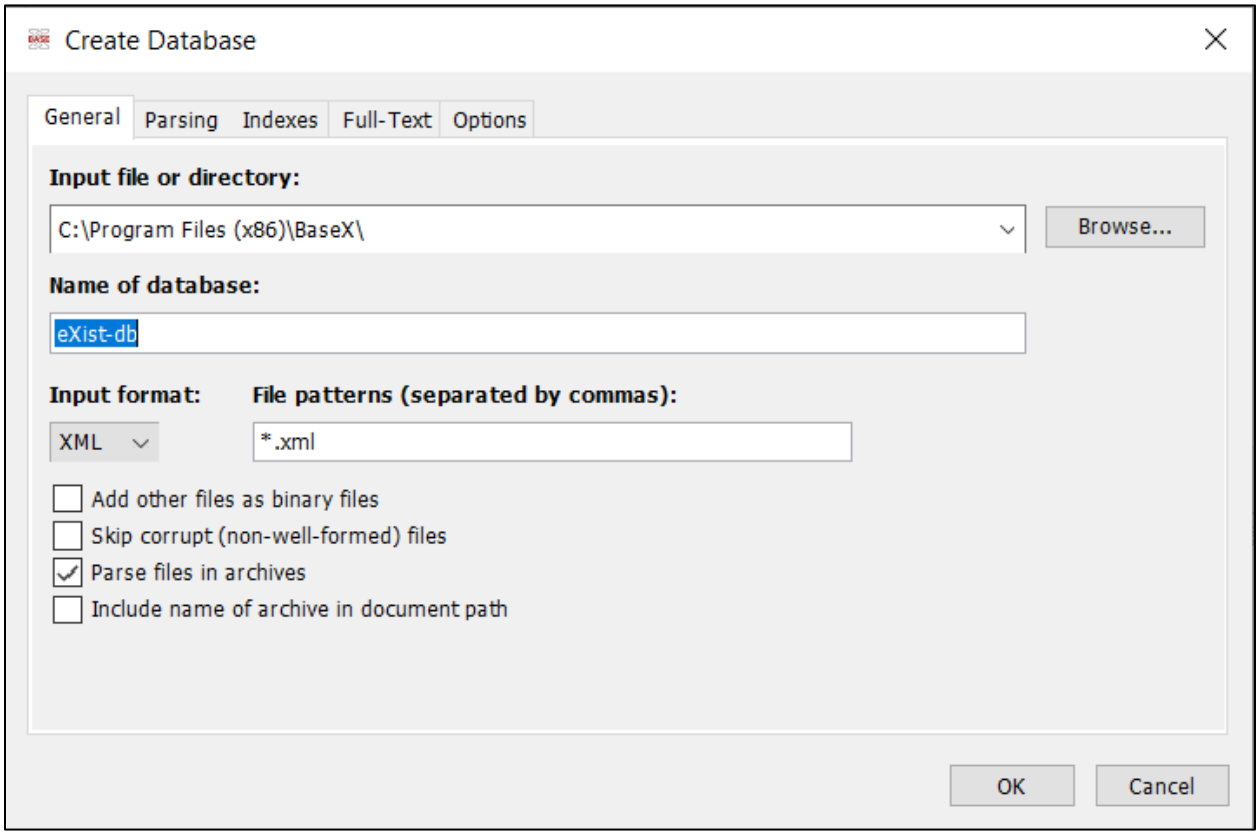


Figure 2.3: Create Database

Specify the name of the database as myDB. The left pane under BaseX displays this as shown in Figure 2.4.

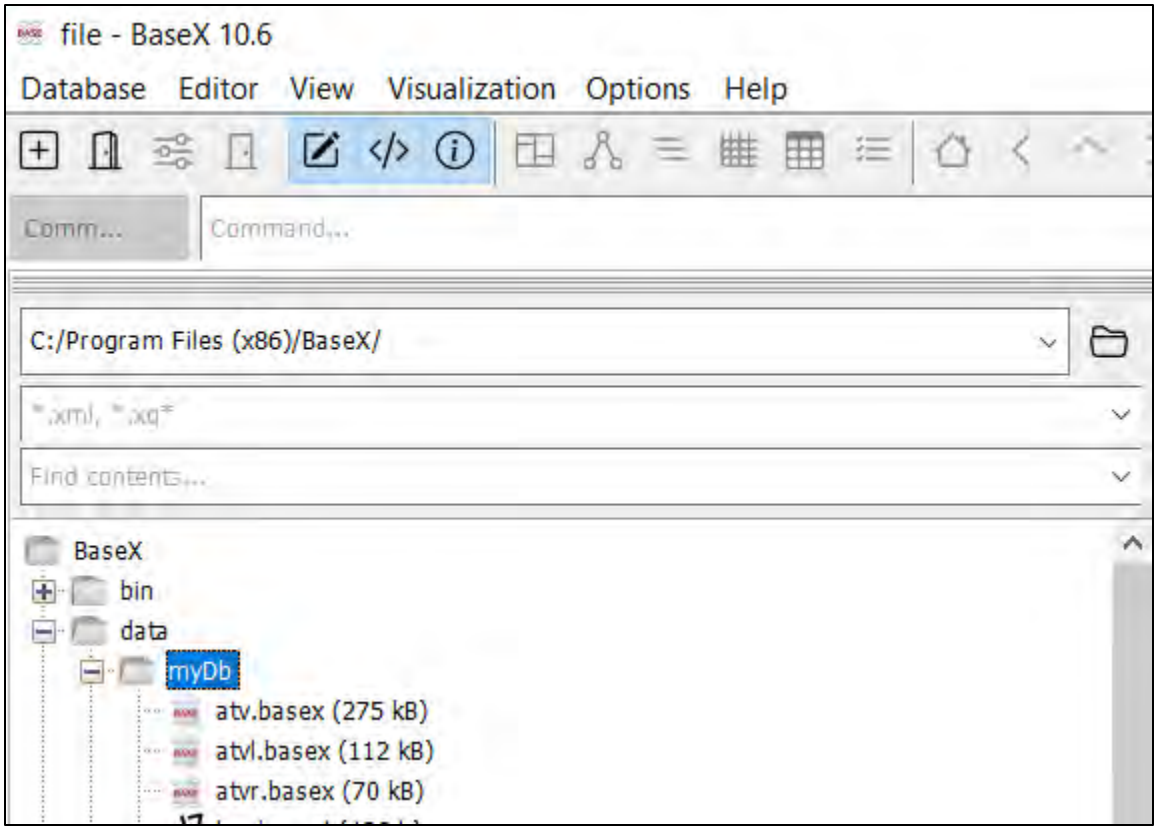


Figure 2.4: Db Created

6. To create a new XML file, click + symbol, add the code from Code Snippet 1 into the blank file and then, save it as `books.xml`, as shown in Figure 2.5.

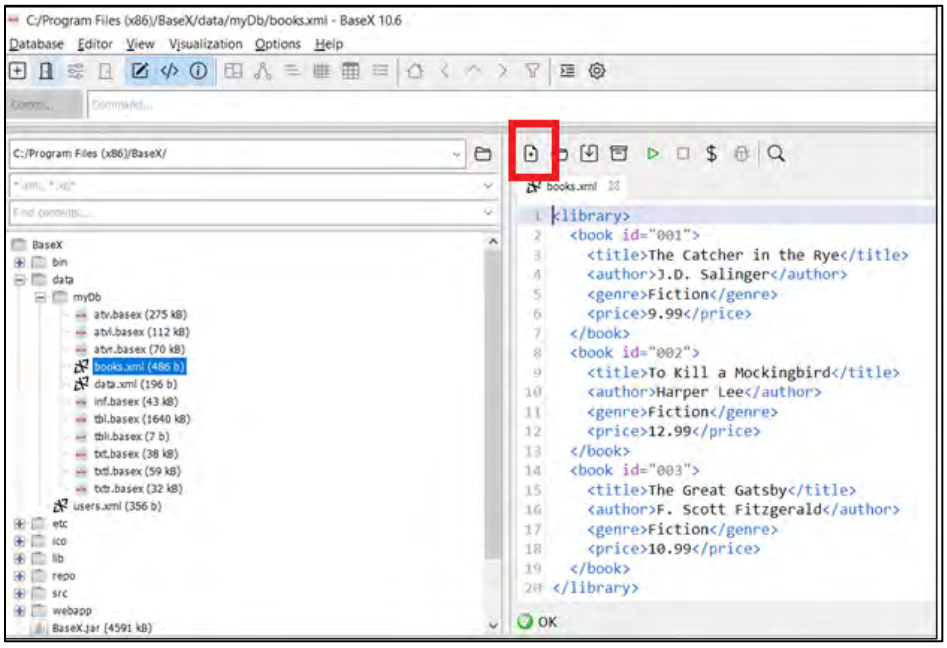


Figure 2.5: Create New File

7. To execute the XQuery in **Code Snippet 2**, open a new file tab. Paste the query in it and click the green Run button, as shown in Figure 2.6.

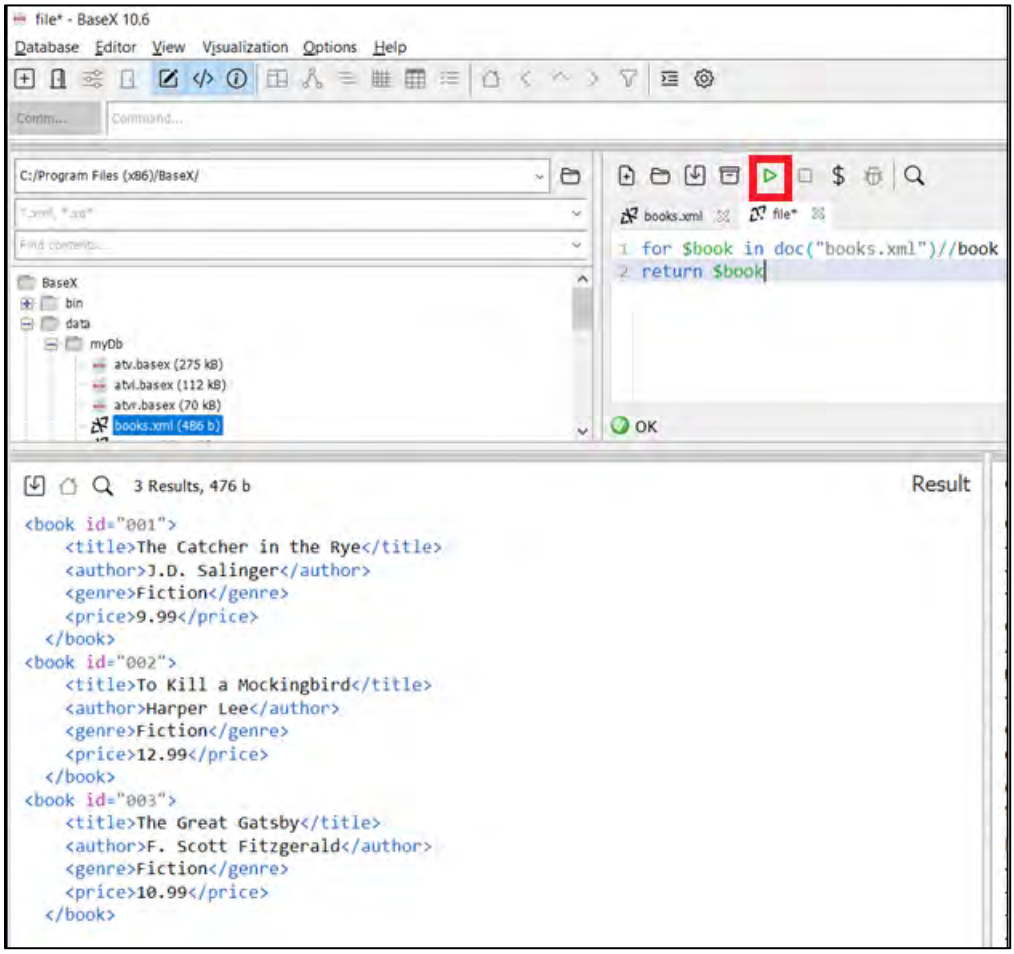


Figure 2.6: Executing Query and Result

The programmer can see the result of the query, which is nothing but XML data with only book nodes.

2.4 XML Namespaces

A namespace is a fundamental concept in computer science and software development that helps organize and manage elements within a system to prevent naming conflicts and provide a way to uniquely identify and access them. Namespaces are used to group related items, such as variables, functions, classes, or other symbols, and ensure their names do not clash with similarly named items from other parts of the codebase. Namespaces are used in various programming languages and operating systems.

XML namespaces avoid naming conflicts in XML documents and uniquely identify elements and properties.

It also uniquely identifies components and attributes in an XML document to avoid naming conflicts when elements or attributes from many sources are joined in a single document.

By associating elements and features with a namespace name, a Uniform Resource Identifier (URI), or a URI reference, XML namespaces enable elements and attributes with the same name to coexist in an XML document.

Namespace declarations in the XML document, placed typically in the root element or in the elements that use the namespaces, are used to define XML namespaces.

Code Snippet 3 depicts an illustration of how to utilize XML namespaces in an XML document.

Code Snippet 3:

```
<catalog xmlns="http://example.com/catalog"
         xmlns:book="http://example.com/book">
  <book:book>
    <book:title>The Catcher in the Rye</book:title>
    <book:author>J.D. Salinger</book:author>
    <book:genre>Fiction</book:genre>
    <book:price>9.99</book:price>
  </book:book>
  <book:book>
    <book:title>To Kill a Mockingbird</book:title>
    <book:author>Harper Lee</book:author>
    <book:genre>Fiction</book:genre>
    <book:price>12.99</book:price>
  </book: book>
</catalog>
```

In this example, the `xmlns` attribute in the catalogue element declares the default namespace as `http://example.com/catalog`. The `xmlns:book` attribute in the catalogue element declares a namespace prefix 'book' associated with the namespace `http://example.com/book`. The elements `book`, `title`, `author`, `genre`, and `price` are prefixed with the namespace prefix `book`, which indicates that they belong to the `http://example.com/book` namespace.

A single XML document can contain many XML vocabularies or schemas without naming conflicts with the help of XML namespaces. They promote interoperability across various XML-based technologies, applications, provides resource for uniquely identifying components, and attributes. To ensure practical parsing and processing of the XML data, it is critical to properly handle namespace prefixes, namespace URIs, and namespace declarations when processing XML documents that employ namespaces.

XML namespaces are used to prevent naming conflicts when combining various XML vocabularies or schemas in a single XML document.

For instance, an XML document could include components from multiple XML schemas or vocabularies, with its namespace, such as XHTML, SVG, and MathML. Namespaces allow these elements to reside in the same XML document without conflict.

Determining the sources of elements or attributes: In an XML document, namespaces can denote the origin or source of elements or attributes. Namespaces, for instance, can identify the origin of individual elements or attributes when combining data from various sources into a single XML document. It can be helpful to when data from many sources requires to be processed or evaluated individually.

Extending current XML vocabularies: XML schemas or vocabularies can be extended using XML namespaces. For instance, a namespace can add unique elements or attributes without changing the original schema or vocabulary if an XML schema or vocabulary does not include a specific element or property required for a particular use case. It enables XML documents to be flexible and extensible.

Enforcing data integrity and validation: XML namespaces can be used to specify guidelines for data integrity and validation. It is possible to create constraints or validation rules for those components or attributes using XML schema or other validation mechanisms by associating them with specific namespaces. Doing so makes it possible to ensure that the data in the XML document follows the desired structure or format.

Interoperability between XML based technologies: XML namespaces provide interoperability between various XML-based applications and technologies. For instance, XML namespaces are frequently employed in XML-based Web services where various components could use different XML vocabularies or schemas. These components can communicate and exchange data in a standardized, compatible way using namespaces.

XML namespaces are crucial for resolving naming disputes and promoting interoperability in XML documents. They offer a resource for explicitly designating elements and attributes and permit blending various XML vocabularies or schemas into a single XML document.

2.5 Summary

- XML is a popular markup language that stores and exchanges structured data in a legible format by humans.
- The Document Object Model (DOM) is a data structure and resembles a tree depicting the hierarchical organization of the XML text.
- XML parsers are used extensively by Web services, data exchange, configuration files, data storage, and many other applications.
- In XML documents, programmers can use namespaces in various ways to avoid naming conflicts and uniquely identify elements and properties.
- XML namespaces are crucial for resolving naming disputes and promoting interoperability in XML documents.

2.6 Check Your Progress

1. _____ is used to read, examine, and comprehend XML documents.

A.	XM Parser	C.	APIs
B.	DOM Tree	D.	SAX

2. Which of the following depicts the hierarchical organization of the XML text?

A.	Event-driven parsing	C.	Document Object Model
B.	Document Type Definition	D.	XML schema

3. What is used by an XML document to represent a tree-lie structure using a programming interface?

A.	Lxml	C.	XML DOM
B.	XML Input Factory	D.	XML Reader

4. What do programming languages such as Java and Python employ to modify and interact with XML document?

A.	XML DOM	C.	Get Tag Name
B.	XML data querying	D.	XML Reader

5. Which type of XML databases store and manage XML data in its original format?

A.	XML-enabled databases	C.	eXist-db databases
B.	Hybrid databases	D.	Native XML databases

2.6.1 Answers for Check Your Progress

Question	Answer
1	A
2	B
3	D
4	A
5	D

Try It Yourself

Task 1: Using BaseX GUI, create a database and create an XML file within the database. The XML file should have root tag as products and the product node should have child nodes as name, price, quantity, and Id.

Task 2: Write a XQuery to display all the products in the XML file.

Session 3

XPath

The session introduces XPath and explains its various elements such as Nodes, Syntax, Operators, Relative, and Absolute Path.

Objectives

- Define XPath and its Expressions
- Define and describe various elements in XPath
- Compare XPath Relative and Absolute Paths
- Explain the use of XPath Operators

3.1 Introduction to XPath

A robust and adaptable language called XPath (also known as XML Path Language), is used to navigate through and pick out specific elements from an XML document. It is frequently used to handle XML data in many programming languages, including Python, Java, and JavaScript.

XPath offers a clear and expressive syntax for traversing and querying XML documents. In an XML document, where elements are arranged into parent-child connections, the hierarchical relationships between them are represented by a tree-like structure.

The placement of an individual or group of elements inside an XML document can be specified using XPath expressions.

XPath also offers a broad range of operators and functions for conducting various operations on XML data, such as comparison, arithmetic, and string manipulation. Axes and path expressions can be coupled with these operators and functions to produce sophisticated and effective queries for extracting and modifying XML data.

XPath is frequently utilized in conjunction with XML technologies such as XML Stylesheet Language Transformations (XSLT) and XML processing.

Additionally, XPath is a strong and flexible language for performing searches on XML documents. Widely used in XML-related technologies for processing and manipulating XML documents, it offers a clear and expressive vocabulary for choosing elements and carrying out operations on XML data.

3.2 XPath Expressions

XML Path Language (XPath) is a query language used for selecting nodes from an XML document. It provides a way to navigate and query the elements and attributes in an XML document, making it easier to extract specific data or perform operations on structured content. XPath expressions are used to define the criteria for selecting nodes within an XML document.

An XPath expression is a mechanism to navigate through and select the nodes from the XML document.

The hierarchy of the elements or attributes in the XML document is represented by the path steps that make up an XPath expression, which is divided into stages by forward slashes (/).

Following are a few examples of how we can use XPath expressions:

How to choose an element by name?

- To select an element by name, use the element name followed by a forward slash. For instance, the XPath expression for selecting all 'book' components in an XML document would be `'/book'`.

Choosing an attribute

- To choose an attribute of an element, use the symbol '@' followed by the name of the attribute. For instance, the XPath expression would be `'/book/@title'` to choose the 'title' attribute of a 'book' element.

Wildcards

- XPath supports `""` and 'node()' as two wildcards. The "" wildcard covers any element and any node (element, attribute, text and so on) is covered by the 'node()' wildcard. For instance, the XPath expression would be `'/bookstore/*'` to select each element under the 'bookstore' element.

Predicates

- Predicates are used to sort objects according to certain qualities or standards. Square brackets ([]) are used to denote predicates. For instance, the XPath expression for selecting all 'book' components with a particular attribute value would be `'/book[@attribute='value']'`.

Axis

- In an XML document, axes describe the navigational direction. 'Child', 'descendant', 'parent', 'ancestor', 'following-sibling', 'preceding-sibling', 'following' and 'preceding' are often used axes. For instance, the XPath expression for selecting all a 'bookstore' element's children would be '/bookstore/child::node()'.

Functions

- A wide range of actions can be carried out on XML documents using XPath's built-in functions, including text manipulation, numerical computations, date and time functions and more. For instance, the XPath expression for selecting all 'book' elements whose 'price' is more than 50 would be `'/book[price > 50]'`.

Parallel Constructions

- Parallel construction in XPath refers to the ability to express multiple location paths or conditions simultaneously in a single XPath expression. It allows the user to select elements or nodes that meet various criteria, which can be especially useful in filtering and navigating XML or HTML documents. The operators used are 'or' and 'and; operators to create complex conditions. For example: `'/book[price > 50 and price>60]'` will select the book in between price 50 and 60.

In many XML technologies that process and manipulate XML data, XPath expressions are a popular and effective tool for traversing and selecting items in XML documents.

Consider an example to understand this. Create an XML document named `Bookstore.xml` as shown in Code Snippet 1.

Then, create `index.html` with the code given in Code Snippet 2. This code uses the XML document that portrays a bookshop with a selection of books and executes some standard operations using XPath. Code Snippet 2 contains the necessary JavaScript and HTML code to execute all the XPath expressions.

Code Snippet 1: Bookstore.xml

```
<bookstore>
  <book category="fiction">
    <title lang="en">The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
    <year>1925</year>
    <price>10.99</price>
  </book>
  <book category="fiction">
    <title lang="en">To Kill a Mockingbird</title>
    <author>Harper Lee</author>
    <year>1960</year>
    <price>12.99</price>
  </book>
  <book category="non-fiction">
    <title lang="en">The Elements of Style</title>
    <author>William Strunk Jr.</author>
    <year>1918</year>
    <price>9.99</price>
  </book>
</bookstore>
```

Code Snippet 2: index.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>XPath Examples</title>
</head>
<body>
    <h2>XPath Examples</h2>
     <div>Add your query here<div>
     <div ><textarea id="xPath"></textarea><div>
    <button onclick="example1()">Execute XPath
Expressions</button>
```

```
    <div id="xpath-result"></div>

    <script>
        function executeXPath(xpathExpression) {

                fetch('bookstore.xml')
    .then(response => {
             return response.text();
    })
    .then(xmlText => {
        // Parse the XML data
        const parser = new DOMParser();
        const xmlDoc = parser.parseFromString(xmlText,
'text/xml');

        // Rest of your code

          const result = document.evaluate(
                xpathExpression,
                xmlDoc,
                null,
                XPathResult.ANY_TYPE,
                null
           );

            let output = [];
            let node = result.iterateNext();
           while (node) {
                output.push(node.textContent);
                node = result.iterateNext();
            }

            const resultElement =
document.getElementById("xpath-result");
            resultElement.innerHTML =
`<p>${output.join(",")}</p>`;

                })

    .catch(error => {
        console.error('Error fetching XML file:', error);
    });


        }
        function example1() {
```

```
                var xpathexp =
document.getElementById("xPath").value;
            executeXPath(xpathexp);
        }
    </script>
</body>
</html>
```

To execute the different XPath expressions the user can use XAMPP tool.

X-operating system, Apache, Mysql, Php, Perl (XAMPP) is a software package that simplifies setting up a Web server environment on your computer. It includes components such as Apache (Web server), MySQL (database), PHP (scripting language), and more. XAMPP makes developing and testing Websites locally easier earlier to deploying them to a live server.

Following are the steps to install and start the local Web server:

1. Install XAMPP from this link: https://www.apachefriends.org/download.html.
2. Ensure that Apache Web Server is selected during installation.
3. Start XAMPP Control Panel.
4. Launch or start Apache. This starts the local Web server, as shown in Figure 3.1.

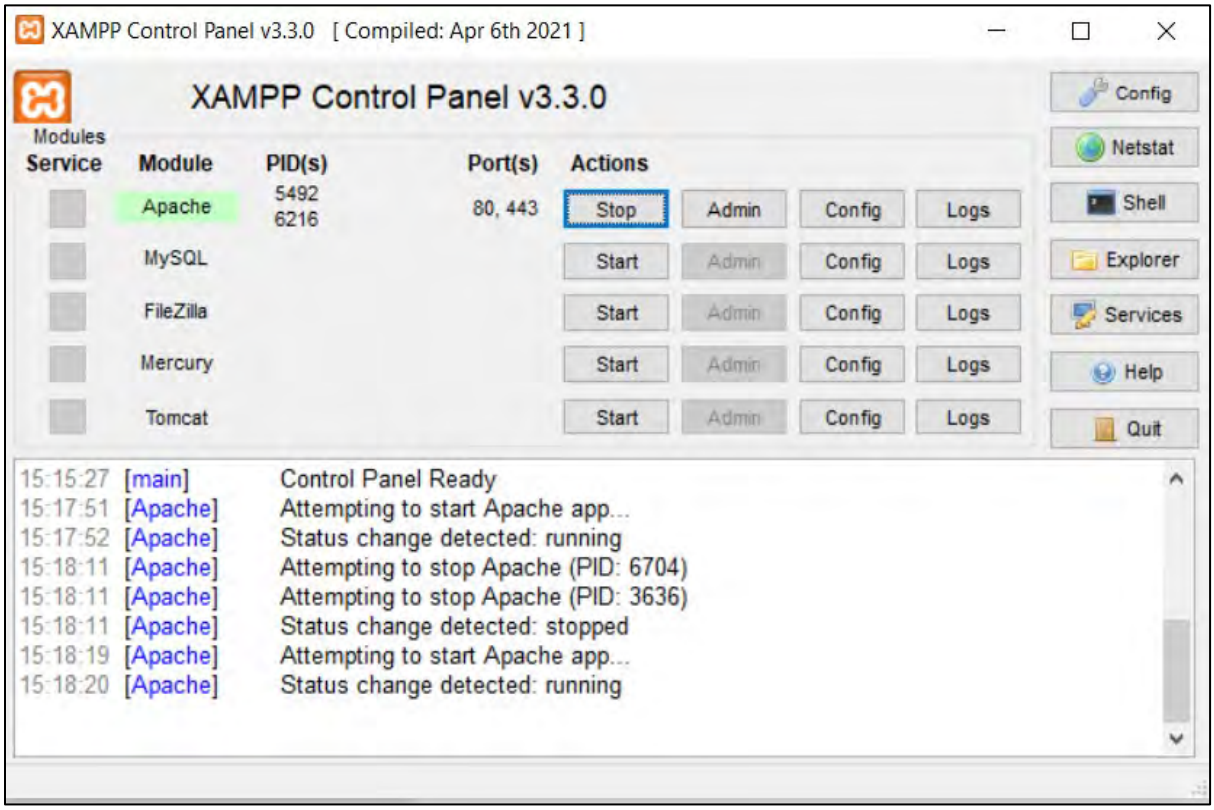


Figure 3.1: XAMPP Control Panel

Code Snippets 1 and 2 are saved in a folder inside the htdocs folder of XAMPP folder. The index.html can be executed using the URL, https://localhost/foldername/index.html.

The output is shown in Figure 3.2.

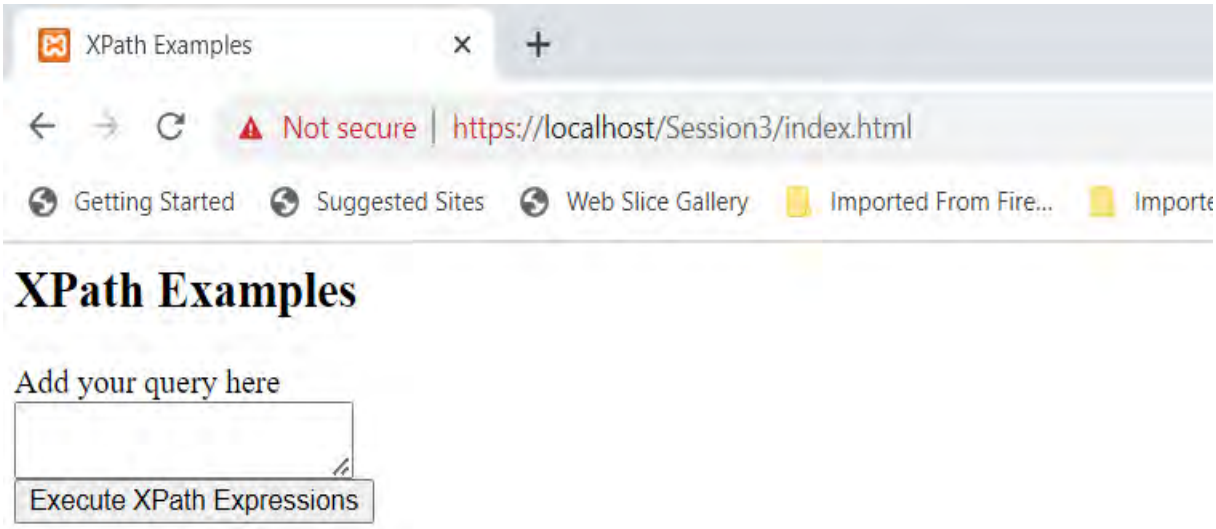


Figure 3.2: XPath Examples

This example can now be used to execute XPath expressions on `Bookstore.xml`.

Following are XPath expressions to perform some common operations:

The XPath Expression 1 selects all book titles:

XPath Expression 1:

```
/bookstore/book/title/text()
```

Paste the XPath in the textarea shown in Figure 3.3 and then, click XPath Expression Output to get the output. XPath Expression 2 displays the output as shown in Figure 3.3.

Figure 3.3: Output for XPath Expression 1

The XPath Expression 2 selects all books published before 1960:

XPath Expression 2

```
/bookstore/book[year < 1960]
```

XPath Expression 2 displays the output as shown in Figure 3.4.

Figure 3.4: Output of XPath Expression 2

The XPath Expression 3 select the price of a specific book:

XPath Expression 3:

```
/bookstore/book[title = 'To Kill a Mockingbird']/price/text()
```

XPath Expression 3 displays the output as shown in Figure 3.5.

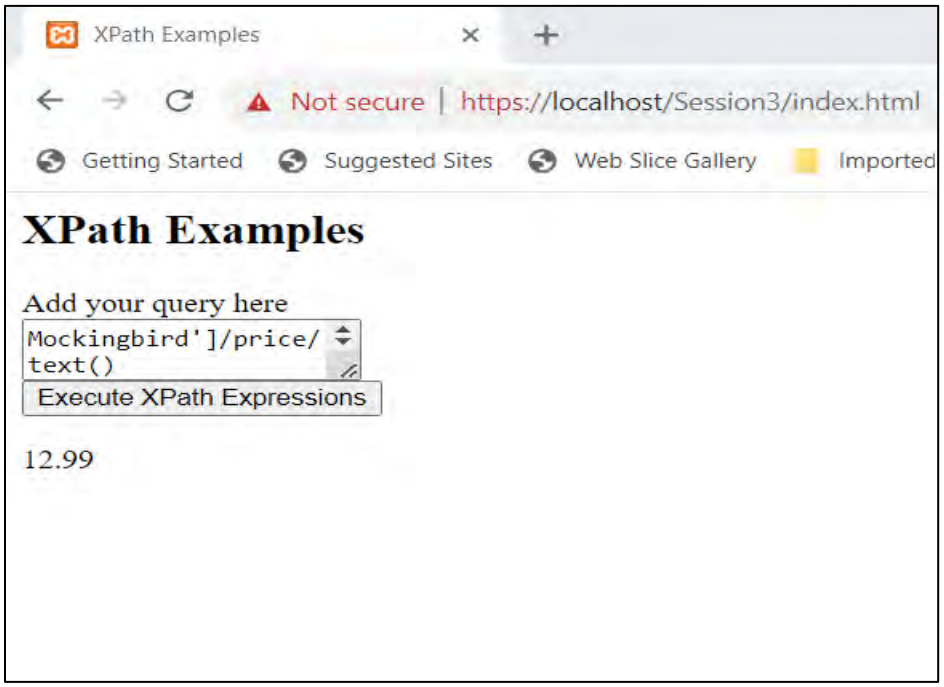


Figure 3.5: Output of XPath Expression 3

These are just a few illustrations of how well XPath can pick and work with items in an XML document. With its wide range of features and capabilities, XPath enables users of different programming languages and XML-related technologies to carry out sophisticated operations on XML data.

3.3 XPath Nodes

Multiple types of nodes that can be selected or modified in an XML document are defined by XPath. These node kinds are utilized in XPath expressions to describe the kind of nodes to be chosen or worked on.

The primary node types in XPath are as follows:

Element nodes

- An XML document's elements are represented by element nodes. They contain other nodes or data and are the main constituents of an XML document. In an XPath expression, element names can be used to pick element nodes; for example, '/book' selects all the 'book'"elements in the XML document.

Attribute nodes

- In an XML document, attribute nodes indicate the attributes of an element. They include metadata or extra details about a piece. The '@' symbol and the attribute name can be used to select attribute nodes in an XPath expression; for example, 'book/@title' chooses the 'title' attribute of a 'book' element.

Text nodes

- In an XML document, text nodes represent an element's text content. They include the essential information kept in every piece. The 'text()' function or the 'node()' wildcard can be used to pick text nodes if the predicate filters for text nodes; for example, '/book/title/text()' chooses the text content of the 'title' element within a 'book' element.

Comment nodes

- In an XML document, comments are represented by comment nodes. Within an XML document, they are used to add annotations or illustrative remarks. The 'Comment()' function or the 'node()' wildcard can be used to pick comment nodes if the predicate filters for them; for example, '/book/comment()' chooses all the comment nodes included within the 'book' element.

Processing Instruction nodes

- In an XML document, processing instructions are represented by Processing Instruction (PI) nodes. They are employed to communicate instructions to programs that handle XML documents. With the 'Processing-instruction()' function or the 'node()' wildcard and a predicate that filters for processing instruction nodes, PI nodes can be chosen. For instance, '/book/processing-instruction()' selects all the PI nodes contained in the 'book' element.

Namespace nodes

- In an XML document, namespace nodes represent namespace declarations. They specify the namespaces utilized in the XML document and link them to nodes corresponding to elements or attributes. The 'namespace::' or 'namespace-uri()' functions can be used to select namespace nodes; for example, '/book/namespace::' chooses all namespace nodes included within the 'book' element.

To precisely navigate and manipulate XML data, XPath nodes specify the type of nodes chosen or operated upon in an XML document.

3.4 XPath Syntax

For expressing expressions to explore and choose elements or attributes in an XML document, XPath has a special syntax. The hierarchy of the elements or attributes in the XML document is represented by a sequence of path steps separated by forward slashes (/).

The XPath expression is as follows:

```
/bookstore/book[@category='fiction']
```

This expression picks out all the book components in the bookstore element with the attribute category set to 'fiction'.

XPath expressions can be concatenated and nested in XML documents to provide increasingly complicated queries and selections, enabling effective and adaptable XML data processing.

3.5 XPath Relative and Absolute Paths

Table 3.1 shows the difference between Relative Path and Absolute Path.

Relative Path	Absolute Path
A relative path concerning the currently selected context node is provided.	An absolute path originates at the root of the XML document.
The current context node is the node from which the XPath expression is evaluated. The forward slash (/), which designates the root of the XML document, does not begin relative pathways. Instead, they start with a node name, axis or wildcard and the path is determined using the context node that is currently in use.	The current context node, absolute paths begin with a forward slash (/), is followed by a node name, an axis, or a wildcard, and are resolved from the root of the XML document.
Relative paths are frequently employed when the user desires to indicate a path relative to the present location in the XML document.	Absolute paths in an XML document always specify the full path to the desired node, regardless of the user's location within the document.
For example, `bookstore/book[@category='fiction']` illustrates a relative path. Assuming that the current context node is the root bookstore element, this expression chooses all book elements within the bookstore element that contains the attribute category with a value of 'fiction'.	For example, `/bookstore/book[@category='fiction']` illustrates an absolute route. This expression, regardless of the current context node, chooses any 'book' elements found in the root 'bookstore' element of the XML document with the attribute category with the value 'fiction'.

Table 3.1: Difference between Relative Path and Absolute Path

3.6 XPath Operators

The XPath language offers a variety of operators that can be used in expressions to manipulate strings, conduct logical processes, and compare elements in XML data.

Following are a few of the XPath operators that are frequently utilized:

Comparison Operators:

- = : Equal to
- != or <> : Not equal to
- < : Less than
- <= : Less than or equal to
- > : Greater than
- >= : Greater than or equal to

Example: `/bookstore/book[@price > 11]` selects all the book elements that have a price attribute greater than 10.

Logical Operators:

- and: Logical AND
- or : Logical OR
- not : Logical NOT

Example: `/bookstore/book[@category='fiction' and @price < 20]` selects all the book elements that have a category attribute equal to 'fiction' and a price attribute less than 20.

String Operators:

- concat(string1, string2): Concatenates two strings
- starts-with(string1, string2): Checks if a string starts with another string
- ends-with(string1, string2): Checks if a string ends with another string
- contains(string1, string2): Checks if a string contains another string
- string-length(string): Returns the length of a string
- normalize-space (string): Removes leading and trailing whitespaces from a string and collapses multiple whitespaces into a single space

For instance, the expression `/bookstore/book[contains(@title, 'XML')]` picks out all the book components that have title attributes with the substring 'XML' in them.

These are a few of the more typical XPath operators.

An XML document can explore and choose components or attributes with path steps, wildcards, predicates, and functions.

3.7 Summary

- XPath defines multiple types of nodes that can be selected or modified in an XML document.
- XPath provides a clear and expressive syntax for traversing and querying XML documents.
- XPath expressions can determine the location of items or attributes within an XML document.
- The path steps that make up an XPath expression determine the hierarchy of the elements or attributes in the XML document.
- XPath enables programmers to carry out sophisticated operations on XML data.
- XPath has a special syntax for expressing expressions and choosing elements or attributes in an XML document.
- XPath uses relative and absolute route paths to navigate and choose components or attributes in an XML document.
- XPath contains various operators to manipulate strings, conduct logical processes, and compare elements in XML data.

3.8 Check Your Progress

1. The hierarchy of the elements or attributes in the XML document is represented by a sequence of path steps separated by _____.

A.	Colon (:)	C.	Backward Slash (\)
B.	Semicolon (;)	D.	Forward Slash (/)

2. Which language is used to pick out specific elements from an XML document?

A.	XPath	C.	XML DOM
B.	XML Reader	D.	XML schema

3. _____ determine where items or attributes should be located within an XML document.

A.	XSLT	C.	XML Axis
B.	XPath Expressions	D.	XML predicates

4. Which of the following is NOT a primary node type in XPath?

A.	Element nodes	C.	Comment nodes
B.	Attribute nodes	D.	Parsing nodes

5. What is used to describe the direction of navigation?

A.	Predicates	C.	Functions
B.	Axes	D.	Logical Operators

3.8.1 Answers for Check Your Progress

Question	Answer
1	D
2	A
3	B
4	D
5	B

Try It Yourself

Create and test XML and HTML codes that perform following tasks:
Task 1: Write XPath expressions to accomplish following tasks:

1. Select the author of the book titled '1984'.
2. Select the genre of the book with the author 'Lewis Carroll'.

Provide the XPath expression and the expected output/result for each task.

Task 2: Write XPath expressions using operators to accomplish following tasks using inventory.xml of Code Snippet 3:

1. Select all products with a price of less than 50.
2. Select the names of products with a quantity greater than 0.

Code Snippet 3: Inventory.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<inventory>
    <product>
        <name>Product A</name>
        <price>49</price>
        <quantity>100</quantity>
    </product>
    <product>
        <name>Product B</name>
        <price>35</price>
        <quantity>50</quantity>
    </product>
    <product>
        <name>Product C</name>
        <price>145</price>
        <quantity>75</quantity>
    </product>
    <product>
        <name>Product D</name>
        <price>99</price>
        <quantity>200</quantity>
    </product>
</inventory>
```

Session 4

XSLT

This session explains the characteristics and features of XSLT. It also describes different elements that are used in XSLT.

Objectives

- Describe XSLT and its various characteristics
- Explain the XSLT syntax
- List different XSLT Elements

4.1 Introduction to XSLT

A potent XML-based language called XSLT converts XML documents into other formats such as HTML, XHTML, or other XML formats. It is a member of the XML family of technologies frequently utilized for the presentation and manipulation of data.

The foundation of XSLT is applying templates to XML documents to convert them into the required output formats. These templates specify the procedures to change the input XML into the desired output structure. In XSLT, the programmer declares what to accomplish rather than describing how to achieve it, using a declarative approach.

An XSLT file can have an extension .xslt or .xsl. XSLT's primary characteristics are explained as follows:

1. **Template-based transformation:** XSLT uses templates to specify the rules for altering particular XML document nodes or elements. Templates can be applied to the entire document or only to a subset of its elements or nodes.
2. **XPath:** To indicate where nodes are located within an XML document, XSLT uses XML Path Language (XPath). The programmer can traverse and choose nodes in an XML document using the powerful and adaptable language XPath based on the element names, attributes, values, and other criteria.
3. **Built-in functions:** XSLT has a wide range of built-in functions that can be used to manipulate strings, do computations, manipulate dates and times, and more on XML data.

   Here are some commonly used built-in functions in XSLT:
   `string()`: Converts a value to a string.
   `number()`: Converts a value to a number.
   `concat()`: Combines two or more strings.
   `substring()`: Extracts a portion of a string.
   `substring-before()`: Returns the substring before a specified delimiter in a string.
   `substring-after()`: Returns the substring after a specified delimiter in a string.

`translate()`: Replaces characters in a string with other characters.
`count()`: Counts the number of nodes in a node-set.
`sum()`: Calculates the sum of numbers in a node-set.
`round()`: Rounds a number to the nearest integer.
`floor()`: Rounds a number down to the nearest integer.
`ceiling()`: Rounds a number up to the nearest integer.
`current()`: Returns the current node being processed.
`document()`: Loads an external XML document for processing.
`format-number()`: Formats a number as a string based on specified formatting parameters.
`key()`: Retrieves values from an index created using the xsl:key instruction.

These are just some of the commonly used built-in functions in XSLT. XSLT provides a rich set of functions to perform various operations on XML data during transformation and can be extended with user-defined functions and extensions in some XSLT processors.

4. **Conditional processing:** Using XSLT, the programmer can apply various templates or processing instructions following specific criteria, such as the values of element attributes or the existence of particular elements.
5. **Modularization:** Programmers can modularize transformations and reuse common logic across several documents using XSLT, which enables them to construct reusable templates and incorporate them in other stylesheets.
6. **Output formatting:** XSLT allows for control over how the output is formatted and presented, creating HTML or XHTML documents with specific styles, or converting XML data into formats such as CSV or JSON.

XSLT is a robust and adaptable language that can convert XML documents into other formats. It is widely used in many industries for activities including data integration, content publication, and data presentation on the Web.

4.2 XSLT Syntax

Here is an example for XSLT stylesheet that converts an XML document into an HTML table.

Code Snippet 1 is the input XML file and Code Snippet 2 is the stylesheet that will be applied to it.

Code Snippet 1: product.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="product.xsl" ?>
<products>
  <product>
    <name>Product 1</name>
    <price>10.99</price>
  </product>
  <product>
    <name>Product 2</name>
    <price>19.99</price>
  </product>
</products>
```

Code Snippet 2: product.xsl

```
<!-- Example XSLT stylesheet -->

<!-- Define the XML namespace for XSLT -->
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">

<!-- Define a template for the root element 'products' -->
<xsl:template match="products">
  <html>
    <body>
      <h1>Product List</h1>
      <table border="true">
        <tr>
          <th>Name</th>
          <th>Price</th>
        </tr>
        <!-- Apply the following template to each 'product'
element -->
        <xsl:apply-templates select="product"/>
      </table>
    </body>
  </html>
```

```
</xsl:template>

<!-- Define a template for 'product' elements -->
<xsl:template match="product">
  <tr>
    <td>
      <xsl:value-of select="name"/>
    </td>
    <td>
      <xsl:value-of select="price"/>
    </td>
  </tr>
</xsl:template>

</xsl:stylesheet>
```

To execute the transformation of XML with the XSLT effects using these two snippets, the programmer can use XAMPP tool. If the codes are executed directly from the local computer such as files from D drive, they may not work correctly. Hence, it is recommended to run them through a local Web server.

X-operating system, Apache, MySQL, PHP, and Perl (XAMPP) is a software package that simplifies setting up a Web server environment on your computer. It includes components such as Apache (Web server), MySQL (database), PHP (scripting language), and so on. In short, XAMPP is a local Web server platform package.

XAMPP makes developing and testing Websites locally easier to deploying them to a live server.

Following are the steps to install and start the local Web server:

1. Install XAMPP from this link: https://www.apachefriends.org/download.html.
2. Ensure that Apache Web Server is selected during installation.
3. Start XAMPP Control Panel.
4. Launch or start Apache. This starts the local Web server, as shown in Figure 4.1.

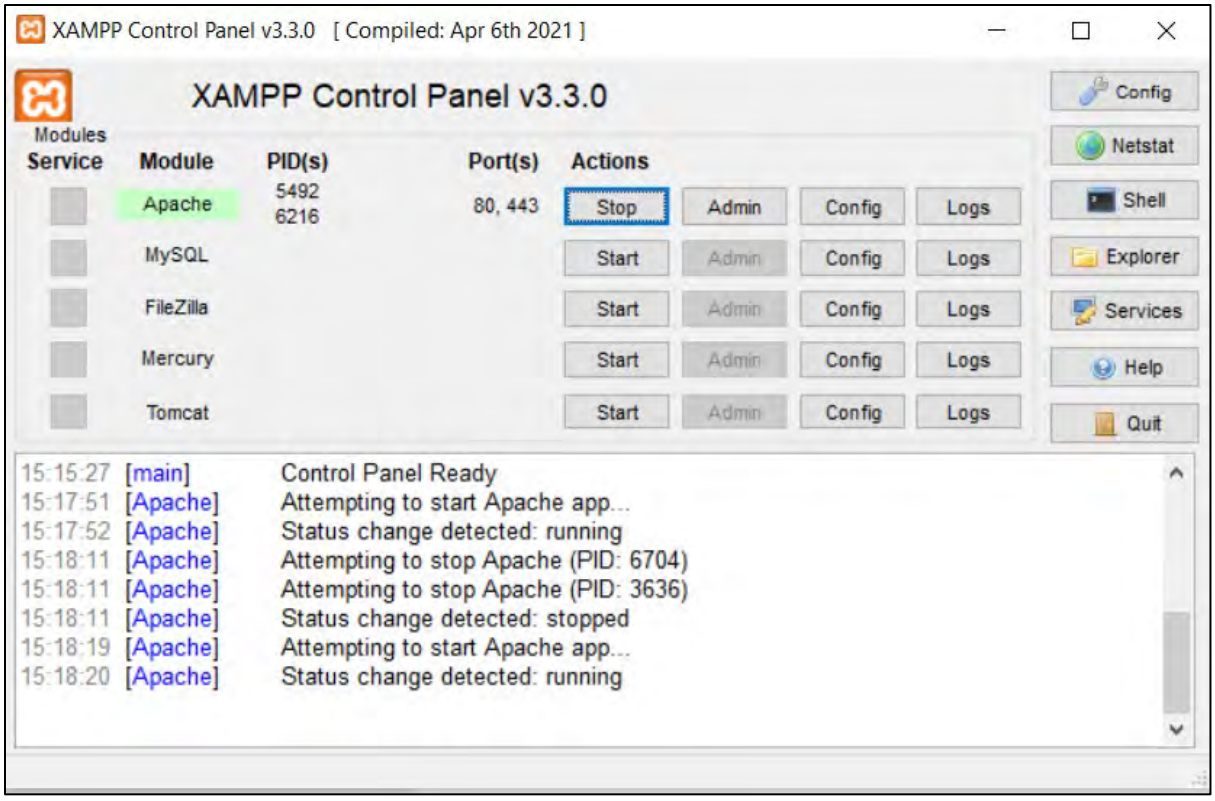


Figure 4.1: XAMPP Control Panel

5. Go to the htdocs folder under XAMPP and save the code in this folder path (Session4 Folder has `product.xml` and `product.xsl`). Refer to Figure 4.2.

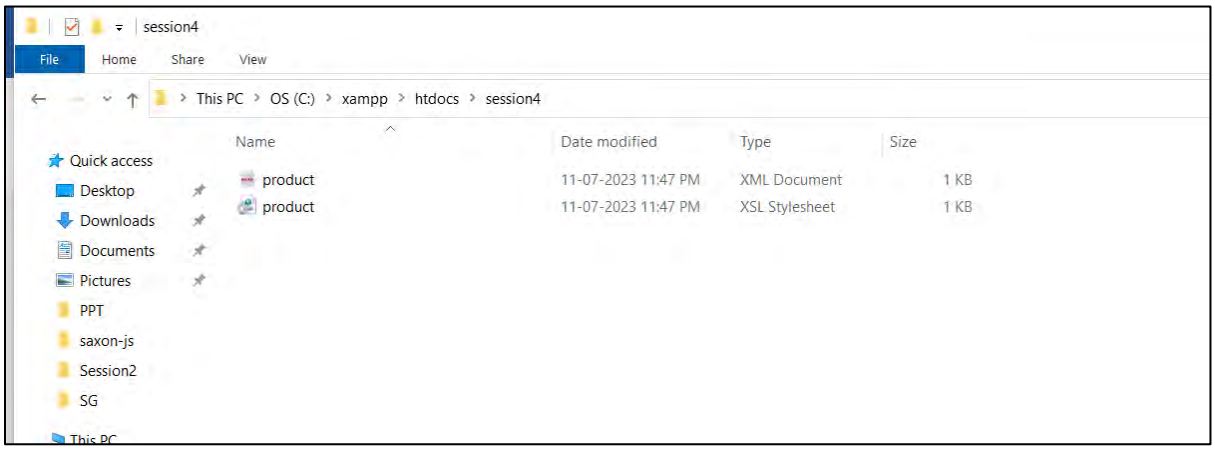


Figure 4.2: htdocs Folder

6. Open browser and type the path: https://localhost/session4/product.xml; the programmer can see the XML data with styles applied as shown in Figure 4.3.

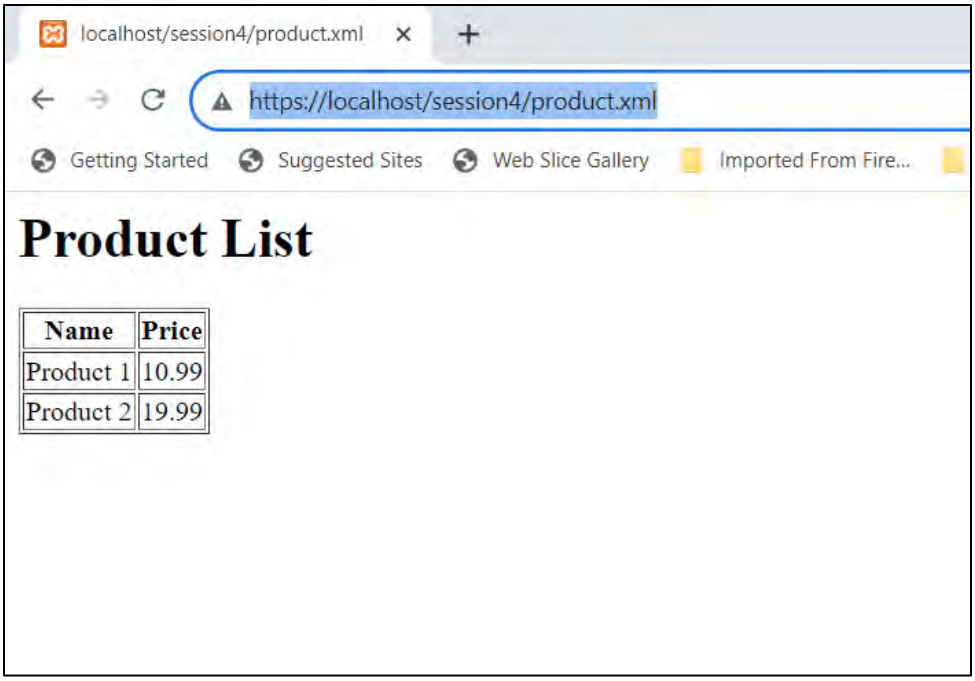


Figure 4.3: Output of product.xml

The output shown in Figure 4.3 is just a simple example of XSLT's powerful and complicated features. XSLT includes its syntactic rules and ideas for more complex transformations, including modes, variables, and conditional processing.

In Code Snippet 2, an XSLT stylesheet with two templates is defined: one for the `root` element `products` and another for individual product components. The `match` attribute identifies the XML element or elements to which the template should be applied, while the `<xsl:template>` element defines a template. To convert the input XML into an HTML table, XSLT elements and methods such as `<xsl:apply-templates>`, `<xsl:value-of>` and `<xsl:select>` inside the templates are utilized.

The output of the transformation will be an HTML page (shown in Code Snippet 3) with a table-formatted list of products along with their names and pricing.

Code Snippet 3: HTML Output

```
<html>
  <body>
    <h1>Product List</h1>
    <table border=true>
      <tr>
        <th>Name</th>
```

```
        <th>Price</th>
      </tr>
      <tr>
        <td>Product 1</td>
        <td>10.99</td>
      </tr>
      <tr>
        <td>Product 2</td>
        <td>19.99</td>
      </tr>
    </table>
  </body>
</html>
```

In order to retrieve this HTML code, right-click the Web page and then, select **Inspect** as shown in Figure 4.4.

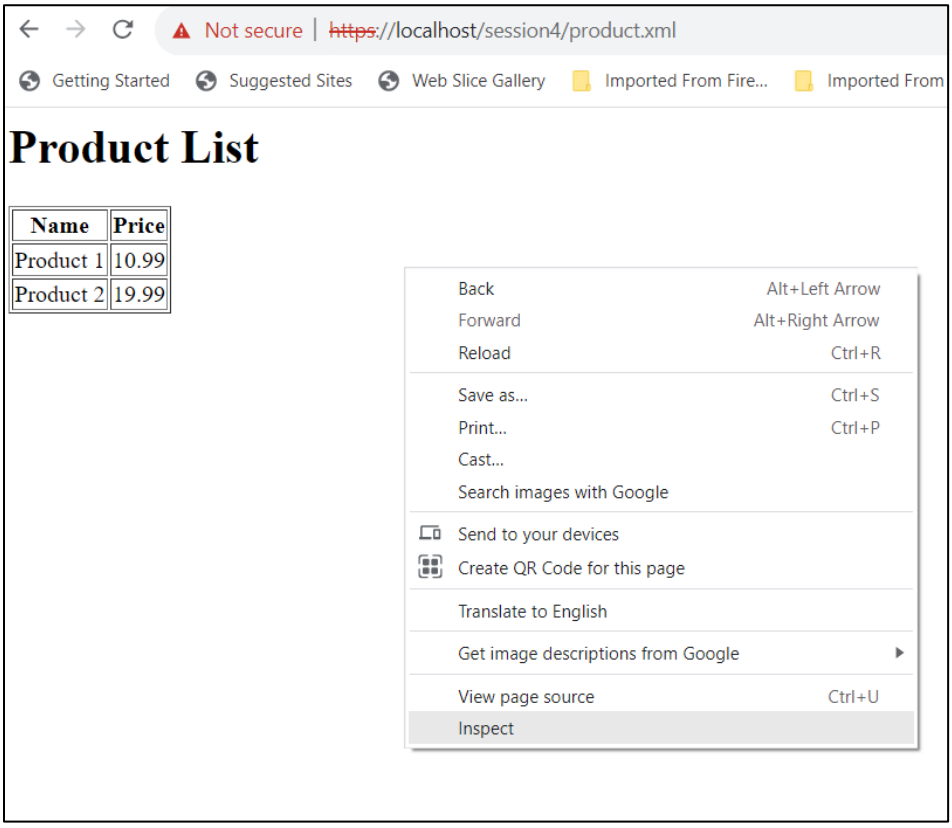


Figure 4.4: Inspect Option

The HTML code can be seen in the **Elements** tab, as shown in Figure 4.5.

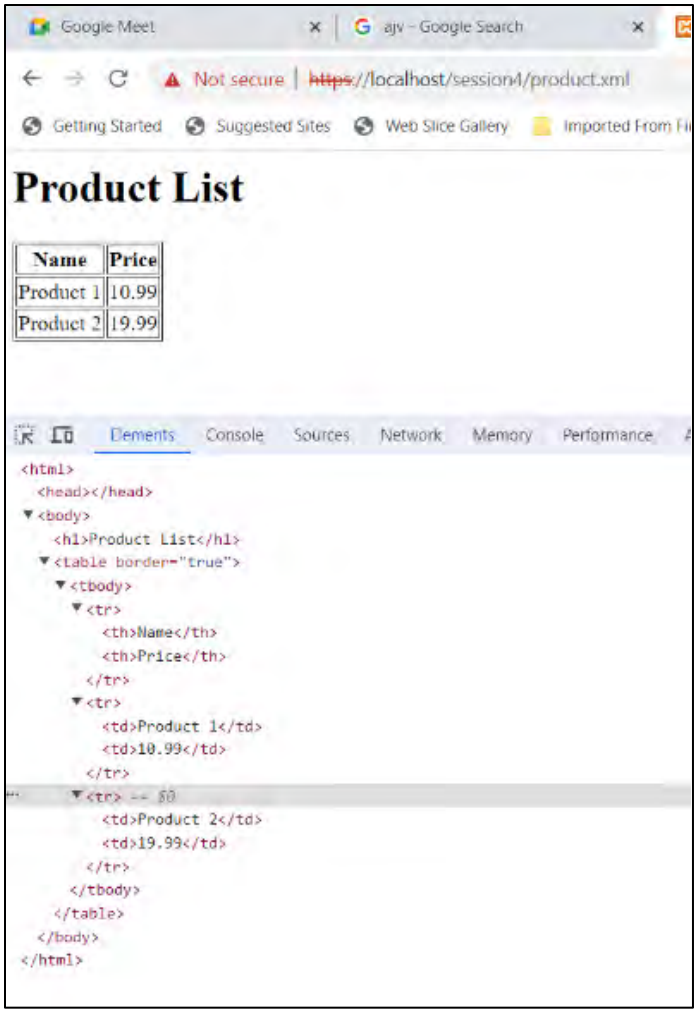


Figure 4.5: Elements Tab

Knowledge Check 1

1. XSLT converts XML documents into other formats such as HTML. (True/False)
2. Templates can be applied to only part of the document. (True/False)

4.3 Different XSLT Elements

The XML elements and attributes that make up the XSLT syntax are used to construct templates, describe rules, and work with XML data.

An explanation of the syntax of certain frequently used XSLT elements are as follows:

- <xsl:stylesheet>: An XSLT stylesheet's root element defines the namespace for XSLT and declares the XSLT version being utilized. It also includes other top-level elements that specify the rules and variables used in the transformation.
  Example:

```
<xsl:stylesheet  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">

  <!-- Other XSLT elements go here -->

</xsl:stylesheet>
```

- <xsl:template>: This element defines a transformation template for an XML element or set of elements. It contains additional XSLT elements and functions that provide the transformation rules and a match property that identifies the XML element or elements to which the template should be applied.

  Example:

```
<xsl:template match="products">

  <!-- Transformation rules for 'products' element go here -->

</xsl:template>
```

- <xsl:apply-templates>: Select XML elements that can have templates applied to them using this element. The XML document can apply templates to child elements, siblings, or any other desired location. It features a select attribute, which is optional, that allows the programmer to specify which XML elements should receive the templates.
  Example:

```
<xsl:apply-templates select="product"/>
```

- <xsl:value-of>: With the help of this element, the value of an XML element or attribute can be extracted and added to the output. It has a select attribute that designates the XML element or attribute from which the value should be obtained.
  Example:

```
<xsl:value-of select="name"/>
```

- <xsl:for-each>: This element is used to repeatedly go over a group of XML elements and apply templates or carry out further actions on each one. It contains various XSLT elements and functions that define the operations to be carried out and it has a select property that specifies the XML elements to be processed.

  Example:

```
<xsl:for-each select="product">

  <!-- Transformation rules for 'product' element go here -->

</xsl:for-each>
```

- <xsl:if> and <xsl:select>: These components are utilized for conditional processing in XSLT. In contrast to <xsl:if>, which applies a template or executes an operation based on a condition, <xsl:select> lets the programmer specify many conditions and select which one to apply.

  Example:

```
<xsl:if test="price > 50">
  <!-- Transformation rules for products with price > 50 go here
-->
</xsl:if>

<xsl:select>
  <xsl:when test="condition1">
    <!-- Transformation rules for condition1 go here -->
  </xsl:when>
  <xsl:when test="condition2">
    <!-- Transformation rules for condition2 go here -->
  </xsl:when>
  <xsl:otherwise>
    <!-- Transformation rules for other conditions go here -->
  </xsl:otherwise>
</xsl:select>
```

- <xsl:variable>: In XSLT, a variable that can store a value or an expression is defined using this element. In the stylesheet, variables are used to simplify complex expressions or hold interim results.

```
<xsl:variable name="varName" select="expression"/>
```

These are some of the fundamental building blocks of the XSLT syntax. However, XSLT also provides additional building blocks and functions for managing namespaces, handling text and whitespace, sorting, grouping, and other operations.

Here is an example of XSLT transformation which illustrates few elements of XSLT.

Code Snippet 4 is a XML file and Code Snippet 5 is the XSLT file used inside XML file.

Code Snippet 4: product2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="groupingproducts.xsl" ?>
<products>
  <product>
    <name>Product 1</name>
    <price>178</price>
  </product>
  <product>
    <name>Product 2</name>
    <price>109</price>
  </product>
  <product>
    <name>Product 3</name>
    <price>10</price>
  </product>
  <product>
    <name>Product 4</name>
    <price>199</price>
  </product>
    <product>
    <name>Product 5</name>
    <price>40</price>
  </product>
</products>
```

Code Snippet 5: groupingproducts.xsl

```
<!-- Example XSLT stylesheet -->
<!-- Define the XML namespace for XSLT -->
<xsl:stylesheet  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">

<!-- Define a template for the root element 'products' -->
<xsl:template match="/">
  <html>
    <body>
```

```
      <h1>Product List</h1>
      <table border="true">
       <caption>Expensive Products</caption>
        <tr>
          <th>Name</th>
          <th>Price</th>
        </tr>
        <!-- Apply the following template to each 'product'
element -->
            <xsl:for-each select="products/product[price &gt;
50]">
                <tr>
                    <td><xsl:value-of select="name"/></td>
                    <td><xsl:value-of select="price"/></td>
                </tr>
            </xsl:for-each>
                    </table>

                     <br>
                     </br>
    <table border="true">
            <caption>Affordable Products</caption>
        <tr>
          <th>Name</th>
          <th>Price</th>
        </tr>

      <!-- Filter and display affordable products -->
     <xsl:for-each select="products/product[price &lt;= 50]">
                <tr>
                    <td><xsl:value-of select="name"/></td>
                    <td><xsl:value-of select="price"/></td>
                </tr>
            </xsl:for-each>
                 </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

This code is an XSLT stylesheet used to transform XML data into HTML. Here is a technical explanation of the code:

1. `<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"`

`version="1.0">`: This line defines the XSLT stylesheet by specifying the XML namespace for XSLT and the version.

2. `<xsl:template match="/">`: This line defines a template for the root element of the XML data. In this case, it matches the root element, typically named 'products'. Inside this template, HTML elements are generated using XSLT instructions. For example:
   - `<html>`, `<body>`, and `<h1>` are standard HTML elements.
   - `<table>` elements are used to create tables.
   - `<xsl:for-each>` is used to loop through XML elements (in this case, 'product' elements within 'products') and apply following template to each of them.

3. `<xsl:for-each select="products/product[price &gt; 50]">`: This line filters 'product' elements with a 'price' attribute greater than 50 and processes them within the loop.

4. `<xsl:value-of select="name"/>` and `<xsl:value-of select="price"/>`: These lines extract and display the 'name' and 'price' attributes of each selected 'product' element, adding them as table data cells.

5. The code also generates another table for ‘Affordable Products’ using a similar structure, but this time, it filters 'product' elements with a 'price' attribute less than or equal to 50.

6. `<xsl:template>`: These are closing tags and the root template define the structure of the generated HTML page.

Overall, this XSLT stylesheet takes XML data containing 'products' and their attributes, filters them based on price, and then, generates an HTML page with two tables: one for expensive products and another for affordable products. The output is shown in Figure 4.6.

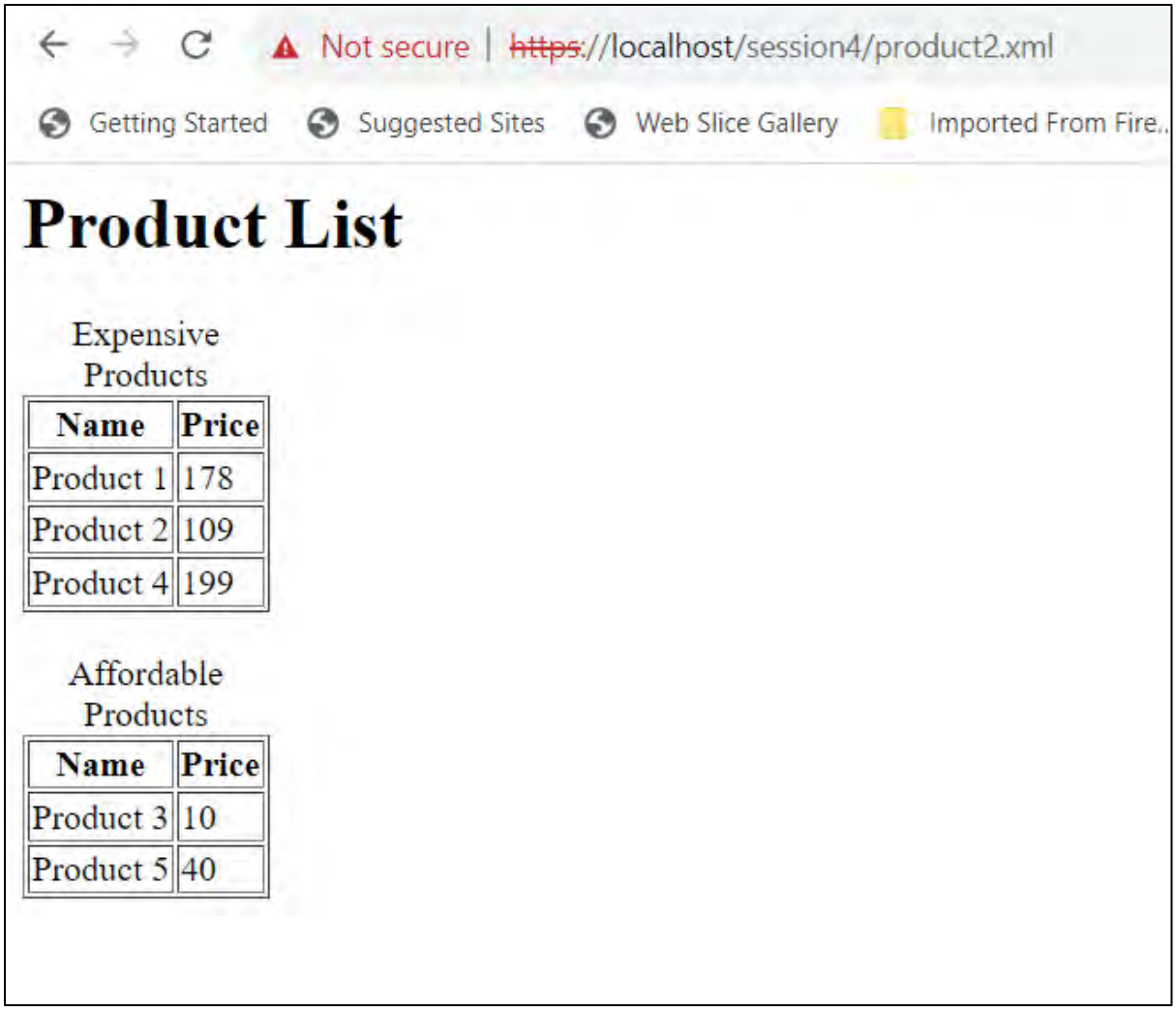


Figure 4.6: Output of product2.xml

To view the HTML code as a result of the transformation, right-click the Web page and then, select **Inspect**. The programmer can view the HTML code for the page.

4.4 Summary

- XSLT converts XML documents into other formats such as HTML, XHTML, or other XML formats.
- A member of the XML family of technologies frequently utilized for the presentation and manipulation of data.
- XSLT applies templates to XML documents to convert them into the required output formats.
- The templates are used to specify the rules for altering XML document nodes or elements.
- XSLT is widely used in many different industries for activities such as data integration, content publication and data presentation on the Web.
- XSLT uses Templates, XPath, built-in functions, conditional processing, modularization, and output formatting.
- XSLT provides building blocks and functions for managing namespaces, handling text, grouping, and other operations.

4.5 Check Your Progress

1. To indicate where nodes are located within an XML document, XSLT uses _____.

A.	XPath	C.	Variables
B.	Labels	D.	Pointers

2. XSLT has a wide range of built-in _____ that can be used to manipulate XML data.

A.	steps	C.	functions
B.	commands	D.	tags

3. XSLT allows for control over how the _____ is formatted and presented.

A.	input	C.	message
B.	output	D.	files

4. The _____ element defines a template.

A.	<xsl:select>	C.	<xsl:template>
B.	<xsl:value-of>	D.	<xsl:for-each>

5. _____ are used to simplify complex expressions or hold interim results.

A.	<xsl:select>	C.	<xsl:template>
B.	<xsl:value-of>	D.	<xsl:for-each>

4.5.1 Answers for Check Your Progress

Question	Answer
1	A
2	B
3	C
4	B
5	C

Answers for Knowledge Check 1:

1) True
2) False

Try It Yourself

Task 1: Create an XSLT file for following XML data. The XML file when viewed in a browser should render HTML output.

Input XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="fiction">
    <title>Harry Potter and the Sorcerer's Stone</title>
    <author>J.K. Rowling</author>
    <year>1997</year>
  </book>
  <book category="fiction">
    <title>The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
    <year>1925</year>
  </book>
  <book category="non-fiction">
    <title>The Elements of Style</title>
    <author>William Strunk Jr.</author>
    <year>1918</year>
  </book>
</bookstore>
```

Output HTML:

```
<html>
  <body>
    <h2>Bookstore Catalog</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Author</th>
        <th>Year</th>
      </tr>
      <tr>
        <td>Harry Potter and the Sorcerer's Stone</td>
        <td>J.K. Rowling</td>
        <td>1997</td>
      </tr>
      <tr>
        <td>The Great Gatsby</td>
```

```
        <td>F. Scott Fitzgerald</td>
        <td>1925</td>
      </tr>
      <tr>
        <td>The Elements of Style</td>
        <td>William Strunk Jr.</td>
        <td>1918</td>
      </tr>
    </table>
</body>
</html>
```

Task 2: Create a new XSLT to display the books created before 1920 and after 1920 in separate tables. Apply it to the XML and document the output.

Session 5

JSON

This session defines JSON and describes its features. It also explains objects and array in JSON and describes nested JSON.

Objectives

- Describe JSON and explain its various features
- List the differences between JSON and XML
- Explain the JSON object and array
- Explain nested JSON

5.1 Introduction

JavaScript Object Notation (JSON) is a simple, text-based, lightweight data transfer standard. It is frequently used to transfer data between clients, servers, or various application components. JSON can be used with almost any programming language because it is language-independent and the de facto standard for data exchange on the Web.

The basic syntax of JSON is built on key-value pairs, where keys are strings and values can be strings, numbers, booleans, arrays, or other JSON objects. JSON employs a few fundamental data types:

- **Strings:** Collections of characters that are encased in double quotes. For instance, "Hello, world!"
- **Numbers:** Floating-point or integer values. 42 or 3.14, for instance.
- **Booleans:** Defined as "true" or "false" statements.
- **Arrays:** Groups of items that are arranged in a particular order and are delimited by commas and square brackets ([ ]). as in [1, 2, 3].
- **Objects:** Unordered groups of key-value pairs delimited by commas and surrounded by curly braces (). For instance, "John's name" and "30".

JSON is compact, legible by humans, and straightforward to interpret; hence, it is frequently used in online applications to transfer data from a server to a client. JSON is frequently used as a data format in databases and data storage and for configuration files and APIs.

Here is an example of a JSON object that shows the details of a person.

Code Snippet 1: person.json

```
{
  "name": "John Doe",
  "age": 30,
  "gender": "male",
  "isStudent": false,
  "courses": ["math", "history", "chemistry"],
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  }
}
```

Code Snippet 1 shows a JSON object with several key-value pairs. The keys are strings (for example, "name", "age", "gender", "isStudent", "courses", and "address"). Values can be text (for example, "John Doe", "male"), numbers (for example, 30), and booleans (for example, false). Values can also include arrays (for example, "courses" which contains elements ["math", "history"]) or nested JSON objects (for example, "address").

The data in the example represents personal information such as a person's name, age, gender, student status, courses taken, and address. The keys indicate the many characteristics of the person, while the values include the appropriate data. Strings are enclosed in double quotation marks (""), while numbers, booleans, arrays, and nested objects are not enclosed in the double quotation marks ("").

5.1.1 Features of JSON

JSON is widely used for data transmission because of various characteristics:

- **Simplicity:** JSON has a straightforward syntax that is simple to read and write. It employs a well-known key-value pair format, similar to how objects are represented in many computer languages.
- **Data Storage:** JSON is a lightweight data interchange format that does not require much overhead. It is text-based and succinct, effectively transmitting data via networks and reducing data storage requirements.

- **Language Independence:** JSON is language-independent and can be utilized with almost any programming language. Most computer languages provide JSON parsers and serializers, making encoding and decoding JSON data simple.
- **Human-Readable:** JSON is intended to be human-readable and straightforward to grasp, making it helpful for both developers and non-programmers. It is commonly used in configuration files, APIs, and other situations requiring human-readable data representation.
- **Supports sophisticated Data Structures:** JSON allows for hierarchical data representation by supporting sophisticated data structures such as nested objects and arrays. This makes it flexible enough to accommodate a variety of data types and data interactions.
- **Widely Accepted:** JSON has become the de facto standard for Web data transmission and it is supported by practically all modern computer languages and platforms. Due to its wide acceptability, JSON can be easily used for data transmission in various circumstances, including Internet applications, mobile apps, and other software systems.
- **Extensibility:** JSON supports custom data types and characteristics, making it adaptable to various use scenarios. JSON Schema, a JSON-based schema validation standard, adds a layer of extensibility and data validation by defining and validating the structure and integrity of JSON data.

Knowledge Check 1:

1) JSON is language-independent and can be used with almost any programming language. (True/False)
2) JSON values can only be of String type. (True/False)

5.2 JSON vs. XML

JSON and XML are both commonly used for representing and transferring structured data over a network, typically between a server and Web applications. However, JSON is more straightforward, concise, and easier to scan and process, whereas XML is more versatile, extendable, and frequently used in legacy systems. The application's specific requirements determine the decision between JSON and XML, the environment in which it runs, and the preferences of the developers working with the data.

Differences between JSON and XML are listed in Table 5.1.

JSON	XML
JSON is easy to learn.	Compared to JSON, XML is more difficult to learn.
Both reading and writing are simple.	Reading and writing it is more difficult than with JSON.
It is data-oriented.	It is document-oriented.
Compared to XML, JSON is less secure.	XML is quite secure.
There are no display capabilities.	It gives the display functionality because it is a markup language.
The array type is supported.	The array type is not supported.

Example of JSON data:
[
{
"name" : "Peter",
"employeeid" : "E231",
"present" : true,
"numberofdayspresent" : 29
},
{
"name" : "Jhon",
"employeeid" : "E331",
"present" : true,
"numberofdayspresent" : 27
}
]
	Example of XML data:
	<name>
	<name>Peter</name>
	</name>

Table 5.1: Differences between JSON and XML

5.3 JSON Object

A JSON object is a collection of key-value pairs, where each pair's value can be one of a string, number, boolean value, or another JSON object. Curly braces -{} surround JSON objects and commas separate key-value pairs. The key-value pairs have the format "key": value, with the key being a string surrounded by double quotes and the value is any acceptable JSON data type.

Here is an example of a JSON object containing information on a person.

Code Snippet 2:

```
{
  "name": "John",
  "age": 30,
```

```
  "isStudent": false,
  "courses": ["math", "history", "chemistry"],
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "state": "NY",
    "zip": "12345"
  }
}
```

The JSON object in Code Snippet 2 contains five key-value pairs:

• "name": "John": "name" is the key and "John" is the value.

• "age": 30: The key is "age," and the value is 30.

• "isStudent": false: "isStudent" is the key and the value is the boolean false.

• "courses": ["math", "history", "chemistry"]: "courses" is the key and the value is a string array.

• "address":...: "address" is the key and the value is another JSON object representing an address.

JSON objects can also be nested, allowing for the creation of sophisticated data structures and hierarchies. The "address" key in the preceding example has a nested JSON object as its value, representing the address information.

JSON objects are frequently used contexts where a lightweight and human-readable data format is required. Modern programming languages and platforms support them via built-in or third-party libraries, making them popular for representing and sharing data in various applications. JSON objects are simple to serialize into strings for network transmission or file storage. They can be de-serialized back into native data structures in programming languages for processing, making them a versatile and frequently used data format.

5.4 JSON Array

An array is a set of values arranged in a certain sequence and enclosed in square brackets ([]). Arrays serve as ordered collections of values, where these values can take on any valid JSON data type, including texts, numbers, booleans, arrays, and JSON objects.

Following is a sample JSON array:

Code Snippet 3:

```
[
  "apple",
  "banana",
  "cherry",
  "date"
]
```

The JSON array in Code Snippet 3 represents a list of fruits. The array has four string values: "apple", "banana", "cherry", and "date" which are ordered by position in the array.

The first element of an array in JSON is at index zero; the second element is at index one and so on. Arrays are indexed starting at zero. Arrays can have any number of elements and elements of various data types. Arrays can also be layered within other arrays or JSON objects to create more complicated data structures.

Arrays in JSON help represent lists of values, such as products in a shopping cart, comments on a blog post, or workers in a company. They are also frequently used in Web APIs to represent data sets or to provide multiple values for a single key.

Here is an example of a JSON array with a variety of data kinds:

Code Snippet 4:

```
[
  "apple",
  42,
  true,
  [1, 2, 3],
  {
    "name": "John",
    "age": 30
  }
]
```

The array in Code Snippet 4 contains a string ("apple"), a number (42), a boolean (true), another array ([1, 2, 3]), and a JSON object ("name": "John", "age": 30). Arrays are adaptable and can be used to represent a wide range of data structures in JSON.

5.4.1 JSON Multidimensional Array

In JSON, there is not a dedicated data structure called a 'multidimensional array'. However, programmers can represent multidimensional data by utilizing nested arrays. By incorporating arrays within arrays, they can create a structure resembling multidimensional arrays.

Following is an illustration of a JSON arrangement that signifies a two-dimensional array:

```
[[1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]]
```

5.5 Nested JSON

A JSON object that has one or more JSON objects as values for its attributes is referred to as nested JSON. Here is an illustration of a nested JSON object:

Code Snippet 5: persondetails.json

```
{
  "name": "John",
  "age": 30,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "555-1234"
    },
    {
      "type": "work",
      "number": "555-5678"
    }
  ]
```

```
}
```

The JSON object in Code Snippet 5 has four properties: name, age, address and phoneNumbers. The address property is a JSON object with four properties: street, city, state, and zip code. The phoneNumbers field is a collection of JSON objects, each with two properties: type and number.

5.6 Working with JSON

JSON files can be created using any text editor or IDE. Apache NetBeans is a versatile Integrated Development Environment (IDE) that supports creation of Java, HTML, XML, and JSON files.

Following are the steps to create a JSON file using NetBeans:

1. To create a JSON file, click **File** -> **New File** as shown in Figure 5.1.

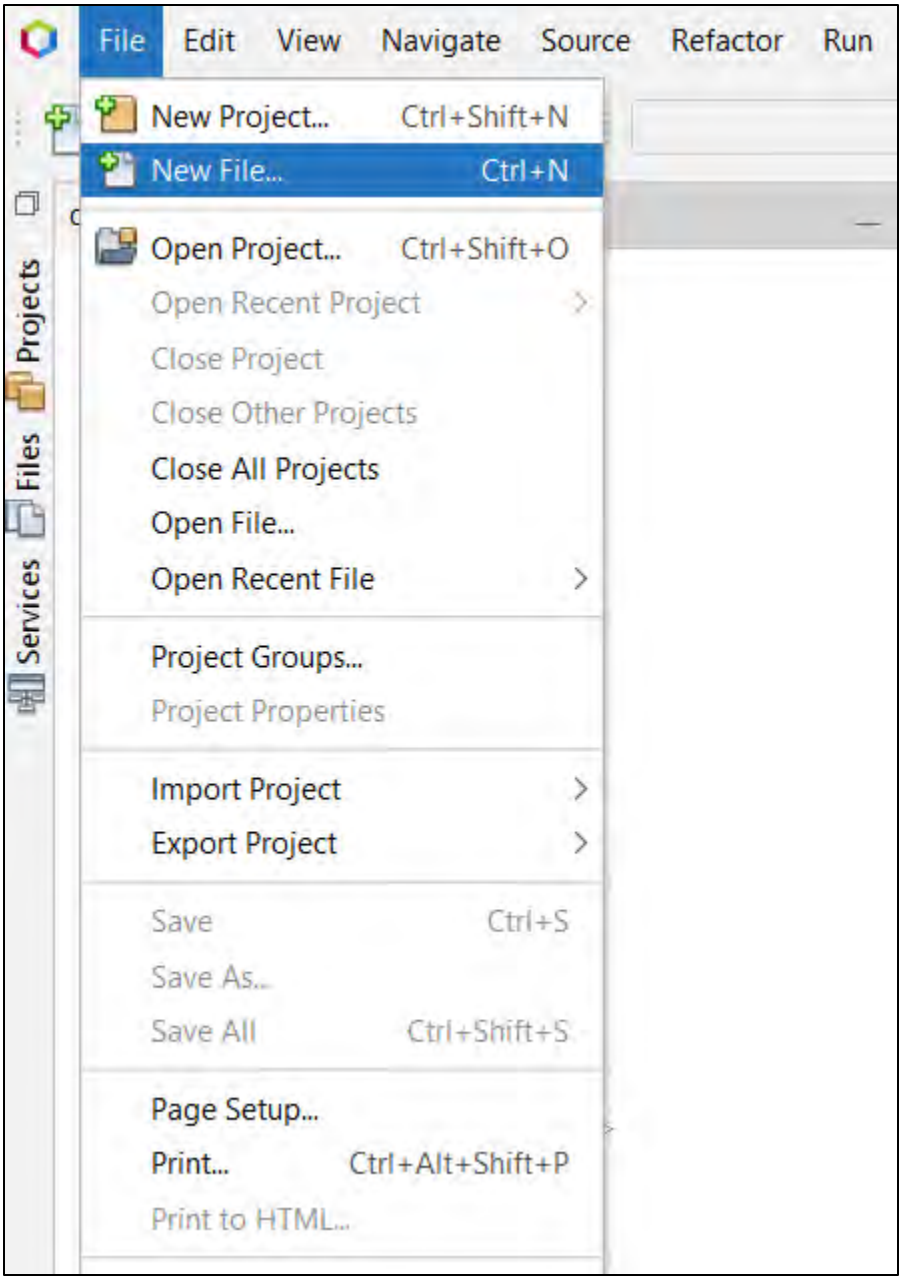


Figure 5.1: New File Option

2. In the **New File** dialog box, select **HTML5/JavaScript**, **JSON File** and click **Next**, as shown in Figure 5.2.

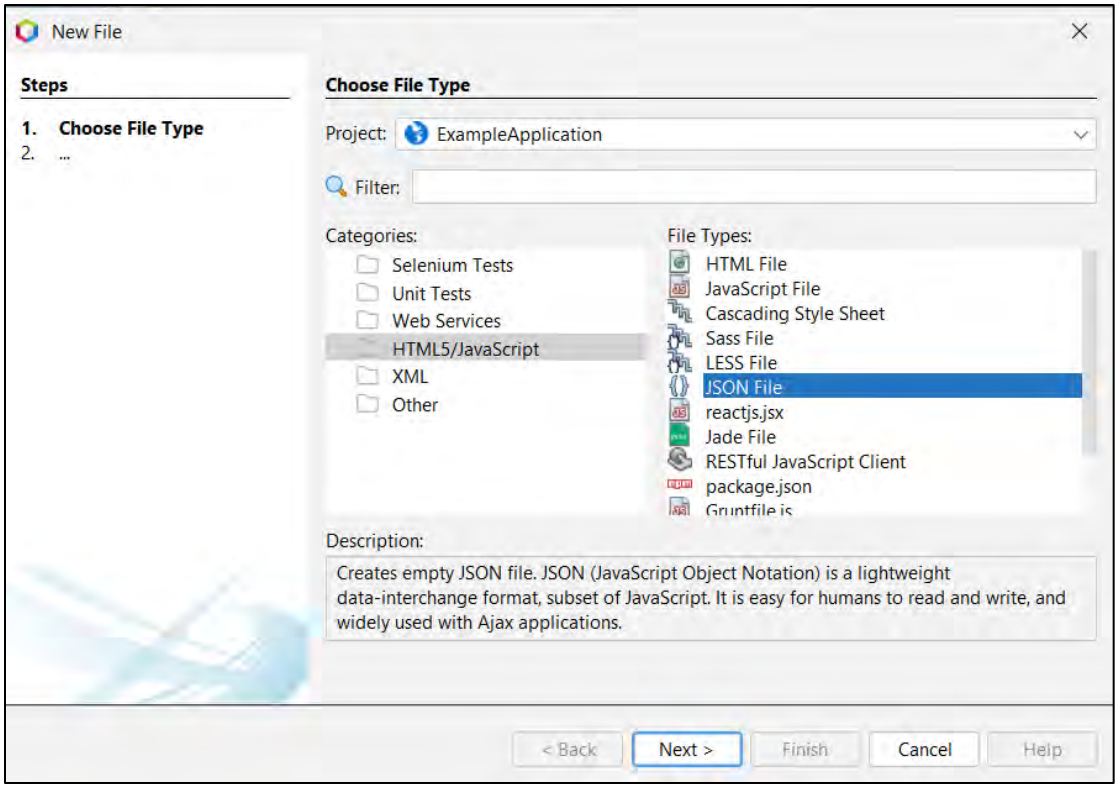


Figure 5.2: New File Dialog Box

3. Specify a name and browse for the location where the file should be saved, as shown in Figure 5.3. Click **Finish**.

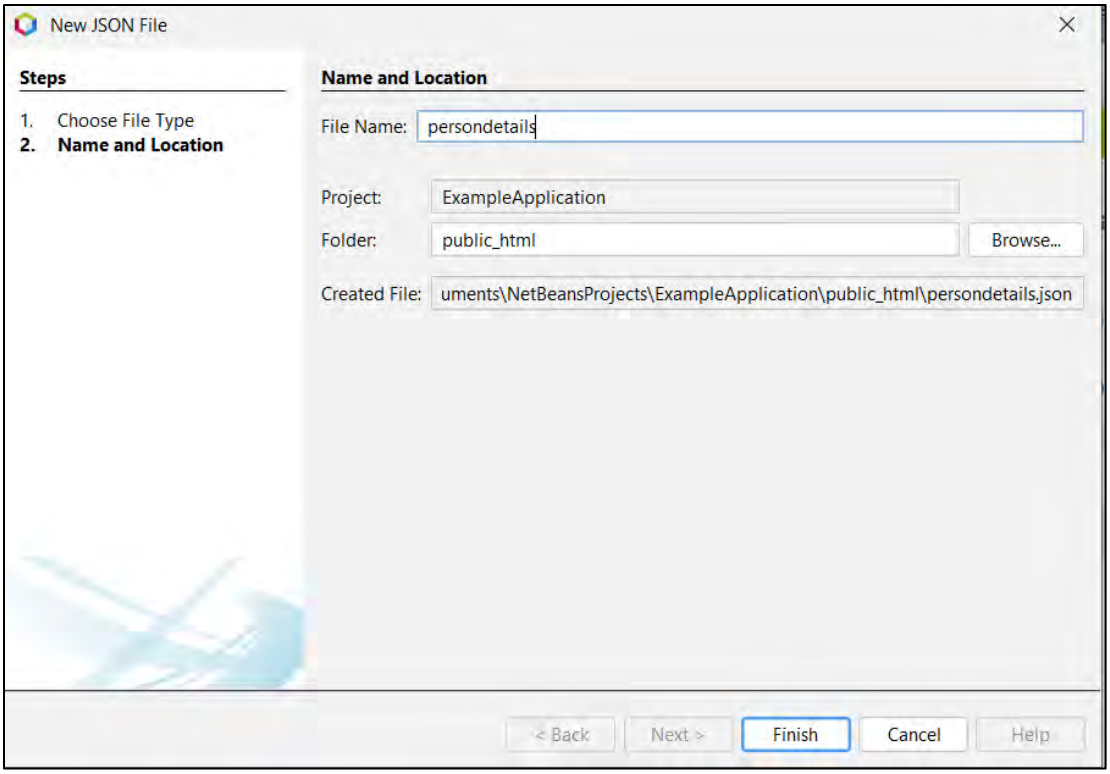


Figure 5.3: Name and Location Tab

A file will be auto-generated with default data as shown in Figure 5.4.

Figure 5.4: Generated JSON File

The programmer can now copy the data given in Code Snippet 5 and paste it in the generated file and save it.

Programmers can also open an already created file in NetBeans. To open an existing file, click **File** -> **Open File** and then, select the JSON file from explorer.

Figure 5.5 depicts the JSON file created from Code Snippet 5.

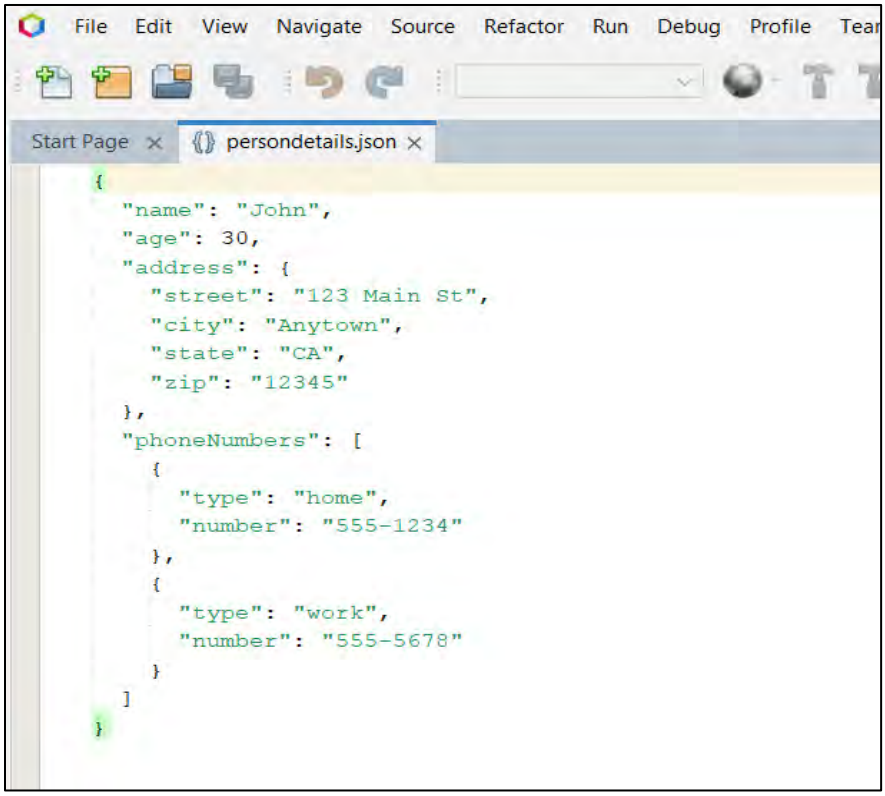


Figure 5.5: JSON File

When the JSON file is opened using Chrome Browser, it looks as shown in Figure 5.6.

Figure 5.6: JSON File Opened with Chrome Browser

5.7 Summary

- JSON is a simple, text-based, lightweight data transfer standard.
- JSON is frequently used to transfer data between clients and servers or between various application components.
- JSON is the de facto standard for data exchange on the Web.
- JSON can be easily used for data transmission in Internet applications, mobile apps, and other software systems.
- JSON is simple, lightweight, language-independent, and human-readable.
- JSON and XML are commonly used to represent and transfer structured data.
- JSON is straightforward, concise, and easier to scan and process, whereas, in comparison, XML is more versatile, extendable, and commonly used in legacy systems.
- Apache NetBeans is a popular IDE and supports creation and working with JSON files.

5.8 Check Your Progress

1. _____ are collections of characters encased in double-quotes.

A.	Booleans	C.	Numbers
B.	Strings	D.	Arrays

2. A JSON _____ is a collection of key-value pairs.

A.	Object	C.	Array
B.	Boolean	D.	String

3. _____ is an ordered group of values enclosed in square brackets and separated by commas.

A.	Object	C.	String
B.	Number	D.	Array

4. _____ braces surround the JSON objects.

A.	Square - []	B.	Angle - <>
B.	Curly – {}	D.	Round – ()

5. _____ has one or more JSON objects as values for its attributes.

A.	Array	B.	Nested JSON
B.	Key	D.	Object

5.8.1 Answers for Check Your Progress

Question	Answer
1	C
2	A
3	D
4	C
5	B

Answers to Knowledge Check 1

1) True
2) False

Try It Yourself

Task 1: Identify other popular IDEs or editors that can be used to create and open JSON files, apart from NetBeans.

Task 2: Create a JSON file (Example 'persondetails.json') and try to open it via Chrome browser.

Task 3: Explain each component in a JSON file.

Session 6

Working with JSON

This session imparts a comprehensive understanding of JSON data types, data structures, schema usage, metadata, and comments. It also explains the practical skills required for creating and parsing JSON messages in JavaScript.

Objectives

- Explain various data types used in JSON
- Explain the fundamental data structures available in JSON
- Define JSON schema and explain its role in defining data structure constraints
- Explain how to create and parse JSON Messages with JavaScript

6.1 JSON Data Types

In JSON, various data types are utilized to represent different kinds of values. JSON supports following data types:

String

- A string is a sequence of characters enclosed in double quotes (""), capable of containing alphanumeric characters, symbols, and escape sequences. For instance, "Hello, world!".

Number

- A number can be either an integer or a floating-point value. JSON treats both as decimal notations. For example, 42 or 3.14.

Boolean

- Boolean represents either true or false and is written as such, without quotes. For instance, true.

Null

- Null signifies a null or empty value and is written as the keyword null, without quotes. For example, null.

Object

- An object is an unordered collection of key-value pairs. Keys are strings and values can be any valid JSON data type. Key-value pairs are separated by commas, enclosed in curly braces ({}). For example, {"name": "John", "age": 30}.

Array

- An array is an ordered list of values enclosed in square brackets ([]). Values can be any valid JSON data type, including arrays and objects. Array elements are separated by commas. For example, [1, 2, 3, 4].

These data types can be combined and nested to represent intricate data structures in JSON.

6.2 Data Structures Supported by JSON

Object

An object, represented by curly braces ({}), is an unordered collection of key-value pairs. Each key is a string and each value can be any valid JSON data type. Objects are frequently used to represent complex data structures or entities.

Example: A JSON object is a versatile data structure that captures key-value relationships. In JSON syntax, data is enclosed within curly braces ({}) with keys as strings and values embracing diverse data types.

Code Snippet 1: student.json

```
{
  "name": "John Doe",
  "age": 30,
  "email": "johndoe@example.com",
  "isStudent": false,
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "country": "USA"
  },
  "favoriteColors": ["blue", "green", "red"]
}
```

6.3 JSON Schema, Metadata, and Comments

JSON Schema

JSON Schema acts as a framework for annotating and validating JSON data, ensuring its structure, format, and content meet standardized criteria. It uses JSON format and follows a specific schema definition to describe expected properties, data types, and constraints in JSON documents.

JSON Schema accomplishes following tasks:

Defining Data Types

- It allows the programmer to define the anticipated data types for properties, such as strings, numbers, booleans, arrays, objects, or null values.

Specifying Property Definitions and Requirements

- JSON Schema permits the programmer to specify the expected properties of an object and determine whether they are mandatory or optional.

Setting Constraints

- The programmer can set constraints, such as minimum and maximum values, string lengths, regular expressions, and so on.

Nesting Objects and Arrays

- JSON Schema enables the definition of nested objects and arrays, outlining their structure and limitations.

Creating Enumerations

- It allows the programmer to establish a list of acceptable values for a property.

Adding References

- JSON Schema enables references to other JSON Schema definitions, promoting the reuse, and composition of schemas.

Specifying Conditional Validation

- The programmer can specify conditional validation rules based on the values of other properties.

Employing JSON Schema empowers the programmer to ensure that JSON data conforms to a predefined structure and adheres to a set of regulations. It facilitates the validation, interpretation, and manipulation of JSON documents.

For example, consider a JSON schema outlining the structure of a person's object, including properties such as `name`, `age`, and `email`.

Following is an illustration of a JSON schema for a person object:

Code Snippet 3: Person.json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Person",
  "type": "object",
  "properties": {
    "name": {
```

```
      "type": "string"
    },
    "age": {
      "type": "integer",
      "minimum": 0
    },
    "email": {
      "type": "string",
      "format": "Email"
    }
  },
  "required": ["name"]
}
```

In this example, the JSON schema states that a person's object must encompass the properties of name, age, and email. The schema stipulates that the name property is obligatory, signified by the `required` keyword. Furthermore, it designates the age property as an `integer` type, defining that its value must not exceed 0. Additionally, the email property is specified as a `string` type, subject to a format restriction of `email`, thereby ensuring that it aligns with the format of a valid email address.

Metadata

Metadata, in the context of JSON, pertains to supplementary information or descriptive details associated with a JSON document. This additional information offers valuable context and insights about the data, enabling inclusion of annotations, properties, or values that go beyond the raw data itself.

In the world of JSON, metadata finds its place as extra key-value pairs enclosed within a JSON object.

These key-value pairs serve to convey pertinent information about the data, encompassing:

Versioning

- Metadata can feature a version number or a historical record of versions for the JSON document.

Authorship or Creato

- Metadata can explicitly state the author or creator of the JSON document.

Timestamps

- Metadata can encompass timestamps that denote the moment of creation or any subsequent modifications to the JSON document.

Descriptive Element

- Metadata can house descriptive text or comments, shedding light on the JSON document as a whole or specific properties within it.

Tags or Categories

- Metadata can include tags or categories that assist in the classification or categorization of the JSON document.

Origin or Source

- Metadata can detail the source or origin of the JSON data.

Integrating metadata in this manner enriches the comprehension and administration of JSON data. It furnishes additional context, annotations, or properties that prove instrumental in effectively interpreting and leveraging the data.

To illustrate, assume a JSON object that represents a book, wherein supplementary metadata is integrated to provide comprehensive insights into the book's attributes. Code Snippet 4 shows an example for Metadata.

Code Snippet 4: Book.json

```
{
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "year": 1925,
  "metadata": {
    "isbn": "9780743273565",
    "genre": "Fiction",
    "publisher": "Scribner",
    "language": "English"
  }
}
```

In this instance, the metadata object serves as a repository for supplementary details regarding the book. It encompasses properties such as `isbn`, `genre`, `publisher`, and `language`, enriching the dataset with contextual and descriptive information beyond the fundamental particulars such as the title, author, and publication year.

Metadata serves as a versatile tool capable of housing diverse types of information. It includes versioning specifics, licensing details, references to sources, or any other pertinent data that augments the core content within the JSON document.

Comments

Comments in JSON are non-functional text pieces providing explanations or information within JSON and it lacks native comment syntax. Some JSON parsers or tools introduce comment support as an extension or supplementary feature.

JSON parsers usually disregard comments, considering them extraneous to the data. Instead, comments function as human-readable annotations, proving valuable for documenting the JSON structure, properties, or values. These comments can significantly aid developers in comprehending the JSON document or providing supplementary instructions.

Though comments are not universally embraced in JSON, specific libraries or tools introduce their conventions or formats for embedding comments in JSON files. Although JSON parsers disregard these comments, they can be invaluable during development or when sharing JSON files among a team.

It is crucial to remember that comments should not serve as the primary means to convey critical information within JSON data. Their existence or absence can vary depending on the parsing or processing environment.

As previously mentioned, JSON lacks native support for comments. However, particular tools or libraries introduce comment inclusion using conventions that some parsers or tools recognize. Following is an example of how comments can be incorporated into a JSON document, adhering to a convention understood by specific parsers or tools:

Code Snippet 5: tools.xml

```
{
  "name": "John Doe",
  "age": 30,
  "city": "New York",

  /* This is a comment explaining the occupation */
  "occupation": "Software Engineer",

  // This is another comment about the hobbies
```

```
  "hobbies": [
    "Reading",
    "Traveling"
  ]
}
```

In this example, two distinct conventions are employed to introduce comments. The initial comment is enclosed within `/*  */` for multiline comments, while the second comment utilizes `//` for single-line comments.

These comments are intended for human consumption, serving the purpose of offering supplementary information or explanations pertaining to specific properties within the JSON document.

Generally, JSON parsers overlook comments and should not be counted as integral or functional components of the JSON data itself.

6.4 Creating and Parsing JSON Messages with JavaScript

Creating a JSON message involves defining a JSON object, a concept which is already familiar to developers by now.

Parsing refers to scrutinizing a string of data or text per a set of predefined rules or a specific structure. Parsing is common in programming and data processing. It involves converting serialized data (Example JSON strings) into a structured format for computer manipulation and understanding.

Parsing JSON refers to considera JSON string and transfers it into a corresponding data structure or object within the programming language. This process entails the interpretation of JSON syntax, its validation, and the creation of an appropriate data structure that faithfully mirrors the JSON data.

During parsing, the JSON string is generally analyzed character by character and the parser confirms its adherence to valid syntax while adhering to the regulations stipulated in the JSON specification. It identifies elements such as objects, arrays, strings, numbers, booleans, and null values along with their respective syntax and nesting protocols.

After the JSON string has been successfully parsed, it becomes accessible and adaptable through the data structures and methods native to the programming language. It empowers developers to work with the JSON data in an organized and meaningful manner. Activities such as accessing properties, iterating through arrays, or conducting operations based on the parsed data become feasible.

Parsing plays a crucial role in data processing by transforming raw data into a structured format suitable for processing and manipulation. It ensures data becomes accessible and adaptable for computer programs enhancing their functionality.

Creating JSON Messages

To create a JSON message in JavaScript, the programmer can define an object and subsequently convert it into a JSON string using the `JSON.stringify()` method. Following is an example for creating JSON Messages:

Code Snippet 6: CreatingJSON.js

```
/*
 * Click
nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
default.txt to change this license
 * Click
nbfs://nbhost/SystemFileSystem/Templates/Other/javascript.js to
edit this template
 */
// Create an object representing a person
var person = {
  "name": "John Doe",
  "age": 30,
  "city": "New York"
};

// Convert the object to a JSON string
var jsonMessage = JSON.stringify(person);

print(jsonMessage);
```

The start part of the code inside comments is auto-generated by NetBeans in an JavaScript file.

In this Code Snippet 6, there is an object named `person` with properties including `name`, `age`, and `city`. The `JSON.stringify()` method transforms this object into a JSON string representation. To execute the code, click the green run button highlighted in Figure 6.1.

The result is the JSON data as depicted in the Output Pane in Figure 6.1.

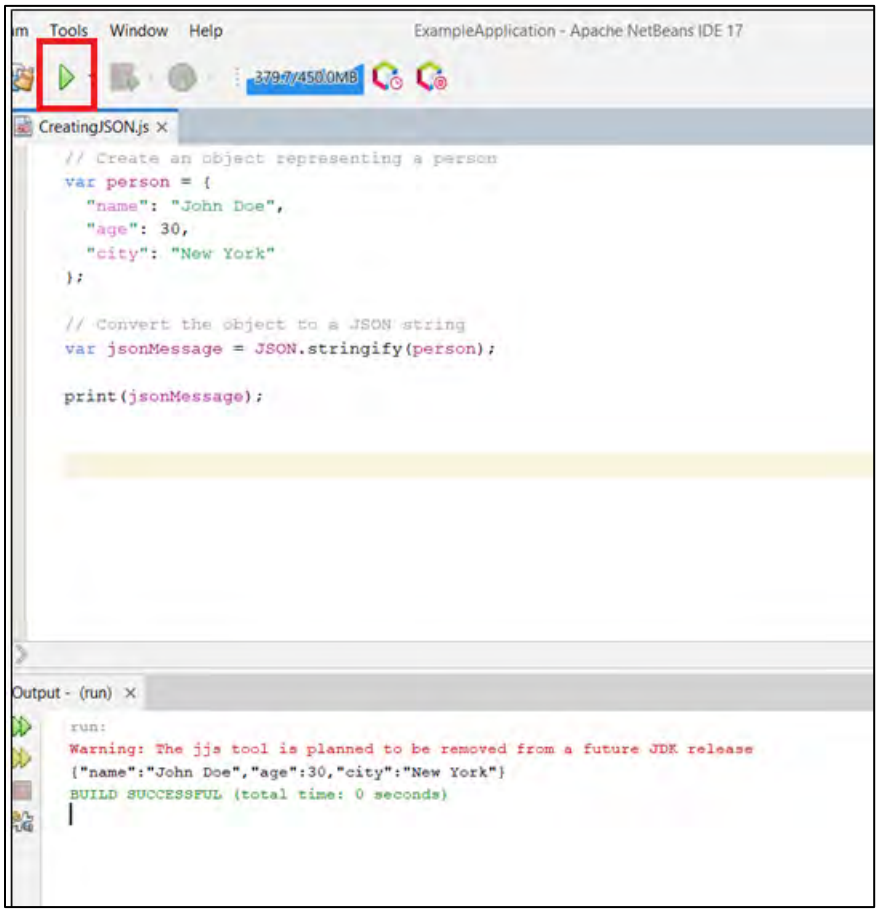


Figure 6.1: Creating JSON Messages

Parsing JSON Messages

To parse a JSON message in JavaScript, the programmer can utilize the `JSON.parse()` method to convert the JSON string into a JavaScript object. An illustrative example is shared in Code Snippet 7.

Code Snippet 7: ParsingJSON.js

```
/*
 * Click
nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
default.txt to change this license
 * Click
nbfs://nbhost/SystemFileSystem/Templates/ClientSide/javascript.
js to edit this template
 */
// A JSON string representing a person
```

```
var jsonMessage = '{"name":"John Doe","age":30,"city":"New
York"}';

// Parse the JSON string into a JavaScript object
var person = JSON.parse(jsonMessage);

print(person.name);  // Output: John Doe
print(person.age);   // Output: 30
print(person.city);  // Output: New York
```

In this instance, a JSON string named `jsonMessage` embodies information about a person. Through the application of the `JSON.parse()` method, this JSON string is transformed into a JavaScript object. Subsequently, the ability to access the properties of the `person` object using dot notation is achieved. The result is the JSON output as depicted in Figure 6.2.

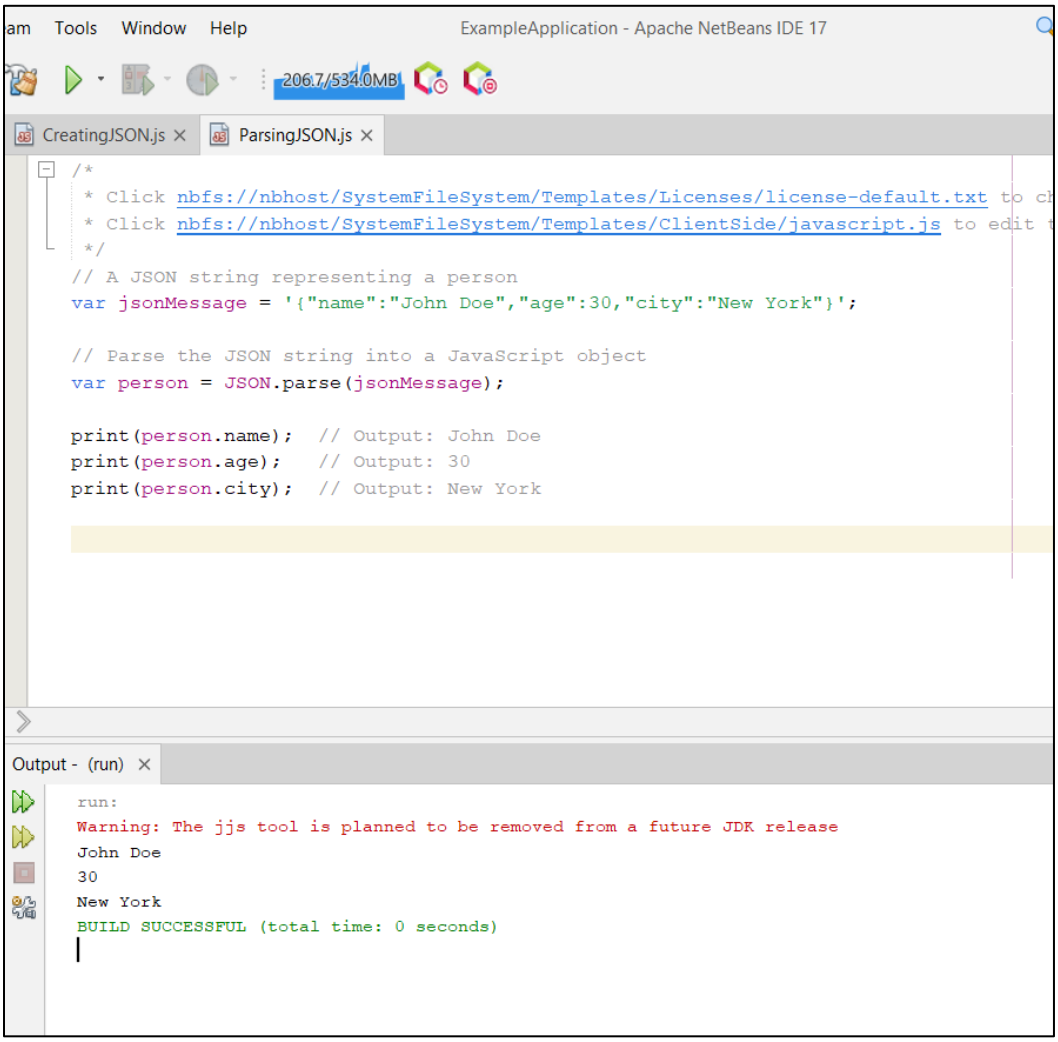


Figure 6.2: Parsing JSON Messages

By crafting and parsing JSON messages in JavaScript, the programmer can efficiently convert data back and forth between JavaScript objects and JSON representations. It facilitates smooth communication and data interchange with APIs, servers, or other systems that operate using JSON.

6.5 Summary

- JSON supports various data types for representing different kinds of values.
- The data types can be combined and nested to represent complex data structures.
- JSON Schema acts as a framework for annotating and validating JSON data.
- Metadata in JSON refers to supplementary information or descriptive details associated with a JSON document.
- Comments in JSON serve as non-functional text pieces providing explanations or information.
- JSON lacks native comment syntax, but some parsers or tools support comments.
- Parsing JSON in JavaScript plays a crucial role in data processing by transforming raw data into a structured format suitable for processing and manipulation.

6.6 Check Your Progress

1. Which JSON data structure is used for an unordered collection of key-value pairs?

A.	Array	C.	Object
B.	String	D.	Number

2. What is the primary purpose of a JSON schema?

A.	Adding comments to JSON data	C.	Storing metadata in JSON
B.	Validating the structure and data types of JSON	D.	Creating JSON data

3. In JavaScript, which method is used to convert a JavaScript object to a JSON string?

A.	JSON.stringify()	C.	JSON.convert()
B.	JSON.parse()	D.	object.toJSON()

4. When parsing a JSON string with JavaScript, which of the following will throw an error if JSON is invalid?

A.	JSON.stringify()	C.	JSON.check()
B.	JSON.parse()	D.	JSON.load()

5. Which of the following is NOT a valid JSON data type?

A.	String	C.	Boolean
B.	Integer	D.	NULL

6.6.1 Answers for Check Your Progress

Question	Answer
1	B
2	B
3	A
4	B
5	B

Try It Yourself

Task 1: Generate a JSON object that depicts an individual's details, encompassing diverse data types such as strings, numbers, booleans, null. Incorporate their name, age, pet ownership status, and any supplementary information.

Task 2: Create a JSON array of objects where each represents a product with properties such as name, price, and quantity in stock.

Session 7

Advanced JSON Features

and

This session describes how JSON is supported in programming languages and explains its use in Web APIs. It also defines Multipurpose Internet Mail Extensions (MIME) types and describes its significance in data storage. Further, it addresses security and data portability concerns.

Objectives

- Identify JSON's support with programming languages
- Define the usage of JSON in Web APIs and MIME types
- Describe JSON's role in data storage, security, and data portability

7.1 JSON Support with Programming Languages

JSON offers robust support in numerous widely-used programming languages, with explanations of its use in some of these languages provided as follows:

JavaScript: JavaScript has native support for JSON. It offers methods such as `JSON.stringify()` to convert JavaScript objects into JSON strings and `JSON.parse()` to parse JSON strings back into JavaScript objects.

Python: Python includes a built-in module called `json` which manages the data effectively. It includes methods such as `json.dumps()` to convert Python objects to JSON strings and `json.loads()` to parse JSON strings into Python objects.

Java: Java relies on libraries such as Jackson, Gson, and org.json for offering extensive support for JSON. These libraries enable seamless serialization and deserialization of Java objects to and from JSON strings, simplifying JSON data handling in Java.

C#: In C#, the `System.Text.Json` namespace provides robust JSON support. It includes classes such as `JsonSerializer` for object serialization and deserialization to and from JSON strings, alongside other utility classes for efficient JSON data manipulation.

Ruby: Ruby incorporates a built-in `json` library for executing JSON operations, providing native support for handling JSON data within the language. Methods such as `json.dump()` allow the conversion of Ruby objects to JSON strings and `JSON.parse()` facilitates parsing JSON strings into Ruby objects.

PHP: PHP incorporates functions such as `json_encode()` and `json_decode()` that facilitate the seamless conversion between PHP data structures and JSON strings. These functions are widely employed for encoding and decoding JSON data in PHP.

Go: Go has an in-built package, encoding/json, designed to handle JSON operations. Functions such as `json.Marshal()` enables the conversion of Go data structures to JSON, while `json.Unmarshal()` facilitates parsing JSON strings into Go data structures.

Swift: Swift offers built-in JSON support through the Codable protocol. By conforming to Codable, Swift objects can be easily encoded to JSON using `JSONEncoder` and decoded from JSON using `JSONDecoder`.

These examples showcase the native or commonly used libraries for JSON support in various programming languages. JSON's widespread adoption and simplicity ensure robust support across diverse programming languages, facilitating seamless integration and data exchange between different systems and platforms.

7.2 Web APIs

Web APIs function as sets of protocols and rules enabling various software applications to communicate and interact via the Internet. They provide a standardized approach for applications to request and exchange data, execute operations, and access functionalities offered by a Web server or service.

Web APIs permits developers to integrate their applications with external services, access remote resources, and construct more dynamic and interactive Web applications. They facilitate data exchange in various formats, including JSON and XML, utilizing HTTP as the fundamental communication protocol. Some of the critical aspects of Web APIs are explained as follows:

Request-Response Model

- Web APIs adhere to a request-response model, where an application sends a request to a Web server or service and receives a response containing the requested data or information.

RESTful APIs

- REST stands as a widely adopted architectural style for designing Web APIs.
- RESTful APIs utilize standard HTTP methods such as GET, POST, PUT, DELETE to perform operations on resources identified by URLs.

Data Formats

- Web APIs commonly use data formats such as JSON or XML to represent the transmitted data between client and server.
- JSON has become the preferred standard due to its simplicity and compatibility with many programming languages.

Authentication and Authorization

- APIs necessitate authentication and authorization mechanisms to ensure secure access and safeguard sensitive data.
- Common methods include API keys, OAuth, and token-based authentication.

Documentation

- Well-designed APIs encompass clear and comprehensive documentation providing information about available endpoints, request/response formats, authentication requirements, error handling, and usage examples.
- This aids developers in effectively comprehending and utilizing the API.

Rate Limiting

- API providers frequently implement rate limiting mechanisms to prevent abuse and ensure equitable usage.
- Rate limiting imposes restrictions on the number of requests a client can make within a specified time frame.

Web APIs find extensive applications across diverse domains, including social media platforms, payment gateways, weather services, mapping services, and so on. They enable developers to harness existing services, integrate various systems, construct innovative applications leveraging the potential of the Internet and interconnectedness.

7.3 MIME Type of JSON

The MIME type of JSON is typically represented as 'application/json'.

MIME types serve the purpose of identifying the nature of data being transmitted across the Internet. They play a crucial role in helping servers and clients determine how to process and interpret the content of files or data in transit.

The application/json MIME type indicates that the content being sent or received is in JSON format. It is employed within HTTP headers, specifically the Content-Type header to convey to the recipient that the data is structured as JSON.

When a server responds with JSON data, it typically includes the Content-Type header with the value set as application/json. It informs the client that the response contains JSON data, allowing the client to handle and parse it correctly.

For instance, a response header featuring the Content-Type set as application/json would appear as follows:

```
Content-Type: application/json
```

The application/JSON MIME type boasts widespread support across diverse platforms, programming languages, and Web browsers. This uniformity guarantees compatibility and consistency when dealing with JSON data across the Internet.

7.4 JSON HTTP and Files

JSON can be used in various contexts related to HTTP and files. Here is an explanation of how JSON is commonly used in these scenarios:

JSON and HTTP Requests/Responses:

JSON is extensively employed as a data format for managing information in HTTP requests and responses during client-server communication. When initiating an HTTP request, a client can incorporate JSON data within the request payload, usually placed in the request's body.

Subsequently, the server can interpret and process the JSON data, crafting a response that is formatted in JSON. This JSON-formatted response is then transmitted as part of the HTTP response.

For instance, consider an HTTP GET request as shown in Code Snippet 1.

Code Snippet 1: ApiFetching.js

```
// Get a reference to the HTML element where you want to display
//the JSON data.
const outputDiv = document.getElementById("output");

// Define the URL of the JSON API you want to fetch.
const apiUrl = "https://restcountries.com/v3.1/name/france";

// Use the Fetch API to make an HTTP GET request to the API.
fetch(apiUrl)
    .then(response => {
        if (!response.ok) {
            throw new Error("Network response was not ok");
        }
        return response.json(); // Parse the response as JSON.
    })
    .then(data => {
  // Handle the JSON data here. You can display it on the
//page or perform any other actions.
        displayData(data);
    })
    .catch(error => {
        console.error("There was a problem with the fetch
operation:", error);
```

```
    });

// Function to display JSON data on the page.
function displayData(data) {
    // Example: Displaying the data in a <pre> element as a
//formatted JSON string.
    outputDiv.innerHTML = `<pre>${JSON.stringify(data, null,
2)}</pre>`;
}
```

Code Snippet 2: APIExample.html

```
 <!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JSON API Example</title>
</head>
<body>
    <div id="output"></div>
    <script src="ApiFetching.js"></script>
</body>
</html>
```

In Code Snippet 1, the server requests the JSON data, which then undergoes processing. The sever then formulates a fitting HTTP response, embedding JSON data within the response body. The interface used (`https://restcountries.com`) is an online REST API available for free.

The HTML code given in Code Snippet 2 should be executed via the local Web server. These Code Snippet files should be saved in the htdocs folder of XAMPP. Ensure the Apache Server is running in XAMPP control Panel. The URL used to execute the JavaScript code for fetching the details using HTTP GET API is https://localhost.APIExample.html.

The response body is as shown in Figure 7.1.

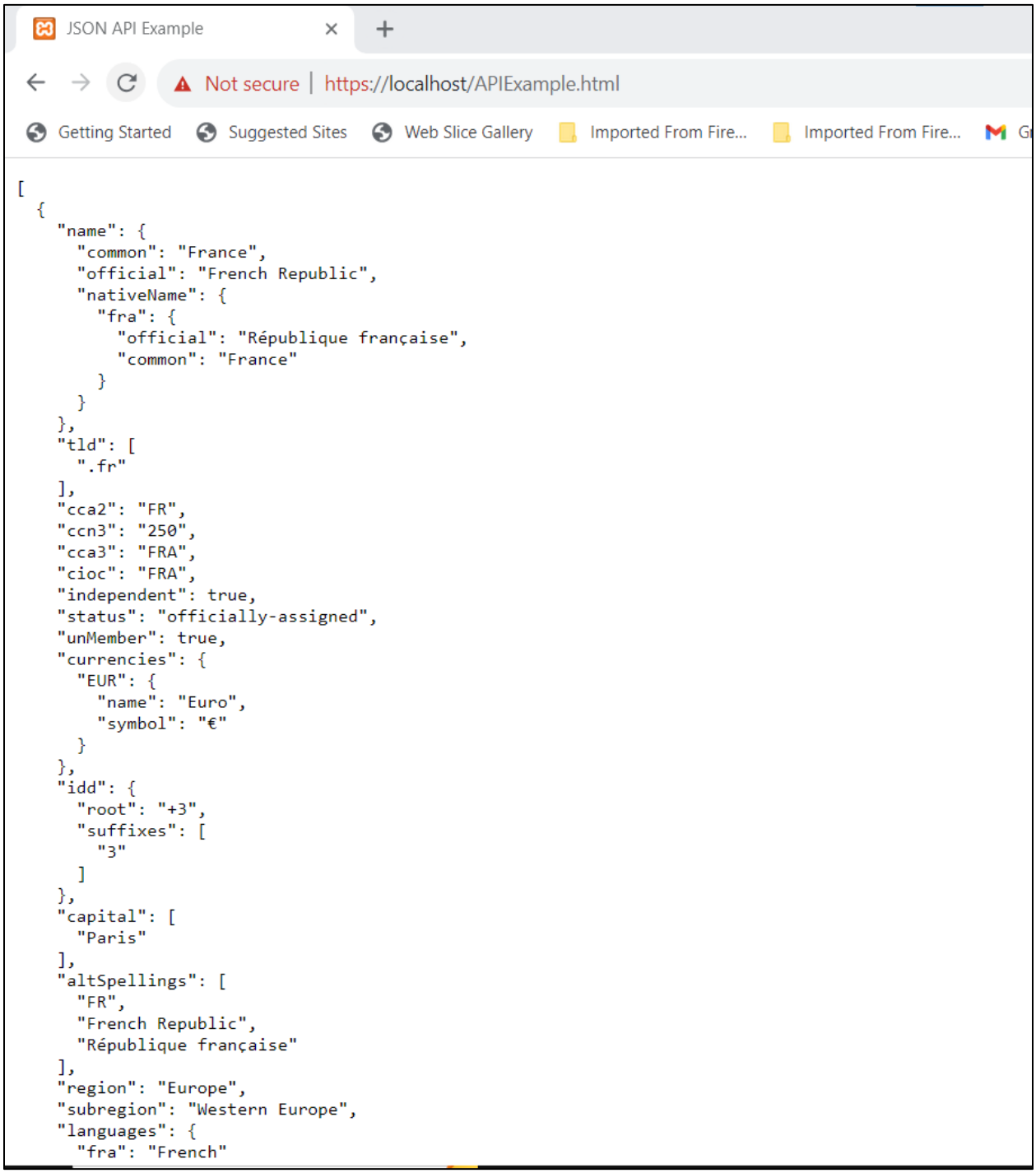


Figure 7.1: JSON Output from a GET API

JSON Files:

JSON files store data in JSON format, typically sporting the `.json` file extension. Within these files, data is structured through key-value pairs, arrays, and nested objects.

JSON files find frequent use in tasks such as configuration file management, data storage, and data exchange.

For instance, `users.json` file could contain an array containing objects.

Code Snippet 3:

```
JSON
[
  {
    "name": "John Doe",
    "email": "john@example.com"
  },
  {
    "name": "Jane Smith",
    "email": "jane@example.com"
  }
]
```

JSON files are accessible for reading and parsing by applications and programming languages, enabling access to and manipulation of the enclosed data. For example, JSON data can be serialized and saved to a JSON-formatted file for storage or sharing purposes.

JSON APIs and Web Services:

JSON is pivotal in APIs and Web services, facilitating data exchange between clients and servers. Web APIs offer endpoints that furnish JSON data as responses, allowing clients to retrieve, modify, or delete resources.

Clients can initiate HTTP requests to these APIs, specifying the resource they require and the preferred format, which is usually JSON.

The server processes the request, executes the necessary actions, and replies with JSON data representing the requested resource or providing status information. JSON's lightweight and easily comprehensible attributes make it an excellent choice for data transmission over the Web.

It can be effortlessly parsed, generated, and interpreted by many programming languages and platforms, ensuring seamless integration, and interoperability across diverse systems.

7.5 JSON for Data Storage

JSON is a widely favored data storage format due to its simplicity, legibility, and extensive support across diverse programming languages and platforms. It finds widespread use in persisting and exchanging structured data. Some of the fundamental aspects of utilizing JSON for data storage are explained as follows:

Data Structure

- JSON employs a blend of key-value pairs, arrays, and nested objects to represent data. This adaptable structure allows for the storage and organization of intricate data hierarchies, effortlessly representing entities, relationships, and attributes.

Human-Readable Format

- JSON is designed for ease of comprehension by both humans and machines. Its syntax is straightforward, simplifying data inspection and debugging. This human-readable format proves advantageous when working directly with data or sharing it with others.

Serialization and Deserialization

- JSON can be serialized, meaning it can be converted from a data structure or object into a string representation. This serialized JSON string can be stored in files, databases, or transmitted across networks. Conversely, JSON can be deserialized, converting the serialized JSON string back into the original data structure or object.

Language Neutrality

- JSON is language-neutral, making it compatible with nearly any programming language. JSON libraries and parsers are available for virtually all programming languages, facilitating seamless integration and manipulation of JSON data.

NoSQL Database Compatibility

- Numerous NoSQL databases, including MongoDB and CouchDB, natively support JSON as a storage format. These databases can directly store JSON documents, establishing smooth integration between the database and the application layer.

Data Interchange Format

- JSON is widely employed as a data interchange format, enabling different systems to exchange data in a standardized, platform-agnostic manner. It is a common choice in APIs and Web services, facilitating data exchange between clients and servers.

Minimal Overhead

- JSON imposes minimal overhead when compared to other data storage formats such as XML. It is less verbose, resulting in smaller file sizes and expedited data transmission over networks.

However, it is essential to recognize that JSON only suits some data storage use cases. Specialized databases or data storage solutions prove more suitable for scenarios necessitating data querying and intricate operations. JSON's forte lies in its simplicity, adaptability, and effectiveness for efficiently storing and exchanging structured data.

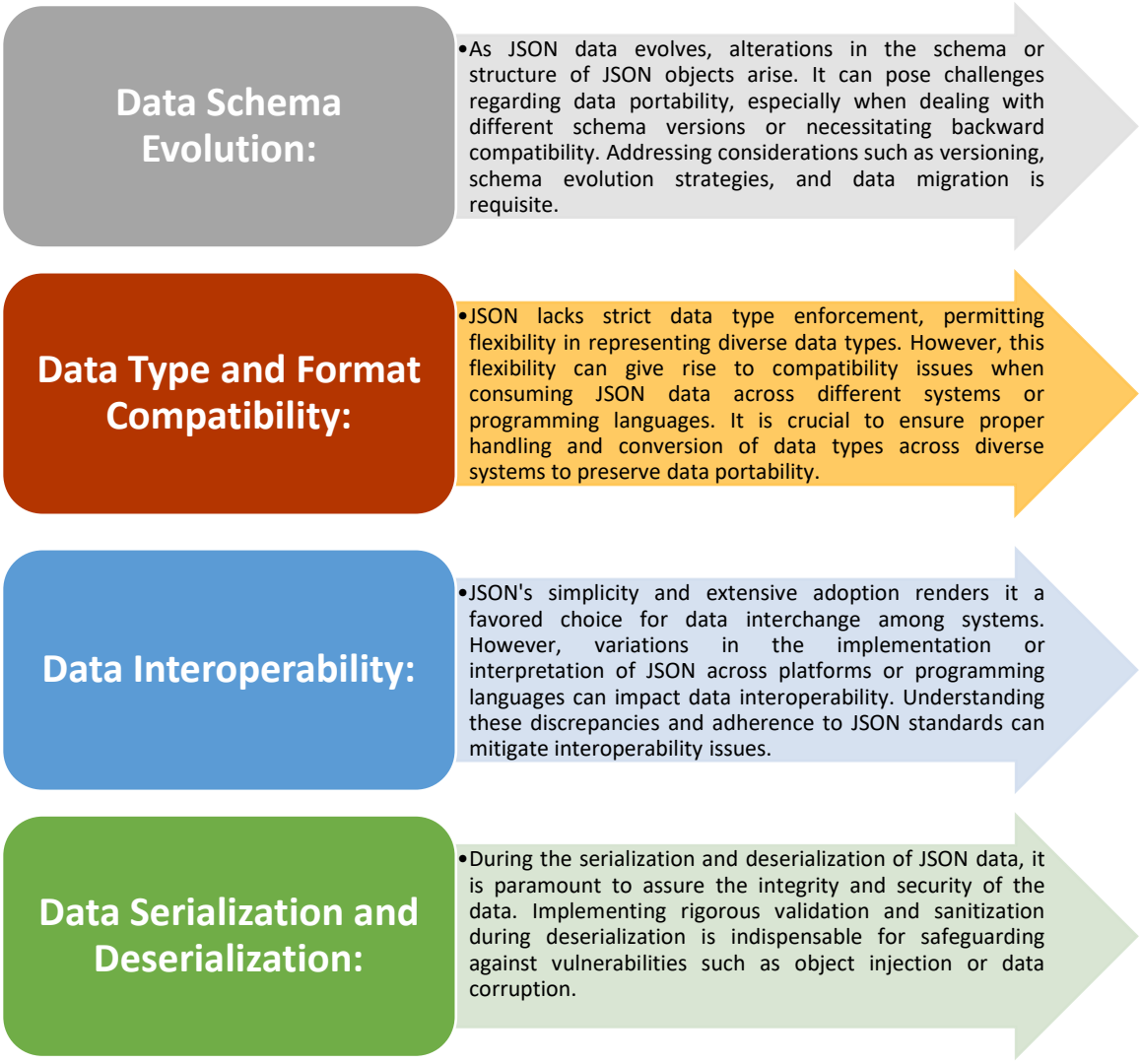
7.6 JSON Security and Data Portability Issues

JSON is primarily a data format and does not inherently encompass security or portability features. However, when working with JSON, it is essential to consider significant aspects related to security and data portability, such as:

Security Considerations

JSON Injection:	•Similar to other injection attacks, such as SQL injection, JSON injection can transpire if untrusted data is directly incorporated into a JSON string without meticulous validation or sanitization. It can result in security vulnerabilities, encompassing code execution, data manipulation, or unauthorized access. When the risks are migrated, it is essential to implement input validation and diligently apply proper escaping or encoding to programmer-supplied data.
Cross-Site Scripting (XSS):	•In scenarios where JSON data is utilized within a Web application without appropriate output encoding, the application can be susceptible to XSS attacks. It is imperative to ascertain that programmer-generated content within JSON data undergoes thorough escaping or sanitization to preclude the execution of scripts within a client's browser.
Access Control and Authorization:	•When exposing JSON APIs or services, enforcing robust access control mechanisms and authorization rules is pivotal. It ensures that solely authenticated and authorized programmers or applications can access sensitive data or execute specific operations.
Secure Transmission:	•During the transmission of JSON data across the network, employing protocols such as HTTPS is imperative. It safeguards the confidentiality and integrity of the data during transit.

Data Portability Considerations



In conclusion, JSON lacks inherent security or data portability features. However, it is crucial to consider and mitigate security vulnerabilities when handling JSON data in applications or systems. Developers should also address data portability challenges associated with JSON. Implementing suitable security measures and adherence to best practices can mitigate risks and guarantee the secure and portable handling of JSON data.

7.7 Summary

- JSON is widely supported in many programming languages, such as JavaScript, Python, Java, C#, Ruby, PHP, Go, and Swift. These languages have built-in and readily available libraries for JSON handling.
- Web APIs function as sets of protocols and rules enabling various software applications to communicate and interact via the Internet.
- MIME types help identify the format of data being transmitted over the Internet.
- JSON is commonly used in HTTP requests and responses, especially in RESTful APIs.
- JSON is favored for data storage due to its simplicity and compatibility with various programming languages.
- Data portability and security considerations include schema evolution, data type compatibility, data interoperability, and secure serialization/deserialization.

7.8 Check Your Progress

1. Which HTTP method typically sends JSON data in an HTTP request?

A.	GET	C.	PUT
B.	POST	D.	DELETE

2. In which programming language is the JSON module commonly used for handling JSON data?

A.	JavaScript	C.	Java
B.	Python	D.	Ruby

3. Which of the following is NOT an essential aspect of Web APIs?

A.	Data interchange in JSON format	C.	MIME type for JSON
B.	Rate limiting	D.	Authentication and authorization

4. Which characteristic makes JSON an excellent choice for data transmission over the Web?

A.	Strict data type enforcement	C.	Human-readability
B.	Complexity of syntax	D.	Limited programming language support

5. Which of the following is NOT a consideration for data portability when working with JSON data?

A.	Data serialization	C.	Data type compatibility
B.	Data schema evolution	D.	API rate limiting

7.8.1 Answers for Check Your Progress

Question	Answer
1	B
2	B
3	C
4	C
5	D

Try It Yourself

Task 1: Write a simple program demonstrating how your chosen programming language can create, manipulate, and parse JSON data. Examples should include creating JSON objects, adding data, and parsing JSON strings.

Task 2: Write a program in your preferred programming language to make a GET request to the chosen API, retrieve JSON data, and display relevant information from the JSON response.

Session 8

Cross-Origin Resource Sharing (CORS)

This session describes Cross-Origin Resource Sharing (CORS) and its related terminology. It also outlines the benefits of implementing CORS policies, the concept of origins and their types, and elaborates how to implement CORS policies in a Web browser.

Objectives

- Elaborate CORS and its terminology
- Summarize the benefits of implementing CORS, Origin and explore its different types
- Outline the process of implementing the CORS policy in Web browsers and various types of CORS requests sent by a Web browser

8.1 CORS and its Terminology

'CORS' stands for Cross-Origin Resource Sharing. It is a protocol and security standard implemented by Web browsers to maintain the security and integrity of Websites, protecting them from unauthorized access.

CORS allows JavaScript/HTML running in Web browsers to connect to APIs and access various Web resources from different providers, such as fonts and stylesheets. It ensures that a browser's scripts can securely access resources outside the browser's domain.

The CORS policy is part of the Fetch standard, defined by the Web Hypertext Application Technology Working Group (WHATWG) community. This community publishes various Web standards such as HTML5, DOM, and URL.

According to the Fetch standard specifications, the CORS protocol involves headers indicating whether a response can be shared across different origins. A CORS-preflight request is executed when requests surpass the capabilities of HTML's form element. It also performs the verification of the URL CORS support.

There are several scenarios in which CORS becomes important when browsers fetch resources:

Displaying a map of a programmers location in an HTML or single-page application hosted on xyz.com by Calling Google's Map API (https://maps.googleapis.com/maps/api/js).

Showing tweets from a public Twitter handle in an HTML hosted on the domain xyz.com by requesting the Twitter API (https://api.x.com/xxx/tweets/xxxxx).

Utilizing Web fonts such as Typekit and Google Fonts in an HTML hosted on xyz.com from their respective remote domains.

CORS is a vital security standard that allows Web applications to access resources from different origins while ensuring the security and integrity of Websites. It is essential to modern Web development, enabling rich and interactive Web experiences.

8.2 Benefits of CORS

CORS serves as a security mechanism that has been integrated into Web browsers. Its primary purpose is to permit Web pages from distinct domains to conduct cross-origin requests. CORS offers several valuable advantages:

Heightened Security: CORS plays a pivotal role in safeguarding programmers and Web applications against malicious attacks such as XSS and Cross-Site Request Forgery (CSRF). It enforces same-origin policies and limits cross-origin requests, thus averting unauthorized access to sensitive resources and data.

Inter-Domain Communication: CORS facilitates secure communication between Web pages hosted on different domains. This capability empowers Web applications to retrieve resources, execute AJAX requests, and interact with APIs from various origins. It greatly streamlines the integration of disparate systems and services.

Enhanced Programmer Experience: CORS enables Web pages to load external resources, such as fonts, scripts, or stylesheets, from diverse domains. It enriches the visual presentation and functionality of Web pages by enabling the incorporation of external assets and services.

API Utilization: CORS permits Web applications to use APIs located on different domains, enabling the integration of third-party services and data sources. Developers can leverage external APIs without requiring route requests through their servers.

Cross-Origin Data Sharing: CORS allows data sharing across origins through technologies such as XMLHttpRequest and the Fetch API. It empowers Web applications to fetch data from various sources and employ it within client-side code, expanding data availability for creating dynamic programmer's experiences.

Cross-Domain Single Sign-On (SSO): CORS simplifies cross-origin authentication and authorization, making scenarios such as SSO feasible. Web applications can securely acquire authentication tokens or session information from an identity provider situated on a different domain, simplifying the authentication process for programmers.

Efficient Development and Testing: CORS permits developers to collaborate effectively with multiple servers and domains without cross-origin constraints during the development and testing phases. It facilitates smooth collaboration, debugging, and testing of applications across diverse environments.

In conclusion, CORS balances security and flexibility by enabling controlled and secure communication among Web applications hosted on distinct domains. It promotes to use the cross-origin requests and data sharing within the parameters set by the CORS policy.

8.3 Origin and its Types

In the context of CORS, an 'Origin' is defined by three distinct elements:

URI Scheme

- This can be represented as 'http://' or 'https://'.

Hostname

- It represents the specific domain, such as 'www.pqr.com'.

Port Number

- This includes numerical values such as '8000' or '80', which is the default HTTP port.

Two URLs are considered part of the exact origin only if all three of these elements match.

The terms 'Cross-origin server' and 'origin server' are not officially defined within CORS terminology. However, the programmer can use these terms to describe the server hosting the source application (origin server) and the server to which the browser sends the CORS request. The 'Origin Server' is where the Web page comes from, whereas the 'Cross-Origin Server' is any server different from the Origin Server.

Figure 8.1 illustrates the primary participants involved in a CORS flow.

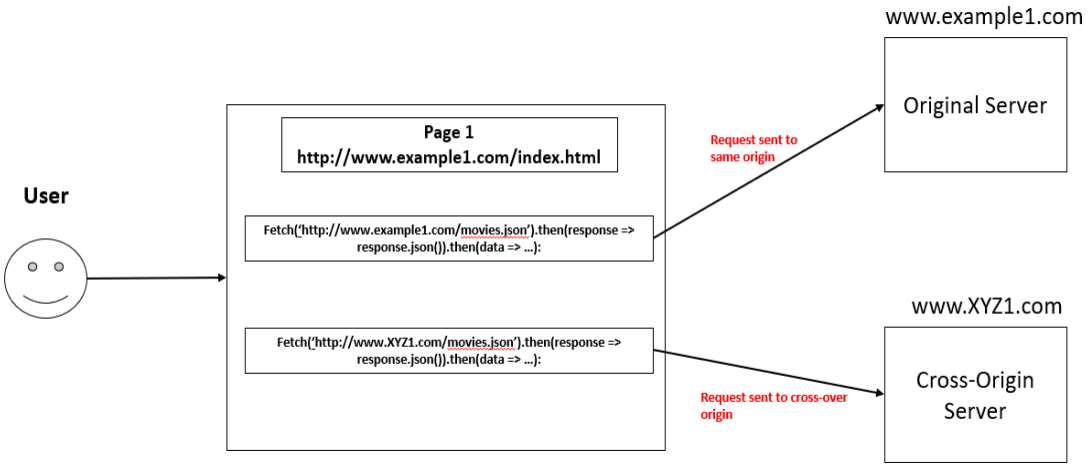


Figure 8.1: Primary Partition in CORS

When a programmer enters the URL `http://www.example1.com/index.html` into the browser, following sequence of actions takes place:

- The browser initiates a request to a server within the domain named www.example1.com. We will refer to this server as the 'Origin server', which hosts the page indexed as index.html.
- The origin server responds by providing the requested page, index.html, as a response to the browser.
- As illustrated in this example, the origin server is also responsible for hosting additional resources, such as the movies.json API.
- Additionally, the browser can retrieve resources from a server in a different domain, for instance, www.XYZ1.com. Programmer will label this server as the 'Cross-Origin server'.
- The browser employs AJAX, utilizing either the built-in XMLHttpRequest object or, the newer `fetch` function within JavaScript. This technology allows the browser to dynamically load content onto the screen without requiring a full-page refresh.

This sequence of events is depicted in Figure 8.2.

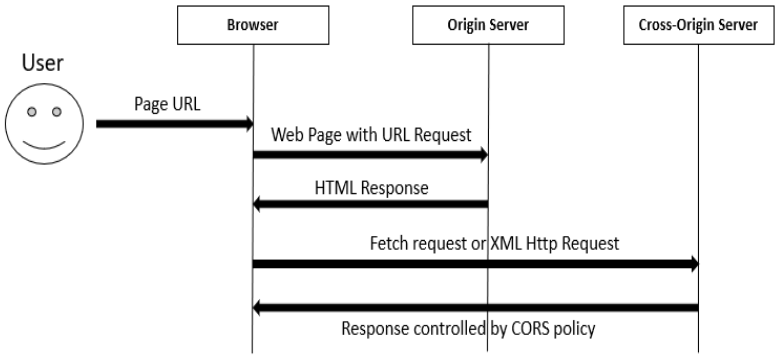


Figure 8.2: Sequence of Events

Same-Origin vs. Cross-Origin

As mentioned earlier, the Same-Origin Policy (SOP) is the default security policy that Web browsers enforce. SOP authorizes the browser to retrieve resources from the Origin Server exclusively.

Without the Same-Origin Policy, scripts downloaded from cross-origin servers could access our Website's DOM. It would enable them to access potentially sensitive data or carry out harmful actions without the programmer's consent.

Table 8.1 compares Same-Origin vs. Cross-Origin.

Aspect	Same-Origin	Cross-Origin
Default Behavior	Permitted by default	Restricted by default
Access to Resources	Allowed	Restricted, subject to CORS policies
XMLHttpRequest	Access Granted	Requires CORS header (Access-Control-Allow-Origin)
Cookies	Shared with origin	Blocked unless specified by CORS
Security	Higher security	Lower security, potential vulnerabilities
Example	Page on www.example1.com makes a request to www.example1.com	Page www.example1.com makes a request to www.XYZ1.com

Table 8.1: Same Origin Vs Cross-Origin

Figure 8.3 illustrates an HTML page named 'initialPage.html' making requests, whether from the exact origin or cross-origin to a target page called 'target1Page.html'. The figure shows that the browser naturally permits same-origin requests while automatically obstructing cross-origin requests.

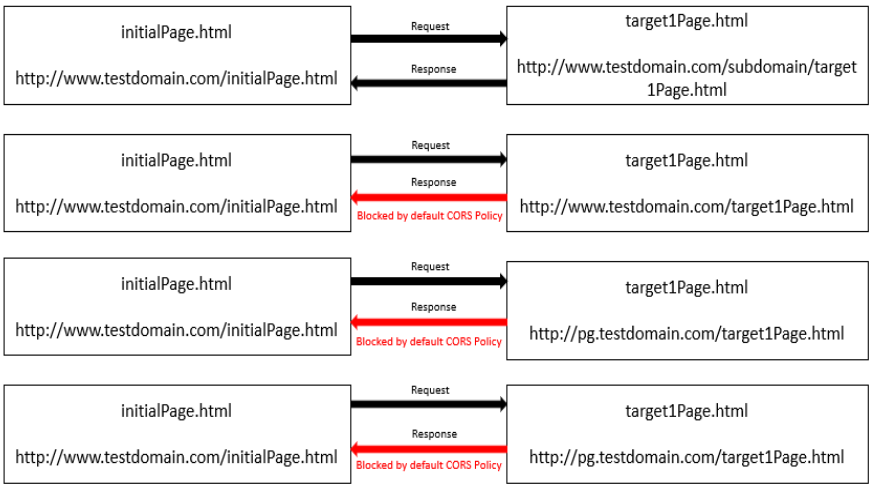


Figure 8.3: HTML Page Making Requests

The URLs for 'target1Page.html' which the browser rendering 'initialPage.html' categorizes as either same-origin or cross-origin, are detailed in Table 8.2. It is important to note that for URLs without explicitly specifying a port, the default port is 443 for HTTPS and 80 for HTTP.

URLs being Matched	Same-Origin or Cross-Origin	Reason
http://www.testdomain.com/target1Page.html	Same-Origin	same scheme, host, and port
http://www.testdomain.com/subpage/target1Page.html	Same-Origin	same scheme, host, and port
https://www.testdomain.com/target1Page.html	Cross-Origin	same host but different scheme and port
http://pg.testdomain.com/target1Page.html	Cross-Origin	different host
http://www.testdomain.com:8080/target1Page.html	Cross-Origin	different port
http://pg.testdomain.com/mypage1.html	Cross-Origin	different host

Table 8.2: URL Matching with Cross-Origin and Same-Origin

When the origins associated with the URLs are identical, it is possible to execute JavaScript code in 'initialPage.html' that can retrieve content from 'target1Page.html.'

Conversely, when dealing with cross-origin URLs, JavaScript code running within 'initialPage.html' will encounter restrictions and can only fetch content from 'target1Page.html 'if an adequately configured CORS policy is in place.

The CORS issue can be seen in Code Snippet 1, which will be executed using the XAMPP tool.

Code Snippet 1: CORS-issue.html

```
<!DOCTYPE html>
<html>
<head>
    <title>CORS Test</title>
</head>
<body>
    <h1>CORS Test Page</h1>
    <button id="fetchData">Fetch Data</button>
    <p id="result"></p>
```

```
    <script>
        const fetchDataButton =
document.getElementById('fetchData');
        const result = document.getElementById('result');

        fetchDataButton.addEventListener('click', () => {
            // Make a request to a different domain here (such
as example.com)
            fetch('https://example.com/api/data')
                .then(response => response.text())
                .then(data => {
                    result.textContent = data;
                })
                .catch(error => {
                    result.textContent = `Error: ${error}`;
                });
        });
    </script>
</body>
</html>
```

The CORS-issue.html has a Fetch data button, that upon clicking, causes a URL of a different domain to be called. This results in the error shown in Figure 8.4.

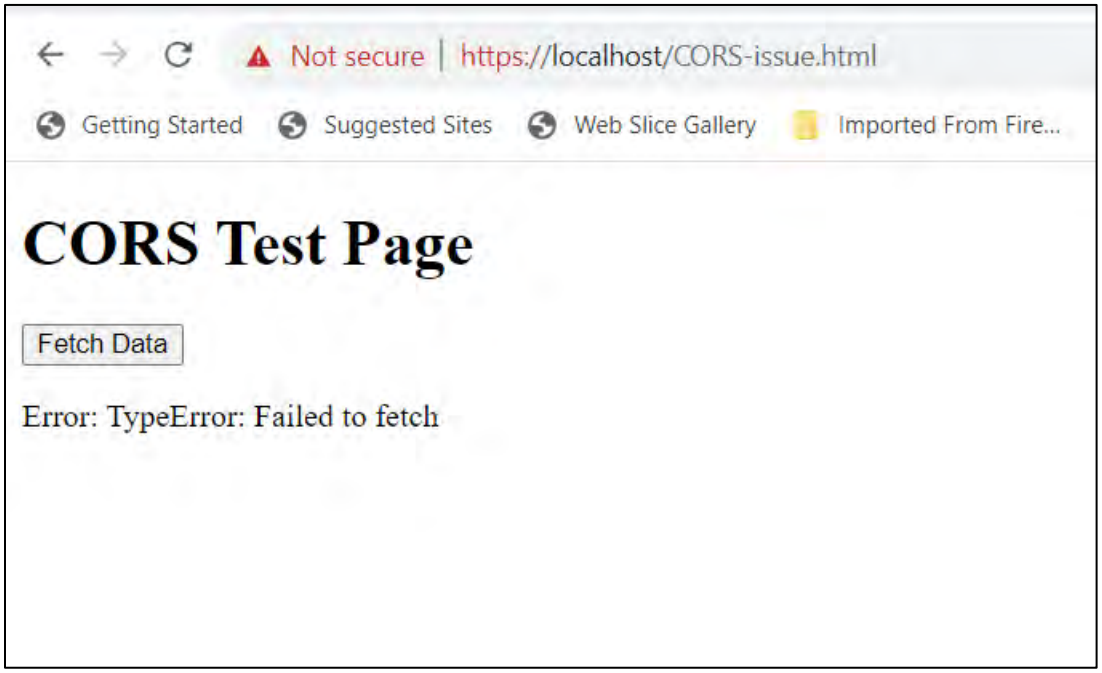


Figure 8.4: CORS-Test Page

To see the actual error, right-click the Web page and select **Inspect** → **Console** and the error will be seen there, as shown in Figure 8.5.

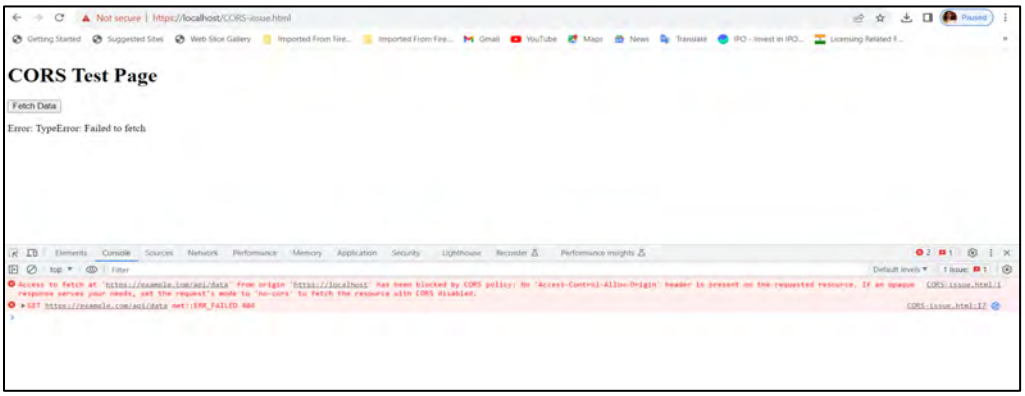


Figure 8.5: Error

The exact error message depends on the browser used, but it will typically mention the CORS policy violation and provide details about the origin and destination domains.

8.4 Implementing the CORS Policy in Browser

The CORS protocol is exclusively implemented and enforced by Web browsers. It is achieved by transmitting a set of CORS headers to the cross-origin server, which, in turn, responds with specific header values. Depending on the header values received in the response from the cross-origin server, the browser grants access to the response. It restricts the same, signaling a CORS error in the browser console when access is denied.

Using the header-based Protocol of CORS

When a Web page initiates a request to fetch a resource, the browser discerns whether the request is destined for the origin server or a cross-origin server. If it is for the latter, the browser applies the CORS policy.

The browser includes a header called 'Origin' in the request sent to the cross-origin server. The cross-origin server processes this request and responds with a header called 'Access-Control-Allow-Origin' in its response.

Subsequently, the browser scrutinizes the value of the 'Access-Control-Allow-Origin' header within the response. It will only render the response if the value of this header matches the 'Origin' header sent in the initial request.

Furthermore, the cross-origin server can employ wildcard characters, such as '*', as the value of the 'Access-Control-Allow-Origin' header. This wildcard value denotes a partial match with the 'Origin' header's value received in the request.

CORS Failures

CORS failures result in errors, but the browser deliberately withholds specific error details from being readily accessible for security reasons. This precaution is taken to prevent potential attackers from gaining insights from error messages that could be used to craft more effective subsequent attacks.

As a result, the primary means of accessing information about the error is by inspecting the browser's console, where details of the error are typically presented in a specific format:

Access to XMLHttpRequest at 'http://localhost:8000/orders' from the origin

'http://localhost:9000' has been blocked by CORS policy:

No 'Access-Control-Allow-Origin' header is present on the requested resource.

The error messages displayed in the browser console are often accompanied by a specific ‘reason’ message. It is important to note that the content and format of these ‘reasons' messages vary from one browser to another due to differences in implementation. Programmers can examine the error ‘reason’ messages specific to the Firefox browser to gain insights into some of the reasons behind CORS errors.

To avoid the CORS policy issue, change the configuration of server as follows:

1. Open the httpd.conf file located at path C:/xampp/apache/conf.
2. Remove the # before `LoadModule headers_module modules/mod_headers.so` if it exists to, enable the mod_headers.
3. Locate the httpd-vhosts.conf file under the extra folder.
4. Add following lines within the virtual host definition:

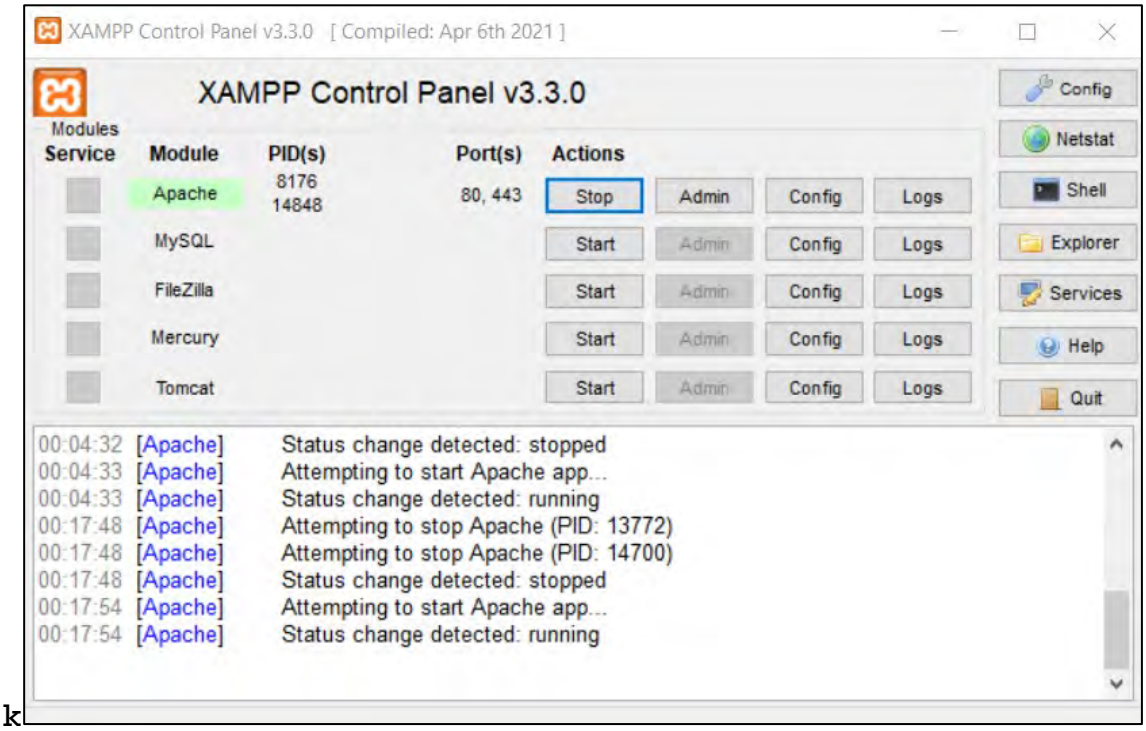
```
<VirtualHost *:80>
    ServerName localhost.local
    DocumentRoot "CORS-issue"

    # Enable CORS
    <Directory "CORS-issue">
        Header set Access-Control-Allow-Origin "*"
        Header set Access-Control-Allow-Methods "GET, POST, PUT,
DELETE, OPTIONS"
        Header set Access-Control-Allow-Headers "Content-Type,
Authorization"
    </Directory>

    # Other virtual host configurations
</VirtualHost>
```

5. Save and restart the Apache Server using the XAMPP Control Panel, as shown in Figure 8.6.

Note: Steps 3 and 4 are optional; try to restart the Apache Server after change of httpds.conf file. If the CORS issue still persists, then change the Virtual host configuration.



k

Figure 8.6: XAMPP Control Panel

Remember that enabling CORS with the wildcard `"*"` for the `Access-Control-Allow-Origin header` (`Header set Access-Control-Allow-Origin "*"`), as shown in the example, allows requests from any origin. In a production environment, the programmer should specify allowed origins for better security.

Once the settings are done, the programmer will get the response on clicking Fetch Data, as shown in Figure 8.7.

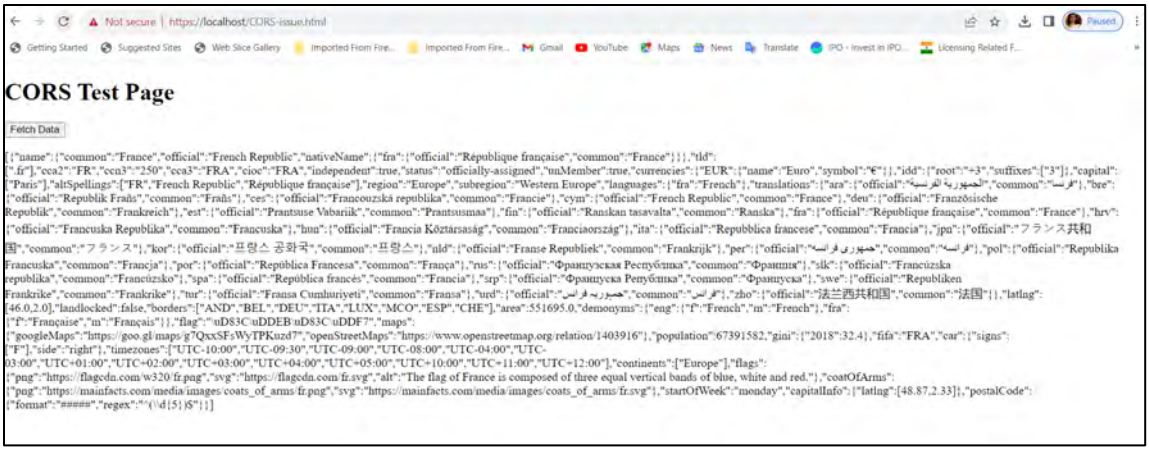


Figure 8.7: Enabling CORS

8.5 Types of CORS Request Sent by a Browser

The type of request sent by the browser to the cross-origin server depends on the programmer's intended operations with the cross-origin resource. There are three primary types of requests that the browser can send:

- Simple Requests
- Preflight Requests
- Requests with Credentials

Simple CORS Requests (GET, POST, and HEAD)

Simple requests, as per the browser's definition, include safe operations such as GET requests for data retrieval or HEAD requests for status checking.

The browser categorizes a request based on the adhered conditions:

- The HTTP request method is GET, POST, or HEAD.
- The HTTP request includes one of following CORS safe-listed headers: Accept, Accept-Language, Content-Language, or Content-Type.
- If the HTTP request contains the Content-Type header, its value must be one of following: application/x-www-form-urlencoded, multipart/form-data, or text/plain.
- There are no event listeners registered on any XMLHttpRequestUpload object.
- The request does not involve the use of a ReadableStream object.

The browser sends a simple request as a standard request, similar to a Same-Origin request, with the addition of the Origin header. Upon receiving the response, the browser checks the Access-Control-Allow-Origin header. Access to the response content is only permitted if the Access-Control-Allow-Origin header's value matches the Origin header sent in the initial request. The Origin header contains the source origin of the request.

In summary, simple requests are treated as safe operations by the browser. They are processed in a manner that ensures the security of cross-origin requests by validating the Access-Control-Allow-Origin header against the Origin header.

Preflight Requests

In contrast to simple requests, preflight requests are dispatched by the browser. This happens when you plan to modify something on a cross-origin server by using HTTP PUT to update or HTTP DELETE to remove. These requests are not considered safe, so the Web browser takes precautions by initiating a preflight request before sending the request to the cross-origin server. Additionally, requests that do not meet the criteria for simple requests also fall into this category.

The preflight request, performed automatically by the browser, employs the HTTP OPTIONS method. The primary purpose is to verify whether the cross-origin server will authorize the forthcoming actual request. Alongside the preflight request, the browser includes following headers:

Access-Control-Request-Method: This header specifies the HTTP method used when making the request.

Access-Control-Request-Headers: This header is a list of headers that will be transmitted along with the request, encompassing any custom headers.

Origin: Similar to the simple request, this header contains the source origin of the request.

The actual request to the cross-origin server will only be dispatched if the outcome of the OPTIONS method indicates that the request is permissible.

Once the preflight request is successfully concluded and the cross-origin server authorizes the request, the actual PUT method is sent with the necessary CORS headers.

CORS Requests with Credentials

In many real-world scenarios, requests sent to a cross-origin server require the inclusion of access credentials, which would be in the form of an Authorization header or cookies. By default, CORS requests are made without including these credentials.

However, when credentials are included in the request to the cross-origin server, the browser imposes a restriction. It will only grant access to the response if the cross-origin server sends a CORS header called ‘Access-Control-Allow-Credentials’ with a value of ‘true’. It is a security measure to ensure that sensitive information is handled carefully during cross-origin requests.

8.6 Summary

- CORS is a Web security protocol implemented by browsers to protect Web sites from unauthorized access.
- CORS enables secure access to resources from different domains, facilitating Web development and integration.
- Benefits of CORS include heightened security, inter-domain communication, enhanced user experience, API utilization, data sharing, and efficient development and testing.
- 'Origin' in CORS is defined by URI scheme, hostname, and port number, distinguishing between same-origin and cross-origin requests.
- Browsers enforce the CORS policy; headers are crucial in determining access to cross-origin resources.
- CORS requests include simple requests (GET, POST, and HEAD), preflight requests, and requests with credentials, each with specific requirements and security considerations.

8.7 Check Your Progress

1. What does CORS stand for in Web development?

A.	Cross-Origin Resource Search	C.	Cross-Origin Resource Sharing
B.	Cross-Origin Request Sharing	D.	Cross-Origin Request Security

2. Which community is responsible for defining the CORS protocol as part of the Fetch standard?

A.	W3C	C.	WHATWG
B.	IETF	D.	ECMA International

3. When does a CORS-preflight request occur?

A.	When the request is for an HTML page	C.	When cookies are involved
B.	When requests involve complex operations	D.	When the server does not support CORS

4. What does the Same-Origin Policy (SOP) primarily authorize a browser to do?

A.	Block Cell Request	C.	Retrieve resources from any origin
B.	Retrieve resources from Cross-Region Servers	D.	Retrieve resources from the same origin

5. What is an ‘Origin’ composed of in a CORS context?

A.	IP address and port	C.	URI scheme, path, and query parameters
B.	URI scheme, hostname, and port	D.	Domain name and cookies

8.7.1 Answers for Check Your Progress

Question	Answer
1	C
2	C
3	B
4	D
5	B

Try It Yourself

Task 1: Write a program in a programming language (Example: JavaScript) to send a simple CORS request (Example: a GET request) to a remote cross-origin server. Inspect the network logs to observe the headers in the response, including the 'Origin' header and the 'Access-Control-Allow-Origin' header.

Task 2: Create a program in your preferred programming language (Example: Python, JavaScript) to initiate a preflight CORS request (Example: an HTTP OPTIONS request) to a simulated cross-origin server. Include the necessary headers such as 'Access-Control-Request-Method' and 'Access-Control-Request-Headers'. Analyze the preflight request and the subsequent actual request to understand how CORS handles requests with changes and custom headers.